- Azure Media Services CDSA CPS audit certificate
- Azure Media Services CDSA CPS audit report

Extra guidance documentation is available from the Service Trust Portal Media and Entertainment section:

- Azure CDSA CPS implementation guide

Frequently asked questions

What is the significance of Azure Media Services CDSA certification?
Since launching the TPN, CDSA ceased its own security assessment program to focus on managing and developing a unified assessment program through the TPN together with the MPA. However, Azure Media Services was the first hyper-scale cloud services media platform certified to comply with the CDSA CPS standard while the CDSA assessment program was operational. This certification demonstrates that the intellectual property rights of media assets stored, managed, and distributed from within Azure are protected according to a well-respected industry standard.

Moreover, the Azure CDSA CPS implementation guidance provides valuable insight into Azure capabilities aimed at protecting the intellectual property rights of media assets stored, managed, and distributed from within Azure. It describes how Azure Media Services can help you create CDSA CPS compliant solutions securely and how you can create, protect, and operate digital media services on Azure.

Does my organization still need to undergo a TPN assessment?
Production facilities, visual effects houses, and other service partners should work with their executive producers and directors to understand the new security requirements, including the annual TPN assessment. You can hire a qualified TPN assessor and then manage your assessment process using the secure online platform. The TPN assessment does not provide a "pass/fail" grade, certification, or rating. It provides an assessment of a facility's security preparedness for conformance with the MPA content security best practices. If an assessment indicates non-conformance with a security best practices control, you can validate remediation via follow-up assessment or furnish your own evidence of remediation to the TPN.

Since launching the TPN, both the MPA and CDSA have ceased their individual security assessment programs to focus on managing and developing the TPN program and TPN annual assessments. Past audits or assessments will remain valid for the period originally indicated but will not be renewable within their individual programs. For both the MPA and CDSA, the primary focus is to provide a unified assessment program through the TPN.

Resources

- Azure compliance documentation
- Azure enables a world of compliance
- Microsoft 365 compliance offerings
- Compliance on the Microsoft Trust Center
- Microsoft Intelligent Media and Entertainment
- Content Delivery and Security Association (CDSA)
- Motion Picture Association (MPA)
- Trusted Partner Network (TPN)
- Azure Virtual Desktop infrastructure hardening guide by Independent Security Evaluators (ISE)
- Editorial and asset management hardening guide for Azure by Independent Security Evaluators (ISE)
- Azure media hardening guide for 3D graphics rendering by Independent Security Evaluators (ISE)

Media Services regulatory compliance

Article • 01/10/2023

Media Services API v3

> ⚠ **Warning**
>
> Azure Media Services will be retired June 30th, 2024. For more information, see the **AMS Retirement Guide**.

Media Services meets the demanding requirements of the US Federal Risk & Authorization Management Program (FedRAMP) and of the US Department of Defense (DoD) Cloud Computing Security Requirements Guide (SRG) Impact Level (IL) 2, IL4, and IL5. By deploying authorized services in Azure Government, Office 365 GCC High and DoD, and Dynamics 365 US Government, federal and defense agencies can use a rich array of cloud services while meeting their compliance obligations.

FedRAMP and DoD compliance

Media Services in Azure Public maintains:

- FedRAMP High Provisional Authorization to Operate (P-ATO)
- DoD IL2 Provisional Authorization (PA)

Media Services in Azure Government maintains:

- FedRAMP High P-ATO
- DoD IL2 PA
- DoD IL4 PA
- DoD IL5 PA

For more information about Azure compliance coverage for US government, see Azure FedRAMP High, DoD IL2, DoD IL4, and DoD IL5 documentation. For FedRAMP and DoD audit scope, see Cloud services by audit scope.

Azure compliance documentation

To help you meet your own compliance obligations across regulated industries and markets worldwide, Azure maintains the largest compliance portfolio in the industry

both in terms of breadth (total number of compliance offerings) and depth (number of customer-facing services in assessment scope). For service availability, see Products available by region.

Azure compliance offerings are grouped into four segments - globally applicable, US Government, industry specific, and region/country specific. Compliance offerings are based on various types of assurances, including formal certifications, attestations, validations, authorizations, and assessments produced by independent third-party auditing firms, as well as contractual amendments, self-assessments, and customer guidance documents produced by Microsoft. For more information, see Azure compliance documentation. You will also find additional compliance resources such as audit reports, a checklist for privacy and General Data Protection Regulation (GDPR), compliance blueprints, country and regional guidelines, implementation and mappings, as well as white papers and analyst reports.

Get help and support

You can contact Media Services with questions or follow our updates by one of the following methods:

- Q & A
- Stack Overflow. Tag questions with `azure-media-services`.
- @MSFTAzureMedia or use @AzureSupport to request support.
- Open a support ticket through the Azure portal.

Azure Media Services compliance, privacy and security

Article • 01/10/2023

Media Services API v3

> ⚠ **Warning**
>
> Azure Media Services will be retired June 30th, 2024. For more information, see the **AMS Retirement Guide**.

Compliance, privacy and security

As an important reminder, you must comply with all applicable laws in your use of Azure Media Services, and you may not use Media Services or any Azure service in a manner that violates the rights of others, or that may be harmful to others.

Before uploading any video/image to Media Services, You must have all the proper rights to use the video/image, including, where required by law, all the necessary consents from individuals (if any) in the video/image, for the use, processing, and storage of their data in Media Services and Azure. Some jurisdictions may impose special legal requirements for the collection, online processing and storage of certain categories of data, such as biometric data. Before using Media Services and Azure for the processing and storage of any data subject to special legal requirements, You must ensure compliance with any such legal requirements that may apply to You.

Learn more about compliance

To learn about compliance, privacy and security in Media Services please visit the Microsoft Trust Center. For Microsoft's privacy obligations, data handling and retention practices, including how to delete your data, please review Microsoft's Privacy Statement, the Online Services Terms ("OST") and Data Processing Addendum ("DPA"). By using Media Services, you agree to be bound by the OST, DPA and the Privacy Statement.

> ⓘ **Note**

Media services will be enforcing TLS 1.2 for all requests to KeyDelivery, RESTv2, Streaming Endpoint and Live Event streaming origins. Accounts with existing TLS 1.0 or 1.1 usage will be exempt from this enforcement. If you wish to enforce TLS 1.2 for all your requests to these media services endpoints, please contact AMS support.

Get help and support

You can contact Media Services with questions or follow our updates by one of the following methods:

- Q & A
- Stack Overflow. Tag questions with `azure-media-services`.
- @MSFTAzureMedia or use @AzureSupport to request support.
- Open a support ticket through the Azure portal.

Motion Picture Association (MPA)

Article • 03/21/2023

MPA overview

The Motion Picture Association (MPA) provides content protection best practices and control frameworks to help major studio partners and vendors design infrastructure and solutions to ensure the security of digital film assets.

The Trusted Partner Network (TPN) is a new, global, industry-wide film and television content production initiative. It was launched in 2018 by MPA and the Content Delivery & Security Association (CDSA), the worldwide leaders in third-party entertainment industry assessments. The TPN program helps companies prevent leaks, breaches, and hacks of movies and television shows prior to their intended release. TPN is owned and managed by MPA. TPN has been developed to help the industry improve content security, simplify assessments, and enable content owners to gauge their level of conformance to the MPA content security best practices.

> **Note**
>
> Since launching the TPN, MPA has ceased their individual security assessment programs to focus on managing and developing the TPN program and TPN annual assessments. Past audits or assessments will remain valid for the period originally indicated but won't be renewable within their individual programs. The primary focus for MPA is to provide a unified assessment program through the TPN.

MPA continues to maintain and update their content security best practices. A TPN assessment doesn't provide a “pass/fail” grade, certification, or rating. It provides an assessment of a facility’s security preparedness for conformance with the MPA content security best practices.

Azure and MPA

In February 2016, Microsoft Azure became the first hyper-scale, multi-tenant cloud services platform to successfully complete a formal assessment by independent MPA auditors, and comply with all three of the MPA content security best practices frameworks: Common, Application, and Cloud Security Guidelines.

The MPA assessment covers 48 security topics in the Common Guidelines and an additional six in the Application and Cloud Security Guidelines. These topics are built on industry-accepted security standards such as ISO/IEC 27001 and NIST SP 800-53, and are aligned to industry best practices, such as the Cloud Security Alliance (CSA) Cloud Controls Matrix (CCM).

The formal assessment of Azure compliance means that companies who do business with major studios can use Azure to help reduce the IT costs that are normally associated with the secure creation, management, storage, and distribution of content while complying with MPA requirements. Azure Media Services, Storage, Virtual Network, and more than 30 other services provide a content workflow engine in the cloud that you can use to build secure and scalable production processes while protecting media assets downstream.

Azure has released guidance documentation to help you implement your solutions that meet MPA security best practices.

Guidance documents

For instructions on how to access audit reports and certificates, see Audit documentation. You can download the following guidance documents from the Service Trust Portal (STP) Media and Entertainment section:

- Azure responses to MPA common guidelines
- Azure responses to MPA application and cloud security guidelines

Extra guidance documents are available:

- Azure Virtual Desktop infrastructure hardening guide
- Editorial and asset management hardening guide for Azure
- Azure media hardening guide for 3D graphics rendering

Frequently asked questions

Why are the MPA best practices important?
Content security is critical for feature film development as there are multiple points along the workflow where digital assets could be compromised or stolen. Dailies, rough cuts, and visual effects are just some of the materials exposed during a normal production cycle, and the box office impact of a security breach on a blockbuster project can reach tens of millions of dollars.

MPA guidelines provide major studio vendors and partners with a set of best practices for creating, processing, storing, and distributing digital assets. Cloud service platforms such as Azure can provide an extra layer of assurance that content uploaded to the cloud will be managed in accordance with established industry requirements for encryption, authentication, access control, resiliency, and others.

Does my organization still need to undergo a TPN assessment, or can we rely on Azure MPA assessments?
Production facilities, visual effects houses, and other service partners should work with their executive producers and directors to understand the new security requirements, including the annual TPN assessment. You can hire a qualified TPN assessor and then manage your assessment process using the secure online platform. The TPN assessment doesn't provide a “pass/fail” grade, certification, or rating. It provides an assessment of a facility’s security preparedness for conformance with the MPA content security best practices. If an assessment indicates non-conformance with a security best practices control, you can validate remediation via a follow-up assessment or furnish your own evidence of remediation to TPN.

Compliance with MPA content protection best practices is voluntary – MPA doesn't provide an accreditation program. Best practices outline security expectations and provide a framework for assessing facility's ability to protect content. Microsoft elected to carry out an independent MPA assessment so that you can be confident in the content security and protection capabilities of Azure. However, Microsoft doesn't inspect, approve, or manage your applications deployed on Azure. You're wholly responsible for ensuring your own compliance with all applicable laws and regulations.

Resources

- Azure compliance documentation
- Azure enables a world of compliance
- Microsoft 365 compliance offerings
- Compliance on the Microsoft Trust Center
- Microsoft Intelligent Media and Entertainment
- Motion Picture Association (MPA)
- MPA content protection best practices
- Content Delivery and Security Association (CDSA)
- Trusted Partner Network (TPN)
- Avid video editorial in Azure
- Adobe Creative Cloud best practices in Azure Virtual Machines
- Secure Reliable Transport (SRT) Hub for broadcast workloads

- Azure Virtual Desktop infrastructure hardening guide by Independent Security Evaluators (ISE)
- Editorial and asset management hardening guide for Azure by Independent Security Evaluators (ISE)
- Azure media hardening guide for 3D graphics rendering by Independent Security Evaluators (ISE)

Solutions for the travel and hospitality industry

Article • 08/31/2023

Travel and hospitality are trillion-dollar industries in the United States alone. Travel and hospitality companies are investing in newer products and technologies, looking for that competitive edge. Cloud technology offers innovative, affordable, and versatile solutions. With integrated solutions, services, and templates, Azure gives developers and IT professionals the ability to create web solutions for the Internet of Things (IoT) and mobile devices.

Azure has the flexibility and scalability to handle the challenges that are faced in the travel and hospitality industries. Azure offers the following capabilities:

- Create hybrid environments by integrating your existing IT environment with the cloud, through secure private connections, hybrid databases, and storage solutions. Keep your assets where you want them.
- Increase or lower storage volume based on seasonal fluctuations and high-demand events. Scale to the demands of your travelers.
- Keep data secure and safe through encrypted solutions, firewalls, backup recovery, and redundancy.
- Build, deploy, and manage apps. Customize your cloud software to meet your needs.
- Improve customer service and uncover new business opportunities by using built-in support for analyzing data and key insights.

The following video shows how Azure AI can optimize your guest experience.

https://www.youtube-nocookie.com/embed/EHNBvshgNbA

Architectures for travel and hospitality

The following articles provide detailed analysis of architectures created and recommended for the travel and hospitality industry.

Architecture	Summary	Technology focus
Build a delta lake to support ad hoc queries in online leisure	Make raw data easily accessible. This data is important to technical and customer support	Databases

Architecture	Summary	Technology focus
and travel booking	teams, data engineers, and legal teams.	
Custom business processes for airlines	Monitor customer flight data, when a traveler selects or is assigned a flight.	Integration
Data science and machine learning with Azure Databricks	Improve efficiency, enhance customer experiences, and predict changes in your business with data science and machine learning.	AI
Migrate a web app using Azure API Management	Migrate from your legacy web applications by using Azure API Management.	Web
Predictive aircraft engine monitoring	Combine real-time aircraft data with analytics to create a solution for predictive aircraft engine monitoring and health.	Analytics

Solution ideas for travel and hospitality

The following are other ideas that you can use as a starting point for your travel and hospitality solution.

- Cognizant Safe Buildings with IoT and Azure
- Data science and machine learning with Azure Databricks
- Facilities management powered by mixed reality and IoT

Quickstart: Create a search index in the Azure portal

Article • 11/30/2023

In this Azure AI Search quickstart, create your first *search index* by using the **Import data** wizard and a built-in sample data source consisting of fictitious hotel data hosted by Microsoft. The wizard guides you through the creation of a no-code search index to help you write interesting queries within minutes.

The wizard creates multiple objects on your search service - searchable index - but also an indexer and data source connection for automated data retrieval. At the end of this quickstart, we review each object.

> **Note**
>
> The **Import data** wizard includes options for OCR, text translation, and other AI enrichments that aren't covered in this quickstart. For a similar walkthrough that focuses on AI enrichment, see **Quickstart: Create a skillset in the Azure portal**.

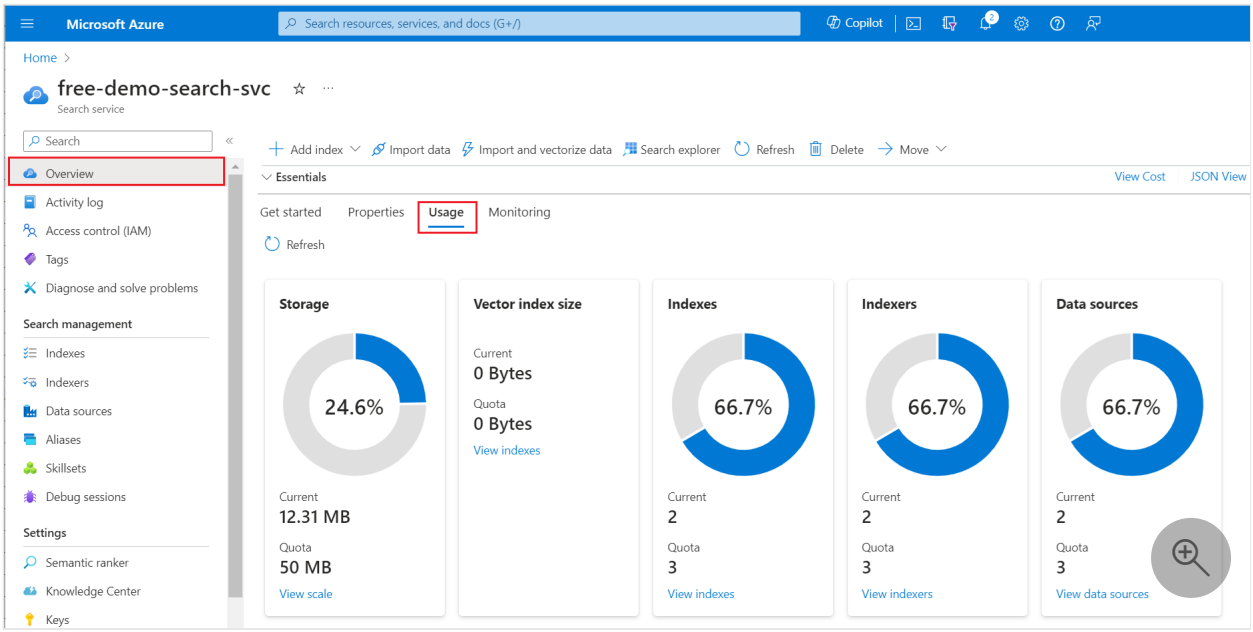
Prerequisites

- An Azure account with an active subscription. Create an account for free.
- An Azure AI Search service for any tier and any region. Create a service or find an existing service under your current subscription. You can use a free service for this quickstart.

Check for space

Many customers start with the free service. The free tier is limited to three indexes, three data sources, and three indexers. Make sure you have room for extra items before you begin. This quickstart creates one of each object.

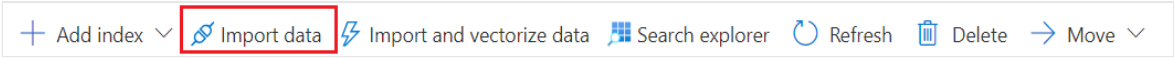
Check the **Overview** > **Usage** tab for the service to see how many indexes, indexers, and data sources you already have.



Start the wizard

1. Sign in to the Azure portal with your Azure account, and go to your Azure AI Search service.

2. On the **Overview** page, select **Import data** to start the wizard.



Create and load an index

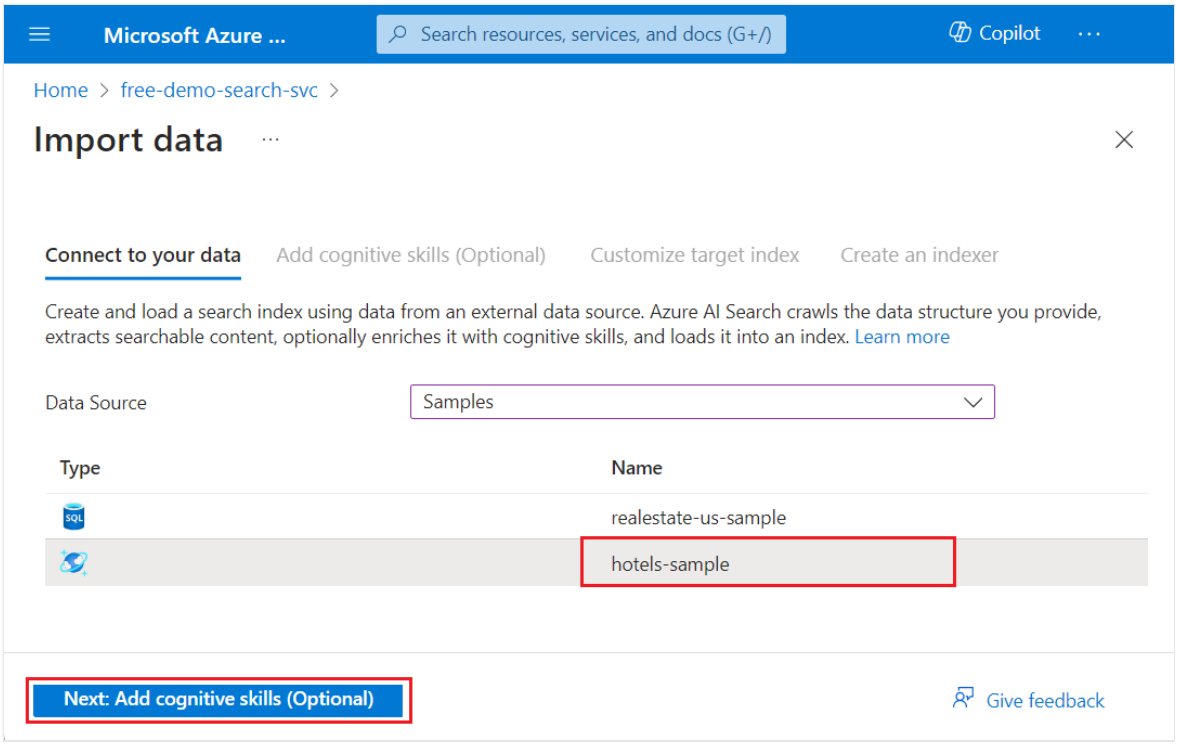
In this section, create and load an index in four steps.

Connect to a data source

The wizard creates a data source connection to sample data hosted by Microsoft on Azure Cosmos DB. This sample data is retrieved accessed over an internal connection. You don't need your own Azure Cosmos DB account or source files to run this quickstart.

1. On **Connect to your data**, expand the **Data Source** dropdown list and select **Samples**.

2. In the list of built-in samples, select **hotels-sample**.



3. Select **Next: Add cognitive skills (Optional)** to continue.

Skip configuration for cognitive skills

The **Import data** wizard supports the creation of a skillset and AI-enrichment into indexing.

1. For this quickstart, ignore the AI enrichment configuration options on the **Add cognitive skills** tab.

2. Select **Skip to: Customize target index** to continue.

> **Tip**
>
> Interested in AI enrichment? Try this **Quickstart: Create a skillset in the Azure portal**

Configure the index

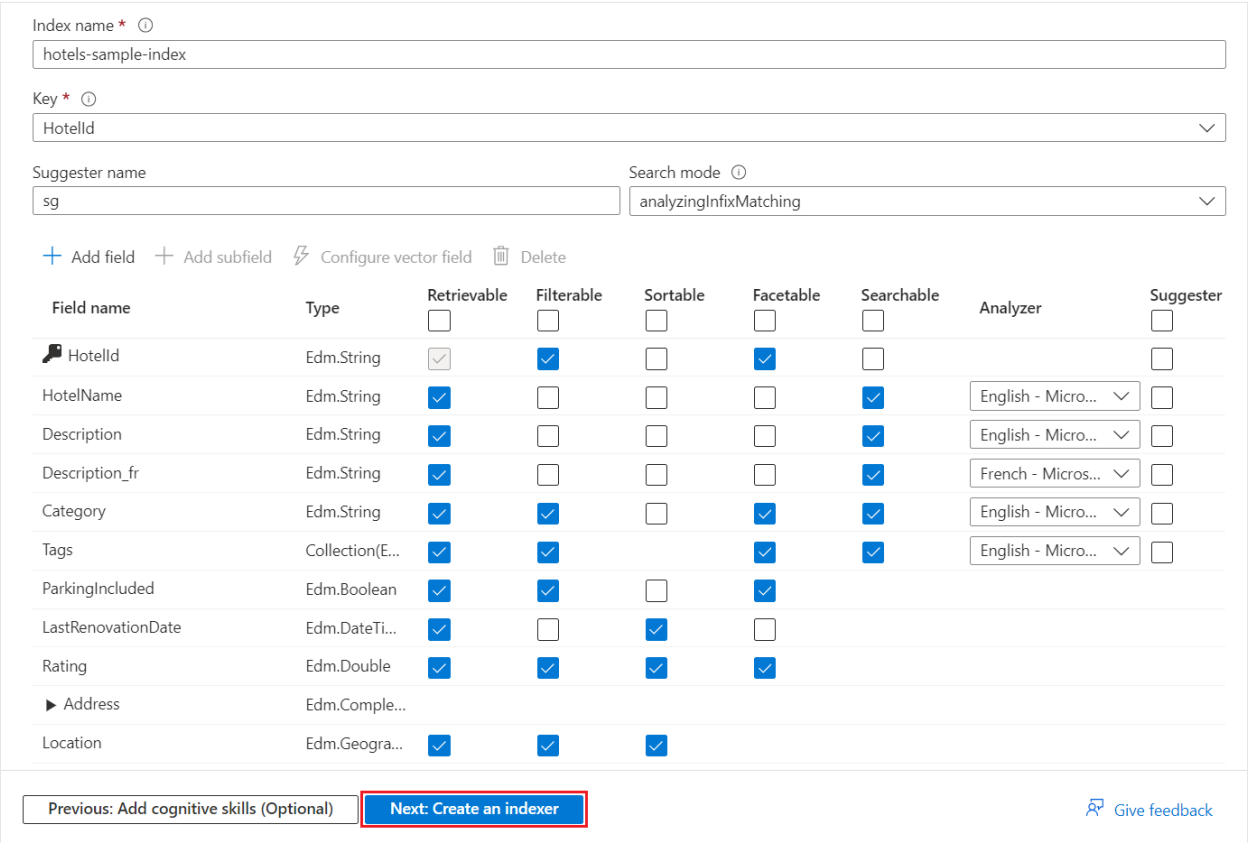
The wizard infers a schema for the built-in hotels-sample index. Follow these steps to configure the index:

1. Accept the system-generated values for the **Index name** (*hotels-sample-index*) and **Key** field (*HotelId*).

2. Accept the system-generated values for all field attributes.

> **Important**
>
> If you rerun the wizard and use an existing hotels-sample data source, the index isn't configured with default attributes. You have to manually select attributes on future imports.

3. Select **Next: Create an indexer** to continue.



At a minimum, the index requires an **Index name** and a collection of **Fields**. One field must be marked as the *document key* to uniquely identify each document. The value is always a string. The wizard scans for unique string fields and chooses one for the key.

Each field has a name, data type, and *attributes* that control how to use the field in the search index. Checkboxes enable or disable the following attributes:

- **Retrievable**: Fields returned in a query response.
- **Filterable**: Fields that accept a filter expression.
- **Sortable**: Fields that accept an orderby expression.
- **Facetable**: Fields used in a faceted navigation structure.

- **Searchable**: Fields used in full text search. Strings are searchable. Numeric fields and Boolean fields are often marked as not searchable.

Strings are attributed as **Retrievable** and **Searchable**. Integers are attributed as **Retrievable**, **Filterable**, **Sortable**, and **Facetable**.

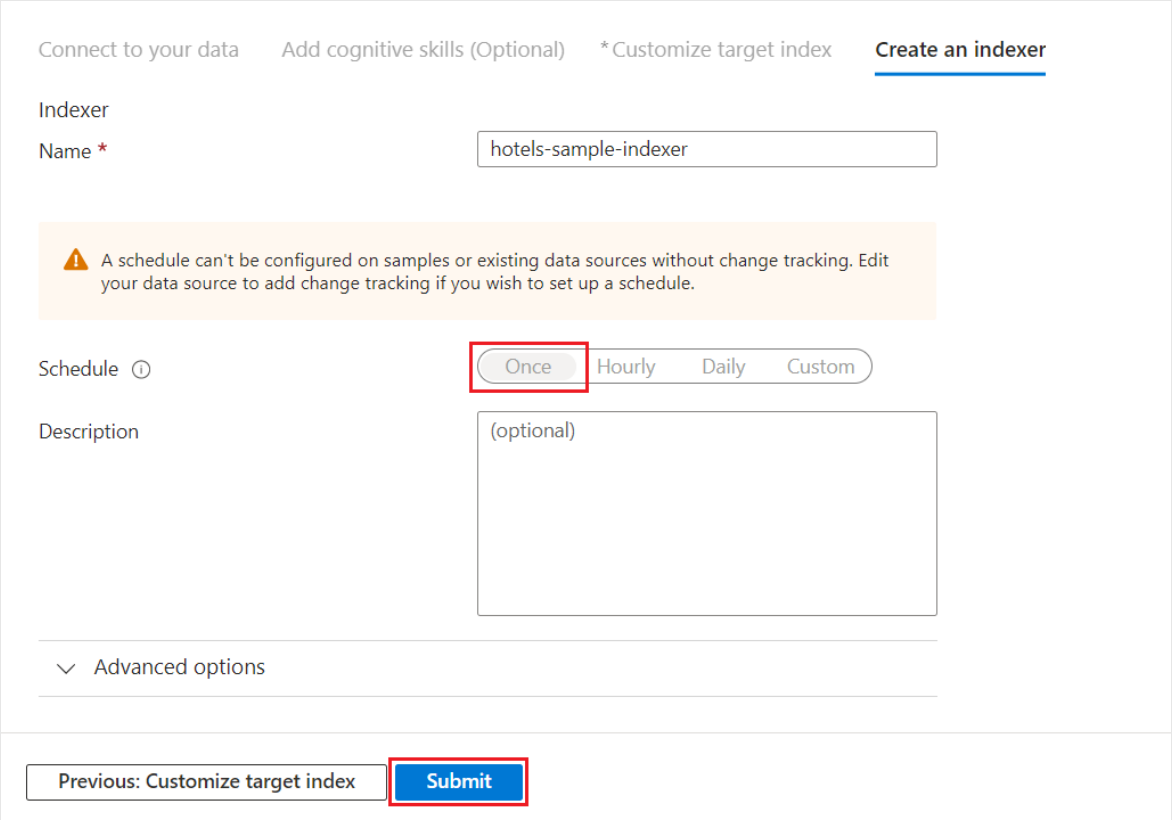
Attributes affect storage. **Filterable** fields consume extra storage, but **Retrievable** doesn't. For more information, see Example demonstrating the storage implications of attributes and suggesters.

If you want autocomplete or suggested queries, specify language **Analyzers** or **Suggesters**.

Configure and run the indexer

The last step configures and runs the indexer. This object defines an executable process. The data source, index, and indexer are created in this step.

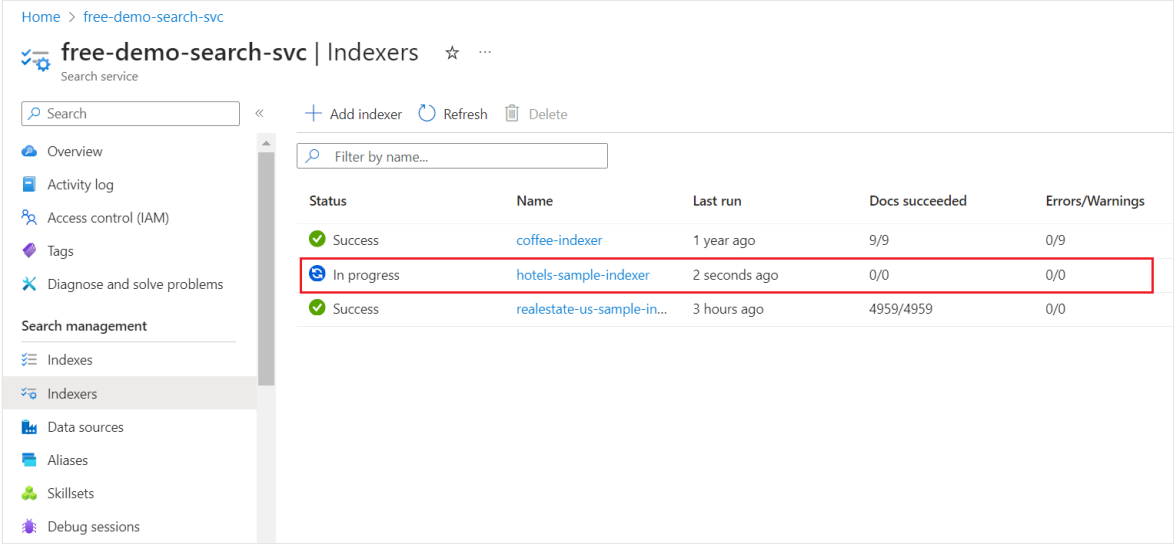
1. Accept the system-generated value for the **Indexer name** (*hotels-sample-indexer*).
2. For this quickstart, use the default option to run the indexer once, immediately. The hosted data is static so there's no change tracking enabled for it.
3. Select **Submit** to create and simultaneously run the indexer.



Monitor indexer progress

You can monitor creation of the indexer or index in the portal. The service **Overview** page provides links to the resources created in your Azure AI Search service.

1. On the left, select **Indexers**.



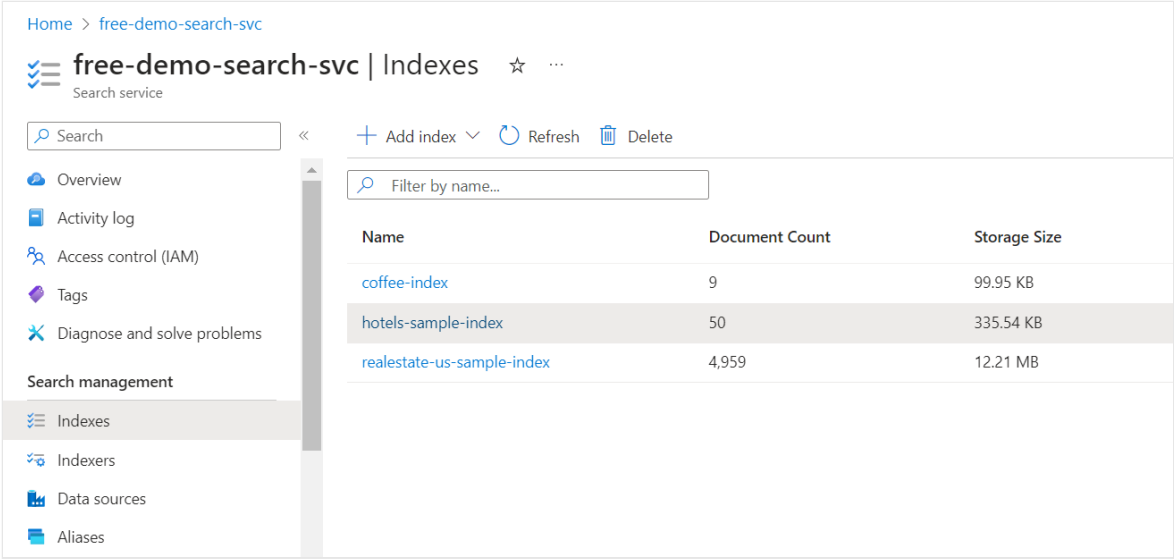
It can take a few minutes for the page results to update in the Azure portal. You should see the newly created indexer in the list with a status of *In progress* or *Success*. The list also shows the number of documents indexed.

Check search index results

1. On the left, select **Indexes**.

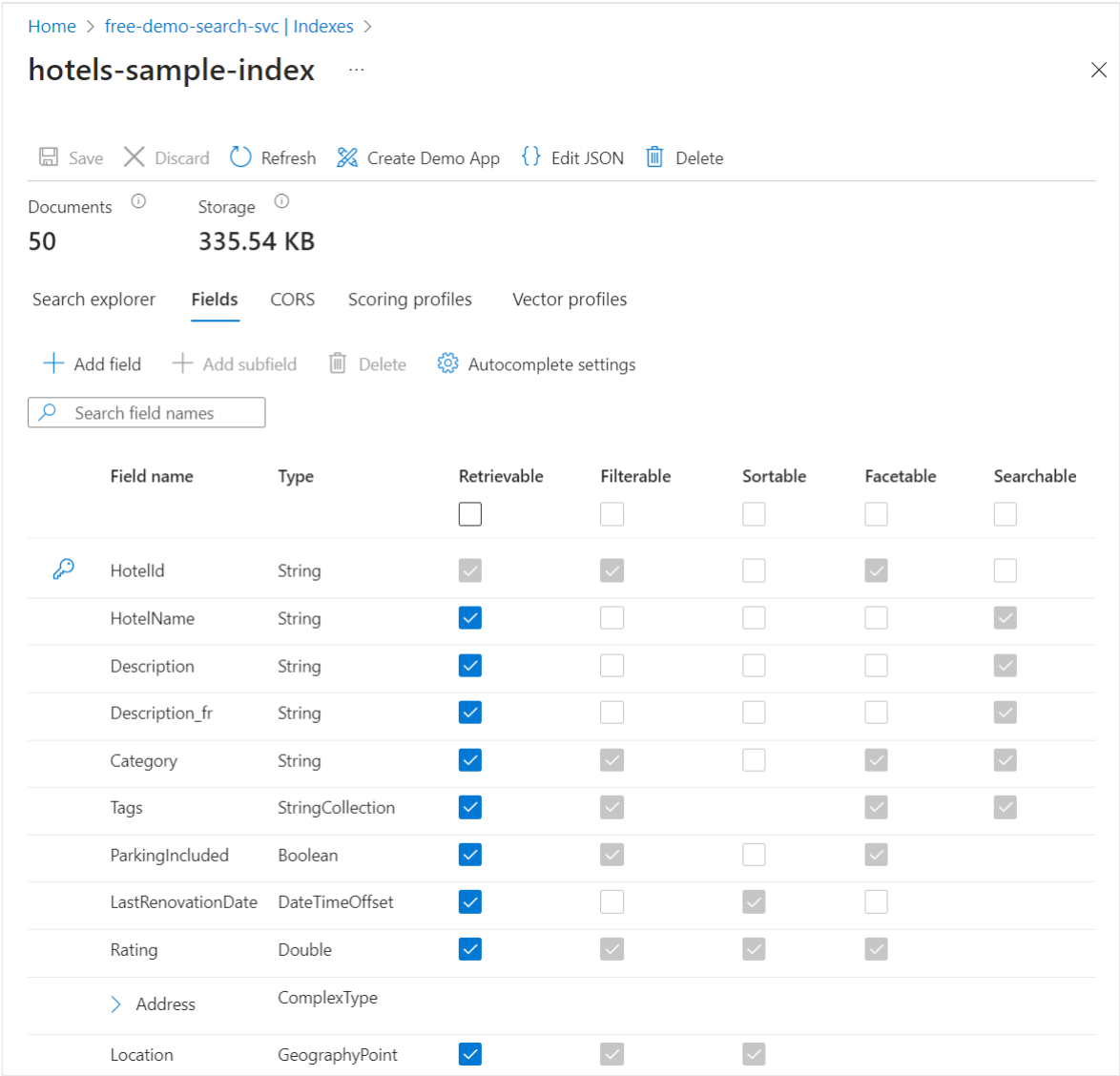
2. Select **hotels-sample-index**.

   Wait for the Azure portal page to refresh. You should see the index with a document count and storage size.



3. Select the **Fields** tab to view the index schema.

   Check to see which fields are **Filterable** or **Sortable** so that you know what queries to write.



Add or change fields

On the **Fields** tab, you can create a new field using **Add field** with a name, supported data type, and attributions.

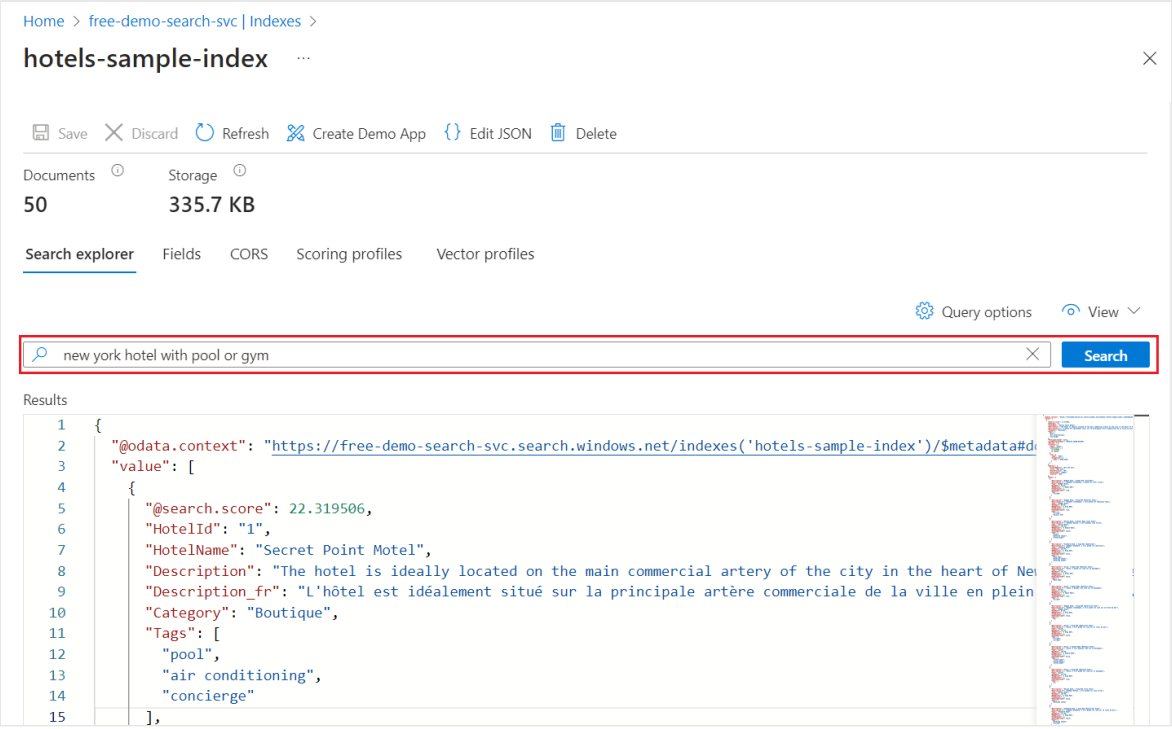
Changing existing fields is harder. Existing fields have a physical representation in the index so they aren't modifiable, not even in code. To fundamentally change an existing field, you need to create a new field that replaces the original. Other constructs, such as scoring profiles and CORS options, can be added to an index at any time.

To clearly understand what you can and can't edit during index design, take a minute to view the index definition options. Grayed options in the field list indicate values that can't be modified or deleted.

Query with Search explorer

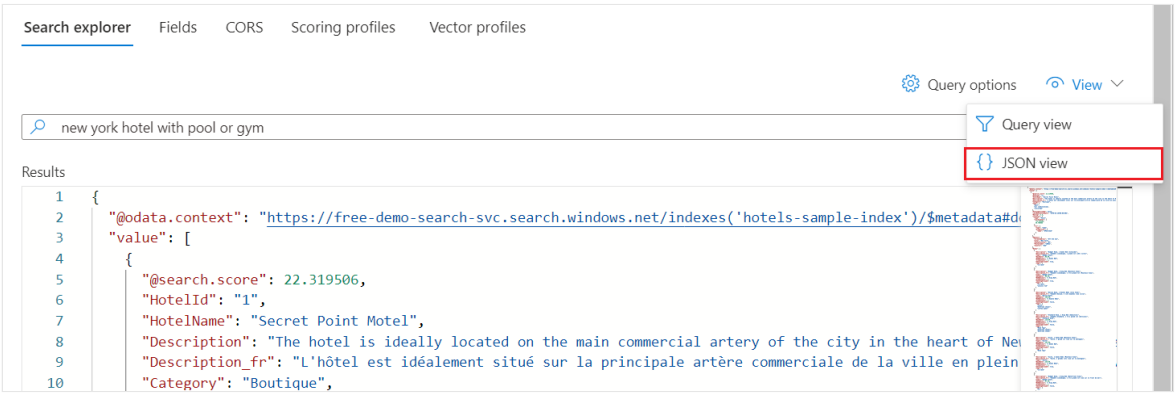
You now have a search index that can be queried with **Search explorer**. **Search explorer** sends REST calls that conform to the Search POST REST API. The tool supports simple query syntax and full Lucene query syntax.

1. On the **Search explorer** tab, enter text to search on.



2. Use the **Mini-map** to jump quickly to nonvisible areas of the output.

3. To specify syntax, switch to the JSON view.



Example queries for hotels sample index

The following examples assume the JSON view and the 2023-11-01 REST API version.

Filter examples

Parking, tags, renovation date, rating and location are filterable.

JSON

```
{
    "search": "beach OR spa",
    "select": "HotelId, HotelName, Description, Rating",
    "count": true,
    "top": 10,
    "filter": "Rating gt 4"
}
```

Boolean filters assume "true" by default.

JSON

```
{
    "search": "beach OR spa",
    "select": "HotelId, HotelName, Description, Rating",
    "count": true,
    "top": 10,
    "filter": "ParkingIncluded"
}
```

Geospatial search is filter-based. The `geo.distance` function filters all results for positional data based on the specified `Location` and `geography'POINT` coordinates. The query seeks hotels that are within 5 kilometers of the latitude longitude coordinates `-122.12 47.67`, which is "Redmond, Washington, USA." The query displays the total number of matches `&$count=true` with the hotel names and address locations.

JSON

```
{
    "search": "*",
    "select": "HotelName, Address/City, Address/StateProvince",
    "count": true,
    "top": 10,
    "filter": "geo.distance(Location, geography'POINT(-122.12 47.67)') le 5"
}
```

Full Lucene syntax examples

The default syntax is simple syntax, but if you want fuzzy search or term boosting or regular expressions, specify the full syntax.

JSON

```
{
    "queryType": "full",
    "search": "seatle~",
    "select": "HotelId, HotelName,Address/City, Address/StateProvince",
    "count": true
}
```

By default, misspelled query terms like `seatle` for `Seattle` fail to return matches in a typical search. The `queryType=full` parameter invokes the full Lucene query parser, which supports the tilde `~` operand. When these parameters are present, the query performs a fuzzy search for the specified keyword. The query seeks matching results along with results that are similar to but not an exact match to the keyword.

Take a minute to try a few of these example queries for your index. To learn more about queries, see Querying in Azure AI Search.

Clean up resources

When you work in your own subscription, it's a good idea at the end of a project to identify whether you still need the resources you created. Resources left running can cost you money. You can delete resources individually or delete the resource group to delete the entire set of resources.

You can find and manage resources for your service in the Azure portal under **All resources** or **Resource groups** in the left pane.

If you use a free service, remember that the limit is three indexes, indexers, and data sources. You can delete individual items in the Azure portal to stay under the limit.

Next steps

Try an Azure portal wizard to generate a ready-to-use web app that runs in a browser. Use this wizard on the small index you created in this quickstart, or use one of the built-in sample data sets for a richer search experience.

Create a demo app in the portal

Data science using Spark on Azure HDInsight

Article • 11/21/2022

This suite of topics shows how to use HDInsight Spark to complete common data science tasks such as data ingestion, feature engineering, modeling, and model evaluation. The data used is a sample of the 2013 NYC taxi trip and fare dataset. The models built include logistic and linear regression, random forests, and gradient boosted trees. The topics also show how to store these models in Azure blob storage (WASB) and how to score and evaluate their predictive performance. More advanced topics cover how models can be trained using cross-validation and hyper-parameter sweeping. This overview topic also references the topics that describe how to set up the Spark cluster that you need to complete the steps in the walkthroughs provided.

Spark and MLlib

Spark is an open-source parallel processing framework that supports in-memory processing to boost the performance of big-data analytic applications. The Spark processing engine is built for speed, ease of use, and sophisticated analytics. Spark's in-memory distributed computation capabilities make it a good choice for the iterative algorithms used in machine learning and graph computations. MLlib is Spark's scalable machine learning library that brings the algorithmic modeling capabilities to this distributed environment.

HDInsight Spark

HDInsight Spark is the Azure hosted offering of open-source Spark. It also includes support for **Jupyter PySpark notebooks** on the Spark cluster that can run Spark SQL interactive queries for transforming, filtering, and visualizing data stored in Azure Blobs (WASB). PySpark is the Python API for Spark. The code snippets that provide the solutions and show the relevant plots to visualize the data here run in Jupyter notebooks installed on the Spark clusters. The modeling steps in these topics contain code that shows how to train, evaluate, save, and consume each type of model.

Setup: Spark clusters and Jupyter notebooks

Setup steps and code are provided in this walkthrough for using an HDInsight Spark 1.6. But Jupyter notebooks are provided for both HDInsight Spark 1.6 and Spark 2.0 clusters.

A description of the notebooks and links to them are provided in the Readme.md for the GitHub repository containing them. Moreover, the code here and in the linked notebooks is generic and should work on any Spark cluster. If you are not using HDInsight Spark, the cluster setup and management steps may be slightly different from what is shown here. For convenience, here are the links to the Jupyter notebooks for Spark 1.6 (to be run in the pySpark kernel of the Jupyter Notebook server) and Spark 2.0 (to be run in the pySpark3 kernel of the Jupyter Notebook server):

Spark 1.6 notebooks

These notebooks are to be run in the pySpark kernel of Jupyter notebook server.

- pySpark-machine-learning-data-science-spark-data-exploration-modeling.ipynb : Provides information on how to perform data exploration, modeling, and scoring with several different algorithms.
- pySpark-machine-learning-data-science-spark-advanced-data-exploration-modeling.ipynb : Includes topics in notebook #1, and model development using hyperparameter tuning and cross-validation.
- pySpark-machine-learning-data-science-spark-model-consumption.ipynb : Shows how to operationalize a saved model using Python on HDInsight clusters.

Spark 2.0 notebooks

These notebooks are to be run in the pySpark3 kernel of Jupyter notebook server.

- Spark2.0-pySpark3-machine-learning-data-science-spark-advanced-data-exploration-modeling.ipynb : This file provides information on how to perform data exploration, modeling, and scoring in Spark 2.0 clusters using the NYC Taxi trip and fare data-set described here. This notebook may be a good starting point for quickly exploring the code we have provided for Spark 2.0. For a more detailed notebook analyzes the NYC Taxi data, see the next notebook in this list. See the notes following this list that compares these notebooks.
- Spark2.0-pySpark3_NYC_Taxi_Tip_Regression.ipynb : This file shows how to perform data wrangling (Spark SQL and dataframe operations), exploration, modeling and scoring using the NYC Taxi trip and fare data-set described here.
- Spark2.0-pySpark3_Airline_Departure_Delay_Classification.ipynb : This file shows how to perform data wrangling (Spark SQL and dataframe operations), exploration, modeling and scoring using the well-known Airline On-time departure dataset from 2011 and 2012. We integrated the airline dataset with the airport weather data (for example, windspeed, temperature, altitude etc.) prior to modeling, so these weather features can be included in the model.

> **Note**
>
> The airline dataset was added to the Spark 2.0 notebooks to better illustrate the use of classification algorithms. See the following links for information about airline on-time departure dataset and weather dataset:
>
> - Airline on-time departure data: **https://www.transtats.bts.gov/ONTIME/**
> - Airport weather data: **https://www.ncdc.noaa.gov/**

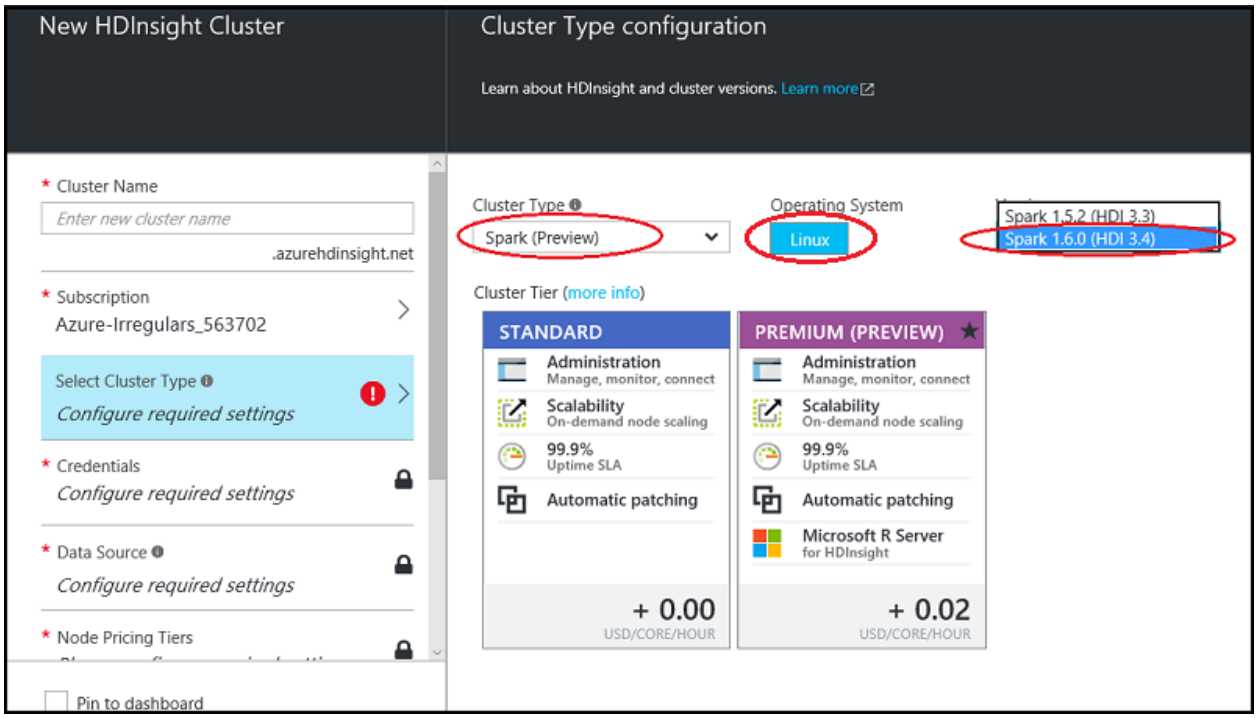
> **Note**
>
> The Spark 2.0 notebooks on the NYC taxi and airline flight delay data-sets can take 10 mins or more to run (depending on the size of your HDI cluster). The first notebook in the above list shows many aspects of the data exploration, visualization and ML model training in a notebook that takes less time to run with down-sampled NYC data set, in which the taxi and fare files have been pre-joined: **Spark2.0-pySpark3-machine-learning-data-science-spark-advanced-data-exploration-modeling.ipynb** . This notebook takes a much shorter time to finish (2-3 mins) and may be a good starting point for quickly exploring the code we have provided for Spark 2.0.

For guidance on the operationalization of a Spark 2.0 model and model consumption for scoring, see the Spark 1.6 document on consumption for an example outlining the steps required. To use this example on Spark 2.0, replace the Python code file with this file .

Prerequisites

The following procedures are related to Spark 1.6. For the Spark 2.0 version, use the notebooks described and linked to previously.

1. You must have an Azure subscription. If you do not already have one, see Get Azure free trial .
2. You need a Spark 1.6 cluster to complete this walkthrough. To create one, see the instructions provided in Get started: create Apache Spark on Azure HDInsight. The cluster type and version is specified from the **Select Cluster Type** menu.



> ⓘ **Note**
>
> For a topic that shows how to use Scala rather than Python to complete tasks for an end-to-end data science process, see the **Data Science using Scala with Spark on Azure**.

> ⚠ **Warning**
>
> Billing for HDInsight clusters is prorated per minute, whether you use them or not. Be sure to delete your cluster after you finish using it. See **how to delete an HDInsight cluster**.

The NYC 2013 Taxi data

The NYC Taxi Trip data is about 20 GB of compressed comma-separated values (CSV) files (~48 GB uncompressed), comprising more than 173 million individual trips and the fares paid for each trip. Each trip record includes the pickup and dropoff location and time, anonymized hack (driver's) license number and medallion (taxi's unique id) number. The data covers all trips in the year 2013 and is provided in the following two datasets for each month:

1. The 'trip_data' CSV files contain trip details, such as number of passengers, pick up and dropoff points, trip duration, and trip length. Here are a few sample records:

```
medallion,hack_license,vendor_id,rate_code,store_and_fwd_flag,pickup_datetime,dropoff_datetime,passenger_count,trip_time_in_secs,trip_distance,pickup_longitude,pickup_latitude,dropoff_longitude,dropoff_latitude
```

```
89D227B655E5C82AECF13C3F540D4CF4,BA96DE419E711691B9445D6A6307C170,CMT,1,N,2013-01-01 15:11:48,2013-01-01 15:18:10,4,382,1.00,-73.978165,40.757977,-73.989838,40.751171
```

```
0BD7C8F5BA12B88E0B67BED28BEA73D8,9FD8F69F0804BDB5549F40E9DA1BE472,CMT,1,N,2013-01-06 00:18:35,2013-01-06 00:22:54,1,259,1.50,-74.006683,40.731781,-73.994499,40.75066
```

```
0BD7C8F5BA12B88E0B67BED28BEA73D8,9FD8F69F0804BDB5549F40E9DA1BE472,CMT,1,N,2013-01-05 18:49:41,2013-01-05 18:54:23,1,282,1.10,-74.004707,40.73777,-74.009834,40.726002
```

```
DFD2202EE08F7A8DC9A57B02ACB81FE2,51EE87E3205C985EF8431D850C786310,CMT,1,N,2013-01-07 23:54:15,2013-01-07 23:58:20,2,244,.70,-73.974602,40.759945,-73.984734,40.759388
```

```
DFD2202EE08F7A8DC9A57B02ACB81FE2,51EE87E3205C985EF8431D850C786310,CMT,1,N,2013-01-07 23:25:03,2013-01-07 23:34:24,1,560,2.10,-73.97625,40.748528,-74.002586,40.747868
```

2. The 'trip_fare' CSV files contain details of the fare paid for each trip, such as payment type, fare amount, surcharge and taxes, tips and tolls, and the total amount paid. Here are a few sample records:

```
medallion, hack_license, vendor_id, pickup_datetime, payment_type, fare_amount, surcharge, mta_tax, tip_amount, tolls_amount, total_amount
```

```
89D227B655E5C82AECF13C3F540D4CF4,BA96DE419E711691B9445D6A6307C170,CMT,2013-01-01 15:11:48,CSH,6.5,0,0.5,0,0,7
```

```
0BD7C8F5BA12B88E0B67BED28BEA73D8,9FD8F69F0804BDB5549F40E9DA1BE472,CMT,2013-01-06 00:18:35,CSH,6,0.5,0.5,0,0,7
```

```
0BD7C8F5BA12B88E0B67BED28BEA73D8,9FD8F69F0804BDB5549F40E9DA1BE472,CMT,2013-01-05 18:49:41,CSH,5.5,1,0.5,0,0,7
```

```
DFD2202EE08F7A8DC9A57B02ACB81FE2,51EE87E3205C985EF8431D850C786310,CMT,2013-01-07 23:54:15,CSH,5,0.5,0.5,0,0,6
```

```
DFD2202EE08F7A8DC9A57B02ACB81FE2,51EE87E3205C985EF8431D850C786310,CMT,2013-01-07 23:25:03,CSH,9.5,0.5,0.5,0,0,10.5
```

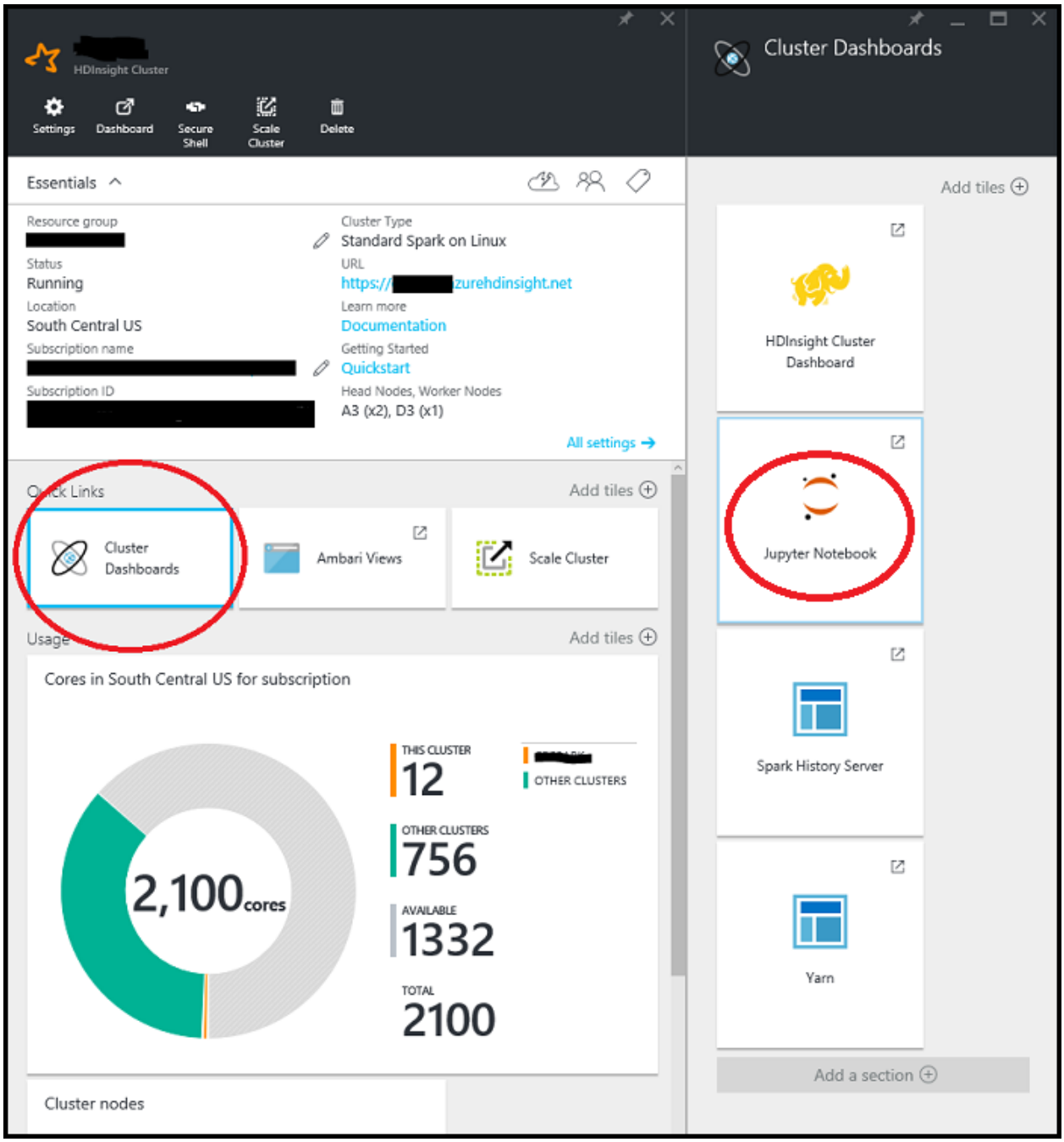
We have taken a 0.1% sample of these files and joined the trip_data and trip_fare CVS files into a single dataset to use as the input dataset for this walkthrough. The unique key to join trip_data and trip_fare is composed of the fields: medallion, hack_licence and pickup_datetime. Each record of the dataset contains the following attributes representing a NYC Taxi trip:

Field	**Brief Description**
medallion	Anonymized taxi medallion (unique taxi id)
hack_license	Anonymized Hackney Carriage License number
vendor_id	Taxi vendor id
rate_code	NYC taxi rate of fare
store_and_fwd_flag	Store and forward flag
pickup_datetime	Pick up date & time
dropoff_datetime	Dropoff date & time
pickup_hour	Pick up hour
pickup_week	Pick up week of the year
weekday	Weekday (range 1-7)
passenger_count	Number of passengers in a taxi trip
trip_time_in_secs	Trip time in seconds
trip_distance	Trip distance traveled in miles
pickup_longitude	Pick up longitude
pickup_latitude	Pick up latitude
dropoff_longitude	Dropoff longitude
dropoff_latitude	Dropoff latitude
direct_distance	Direct distance between pickup and dropoff locations
payment_type	Payment type (cash, credit-card etc.)
fare_amount	Fare amount in

Field	Brief Description
surcharge	Surcharge
mta_tax	MTA Metro Transportation tax
tip_amount	Tip amount
tolls_amount	Tolls amount
total_amount	Total amount
tipped	Tipped (0/1 for no or yes)
tip_class	Tip class (0: $0, 1: $0-5, 2: $6-10, 3: $11-20, 4: > $20)

Execute code from a Jupyter notebook on the Spark cluster

You can launch the Jupyter Notebook from the Azure portal. Find your Spark cluster on your dashboard and click it to enter management page for your cluster. To open the notebook associated with the Spark cluster, click **Cluster Dashboards** -> **Jupyter Notebook**.



You can also browse to `https://CLUSTERNAME.azurehdinsight.net/jupyter` to access the Jupyter Notebooks. Replace the CLUSTERNAME part of this URL with the name of your own cluster. You need the password for your admin account to access the notebooks.

Select PySpark to see a directory that contains a few examples of pre-packaged notebooks that use the PySpark API. The notebooks that contain the code samples for this suite of Spark topic are available at GitHub

You can upload the notebooks directly from GitHub to the Jupyter notebook server on your Spark cluster. On the home page of your Jupyter, click the **Upload** button on the right part of the screen. It opens a file explorer. Here you can paste the GitHub (raw content) URL of the Notebook and click **Open**.

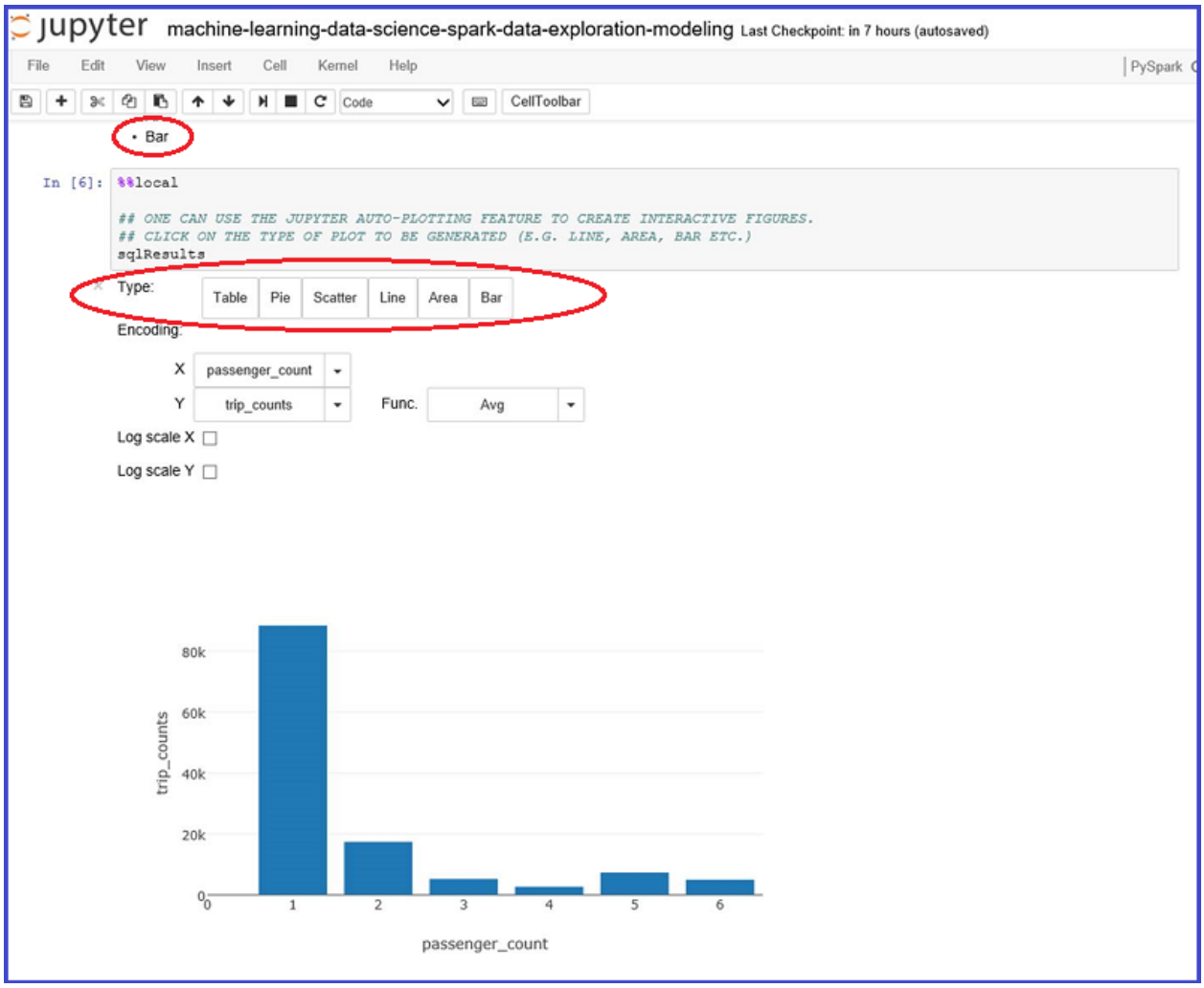
You see the file name on your Jupyter file list with an **Upload** button again. Click this **Upload** button. Now you have imported the notebook. Repeat these steps to upload the other notebooks from this walkthrough.

> **Tip**
>
> You can right-click the links on your browser and select **Copy Link** to get the GitHub raw content URL. You can paste this URL into the Jupyter Upload file explorer dialog box.

Now you can:

- See the code by clicking the notebook.
- Execute each cell by pressing **SHIFT-ENTER**.
- Run the entire notebook by clicking on **Cell** -> **Run**.
- Use the automatic visualization of queries.

> **Tip**
>
> The PySpark kernel automatically visualizes the output of SQL (HiveQL) queries. You are given the option to select among several different types of visualizations (Table, Pie, Line, Area, or Bar) by using the **Type** menu buttons in the notebook:



Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- Mark Tabladillo | Senior Cloud Solution Architect

To see non-public LinkedIn profiles, sign in to LinkedIn.

Next steps

- What is the Team Data Science Process?
- Compare the machine learning products and technologies from Microsoft
- Machine learning at scale

Tutorial: Implement IoT spatial analytics by using Azure Maps

Article • 09/22/2023

In an IoT scenario, it's common to capture and track relevant events that occur in space and time. Examples include fleet management, asset tracking, mobility, and smart city applications. This tutorial guides you through a solution that tracks used car rental movement by using the Azure Maps APIs.

In this tutorial you will:

- ✓ Create an Azure storage account to log car tracking data.
- ✓ Upload a geofence to an Azure storage account.
- ✓ Create a hub in Azure IoT Hub, and register a device.
- ✓ Create a function in Azure Functions, implementing business logic based on Azure Maps spatial analytics.
- ✓ Subscribe to IoT device telemetry events from the Azure function via Azure Event Grid.
- ✓ Filter the telemetry events by using IoT Hub message routing.

Prerequisites

If you don't have an Azure subscription, create a free account before you begin.

- An Azure Maps account
- A subscription key
- A resource group
- The rentalCarSimulation C# project

> **Tip**
>
> You can download the entire **rentalCarSimulation** C# project from GitHub as a single ZIP file by going to **the root of the sample** and selecting the green **<> Code** button, then **Download ZIP**.

This tutorial uses the Postman application, but you can choose a different API development environment.

> **Important**

In the URL examples, replace `{Your-Azure-Maps-Subscription-key}` with your Azure Maps subscription key.

Use case: rental car tracking

Let's say that a car rental company wants to log location information, distance traveled, and running state for its rental cars. The company also wants to store this information whenever a car leaves the correct authorized geographic region.

The rental cars are equipped with IoT devices that regularly send telemetry data to IoT Hub. The telemetry includes the current location and indicates whether the car's engine is running. The device location schema adheres to the IoT Plug and Play schema for geospatial data. The rental car's device telemetry schema looks like the following JSON code:

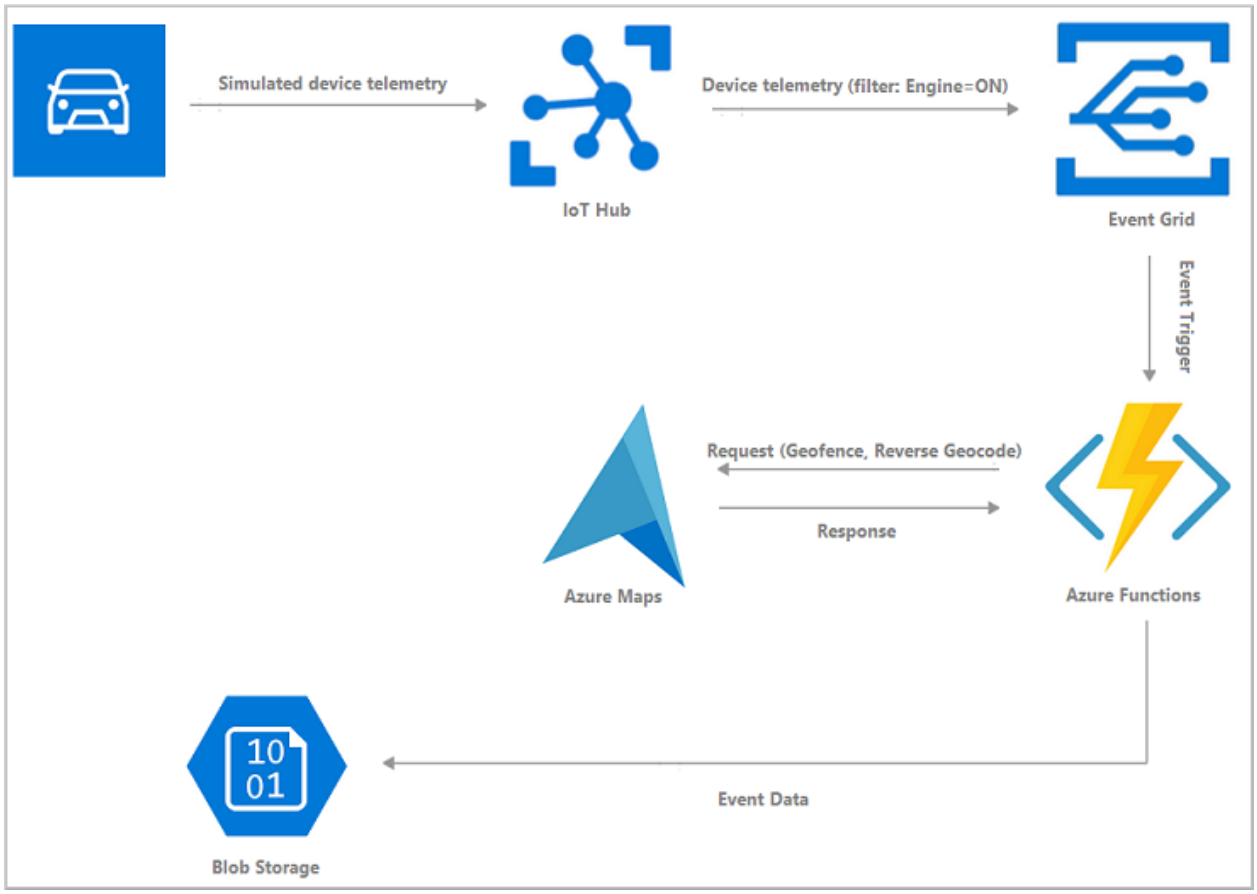
JSON

```
{
    "data": {
        "properties": {
            "Engine": "ON"
        },
        "systemProperties": {
            "iothub-content-type": "application/json",
            "iothub-content-encoding": "utf-8",
            "iothub-connection-device-id": "ContosoRentalDevice",
            "iothub-connection-auth-method": "
{\"scope\":\"device\",\"type\":\"sas\",\"issuer\":\"iothub\",\"acceptingIpFi
lterRule\":null}",
            "iothub-connection-auth-generation-id": "636959817064335548",
            "iothub-enqueuedtime": "2019-06-18T00:17:20.608Z",
            "iothub-message-source": "Telemetry"
        },
        "body": {
            "location": {
                "type": "Point",
                "coordinates": [ -77.025988698005662, 38.9015330523316 ]
            }
        }
    }
}
```

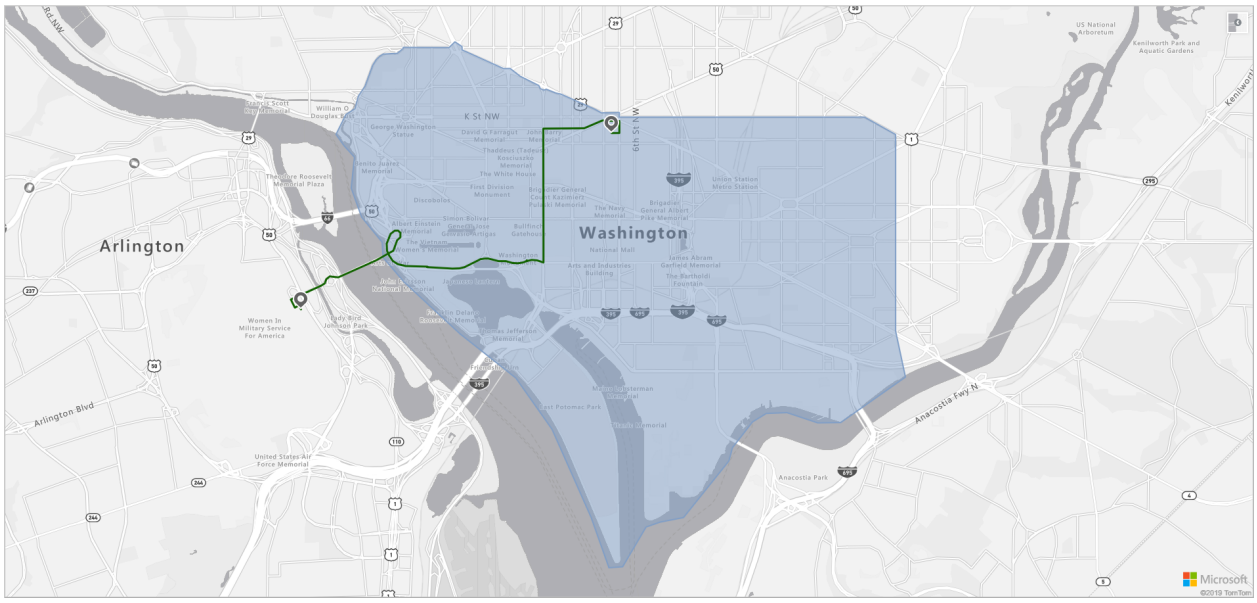
In this tutorial, you only track one vehicle. After you set up the Azure services, you need to download the rentalCarSimulation C# project to run the vehicle simulator. The entire process, from event to function execution, is summarized in the following steps:

1. The in-vehicle device sends telemetry data to IoT Hub.
2. If the car engine is running, the hub publishes the telemetry data to Event Grid.
3. An Azure function is triggered because of its event subscription to device telemetry events.
4. The function logs the vehicle device location coordinates, event time, and the device ID. It then uses the Spatial Geofence Get API to determine whether the car has driven outside the geofence. If it has traveled outside the geofence boundaries, the function stores the location data received from the event into a blob container. The function also queries the Search Address Reverse to translate the coordinate location to a street address, and stores it with the rest of the device location data.

The following diagram shows a high-level overview of the system.



The following figure highlights the geofence area in blue. The rental car's route is indicated by a green line.



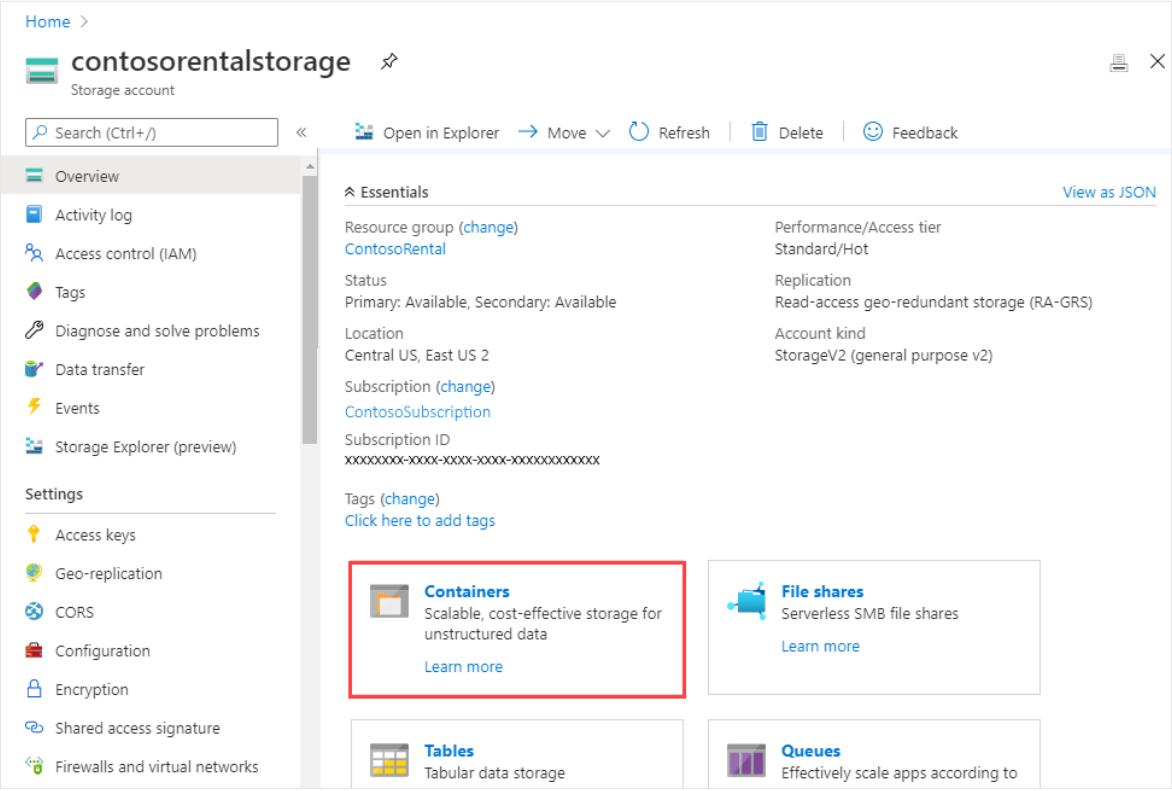
Create an Azure storage account

To store car violation tracking data, create a general-purpose v2 storage account in your resource group. If you haven't created a resource group, follow the directions in create resource groups. Name your resource group *ContosoRental*.

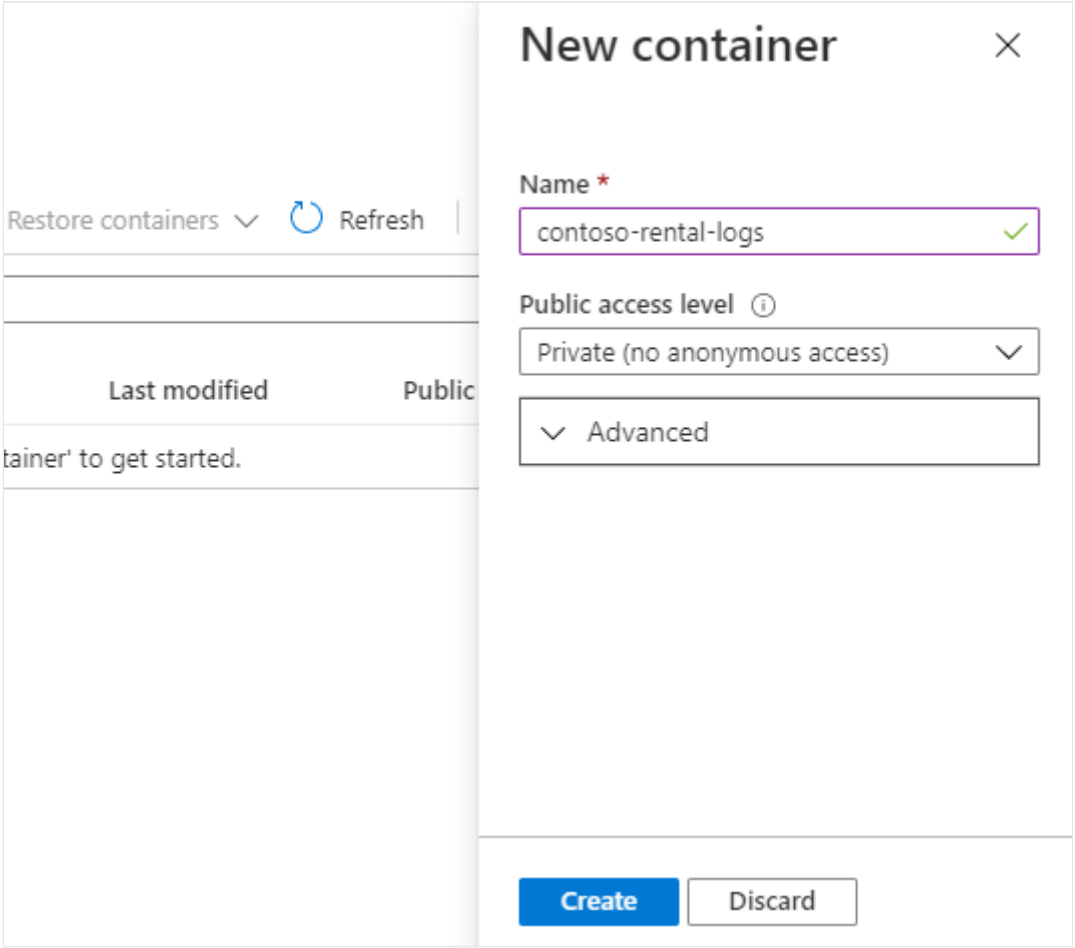
To create a storage account, follow the instructions in create a storage account. In this tutorial, name the storage account *contosorentalstorage*, but in general you can name it anything you like.

When you successfully create your storage account, you then need to create a container to store logging data.

1. Go to your newly created storage account. In the **Essentials** section, select the **Containers** link.

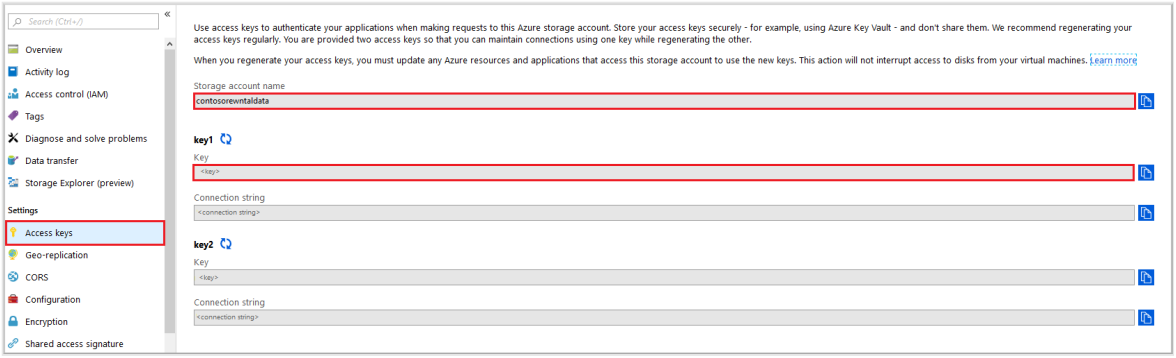


2. In the upper-left corner, select **+ Container**. A panel appears on the right side of the browser. Name your container *contoso-rental-logs*, and select **Create**.



3. Go to the **Access keys** pane in your storage account, and copy the **Storage account name** and the **Key** value in the **key1** section. You need both of these

values in the Create a function and add an Event Grid subscription section.



Upload a geofence into your Azure storage account

The geofence defines the authorized geographical area for our rental vehicle. Use the geofence in your Azure function to determine whether a car has moved outside the geofence area.

Follow the steps outlined in the How to create data registry article to upload the geofence JSON data file into your Azure storage account then register it in your Azure Maps account. Make sure to make a note of the unique identifier (`udid`) value, you'll need it. The `udid` is how you reference the geofence you uploaded into your Azure storage account from your source code. For more information on geofence data files, see Geofencing GeoJSON data.

Create an IoT hub

IoT Hub enables secure and reliable bi-directional communication between an IoT application and the devices it manages. For this tutorial, you want to get information from your in-vehicle device to determine the location of the rental car. In this section, you create an IoT hub within the *ContosoRental* resource group. This hub is responsible for publishing your device telemetry events.

To create an IoT hub in the *ContosoRental* resource group, follow the steps in create an IoT hub.

Register a device in your IoT hub

Devices can't connect to the IoT hub unless they're registered in the IoT hub identity registry. Create a single device with the name, *InVehicleDevice*. To create and register the

device within your IoT hub, follow the steps in register a new device in the IoT hub. Make sure to copy the primary connection string of your device. You'll need it later.

Create a function and add an Event Grid subscription

Azure Functions is a serverless compute service that allows you to run small pieces of code ("functions"), without the need to explicitly provision or manage compute infrastructure. To learn more, see Azure Functions.

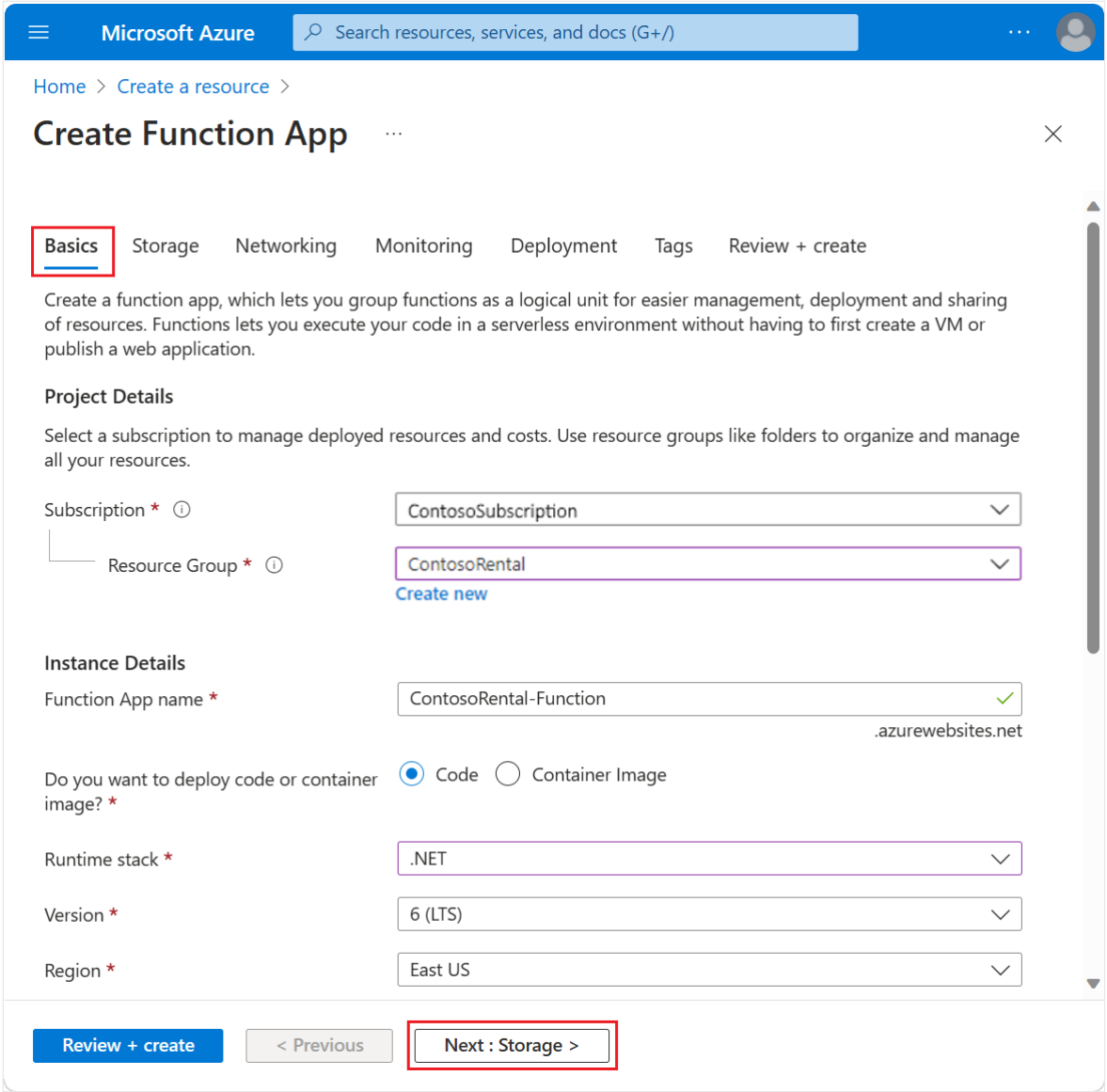
A function is triggered by a certain event. Create a function triggered by an Event Grid trigger. Create the relationship between trigger and function by creating an event subscription for IoT Hub device telemetry events. When a device telemetry event occurs, your function is called as an endpoint, and receives the relevant data for the device you previously registered in IoT Hub.

Here's the C# script code that your function contains.

Now, set up your Azure function.

1. In the Azure portal dashboard, select **Create a resource**. Type **Function App** in the search text box. Select **Function App** > **Create**.

2. On the **Function App** creation page, name your function app. Under **Resource Group**, select **ContosoRental** from the drop-down list. Select **.NET** as the **Runtime Stack**. At the bottom of the page, select **Next: Storage** >.

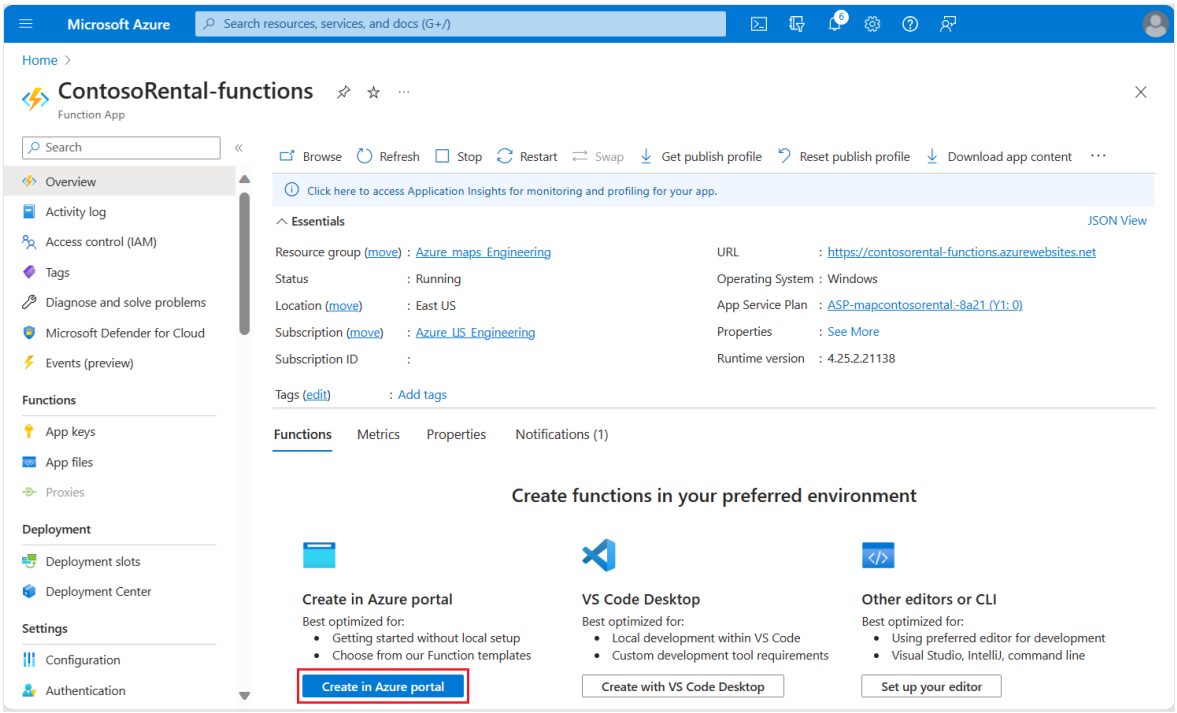


3. For **Storage account**, select the storage account you created in Create an Azure storage account. Select **Review + create**.

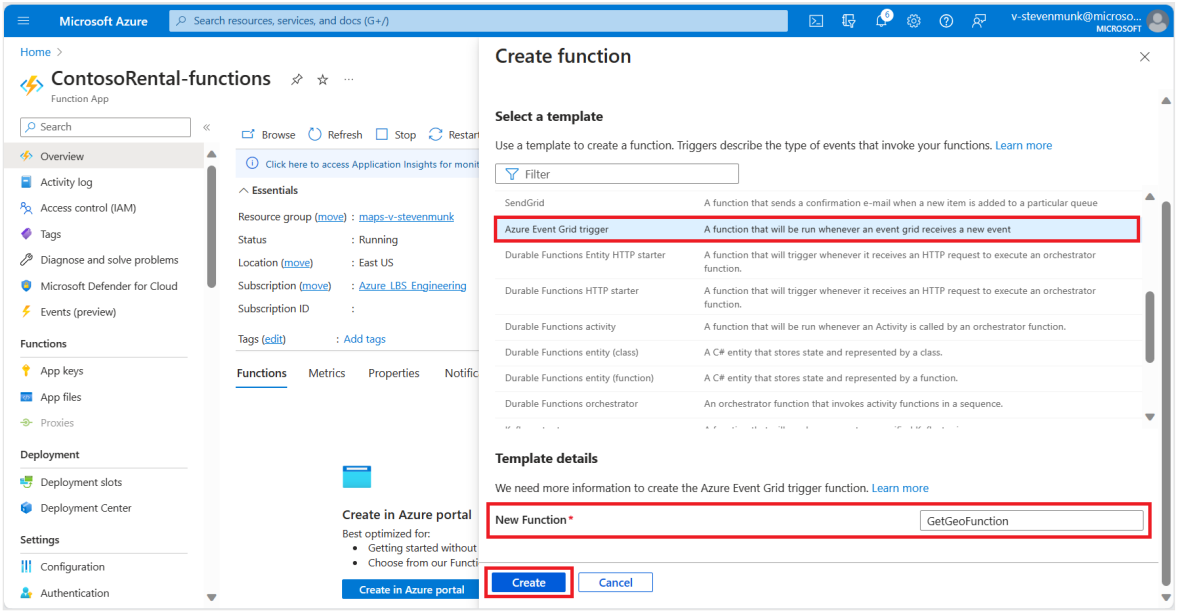
4. Review the function app details, and select **Create**.

5. After the app is created, you add a function to it. Go to the function app. Select the **Create in Azure Portal** button.

> **Important**
>
> The **Azure Event *Hub* Trigger** and the **Azure Event *Grid* Trigger** templates have similar names. Make sure you select the **Azure Event *Grid* Trigger** template.

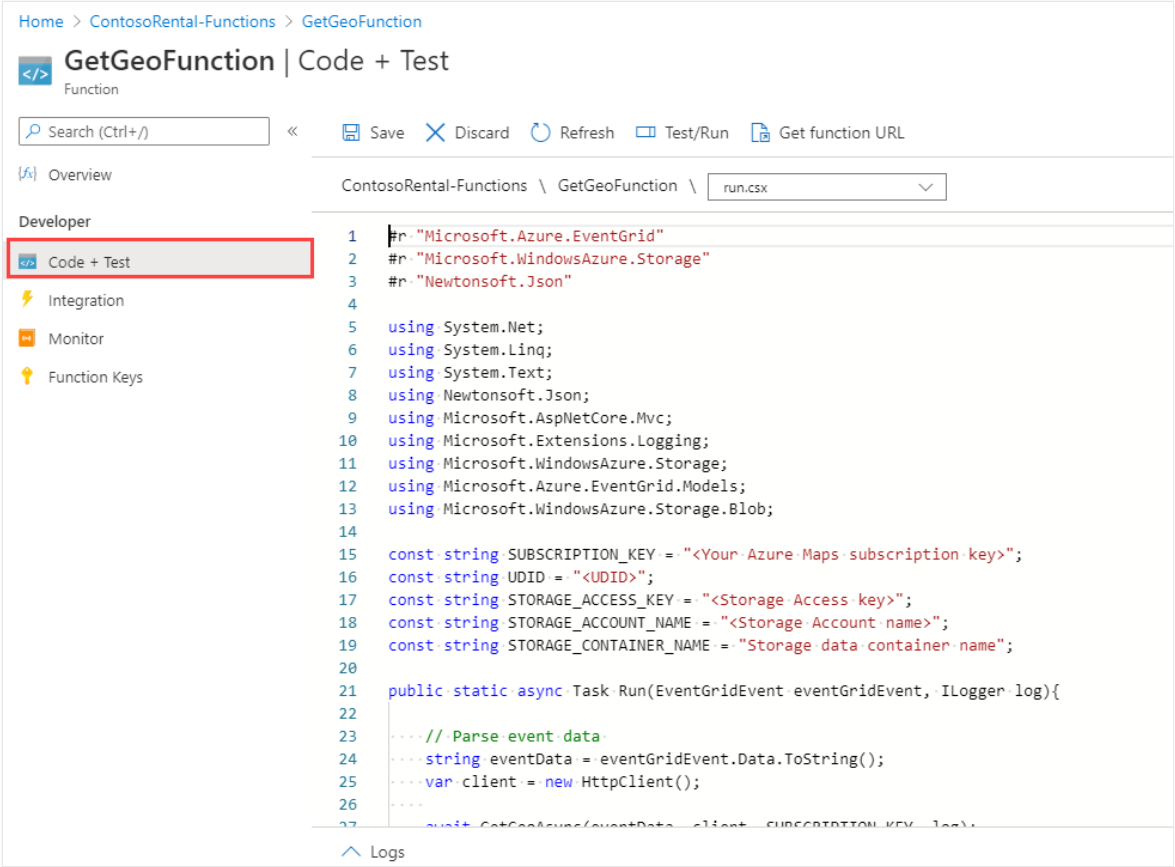


6. The **Create function** panel appears. Scroll down the **Select a template** panel, and select **Azure Event Grid trigger** then select the **Create** button.



7. Give the function a name. In this tutorial, use the name *GetGeoFunction*, but in general you can use any name you like. Select **Create function**.

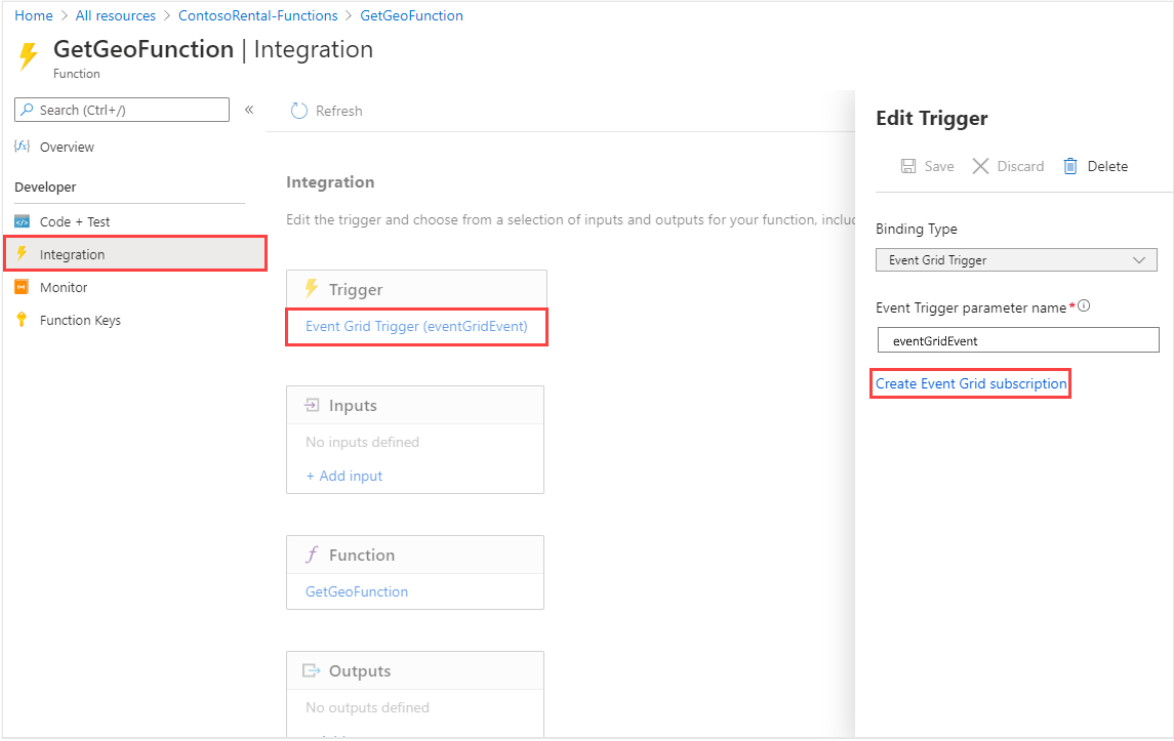
8. In the left menu, select the **Code + Test** pane. Copy and paste the C# script into the code window.



9. In the C# code, replace the following parameters:

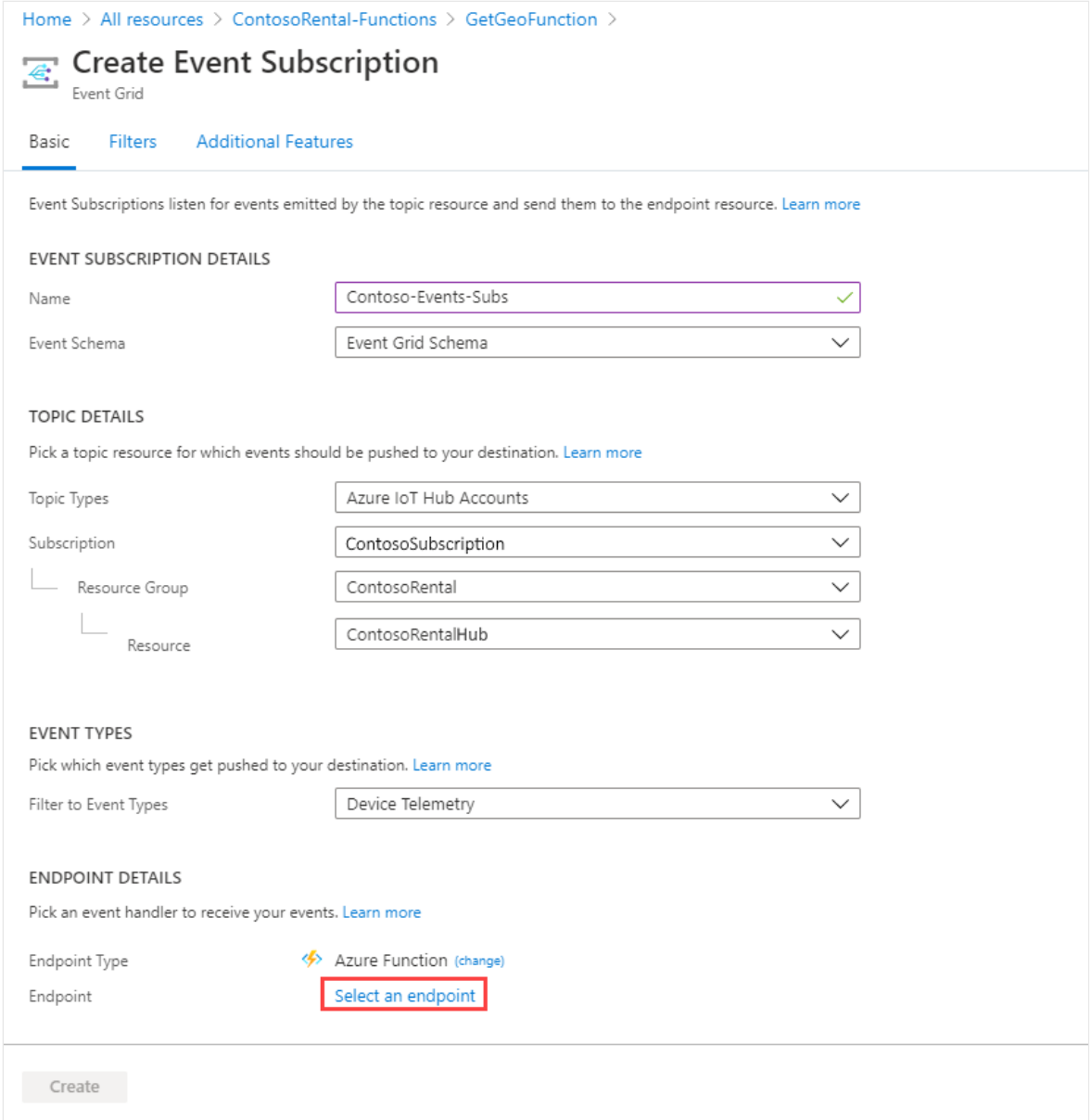
    - Replace **SUBSCRIPTION_KEY** with your Azure Maps account subscription key.
    - Replace **UDID** with the `udid` of the geofence you uploaded in Upload a geofence into your Azure storage account.
    - The `CreateBlobAsync` function in the script creates a blob per event in the data storage account. Replace the **ACCESS_KEY**, **ACCOUNT_NAME**, and **STORAGE_CONTAINER_NAME** with your storage account's access key, account name, and data storage container. These values were generated when you created your storage account in Create an Azure storage account.

10. In the left menu, select the **Integration** pane. Select **Event Grid Trigger** in the diagram. Type in a name for the trigger, *eventGridEvent*, and select **Create Event Grid subscription**.

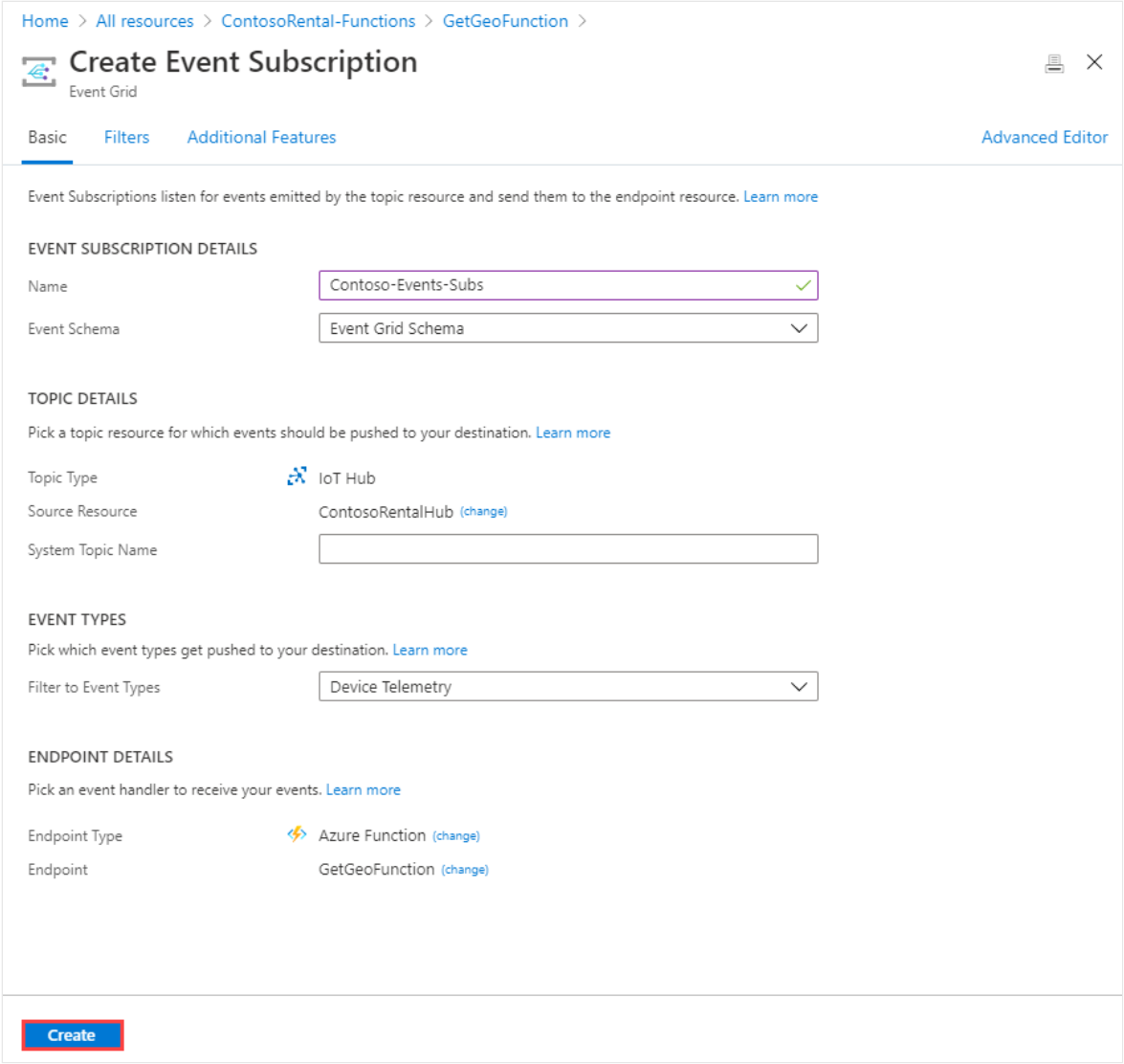


11. Fill out the subscription details. Name the event subscription. For **Event Schema**, select **Event Grid Schema**. For **Topic Types**, select **Azure IoT Hub Accounts**. For **Resource Group**, select the resource group you created at the beginning of this tutorial. For **Resource**, select the IoT hub you created in "Create an Azure IoT hub." For **Filter to Event Types**, select **Device Telemetry**.

    After choosing these options, you'll see the **Topic Type** change to **IoT Hub**. For **System Topic Name**, you can use the same name as your resource. Finally, in the **Endpoint details** section, select **Select an endpoint**. Accept all settings and select **Confirm Selection**.



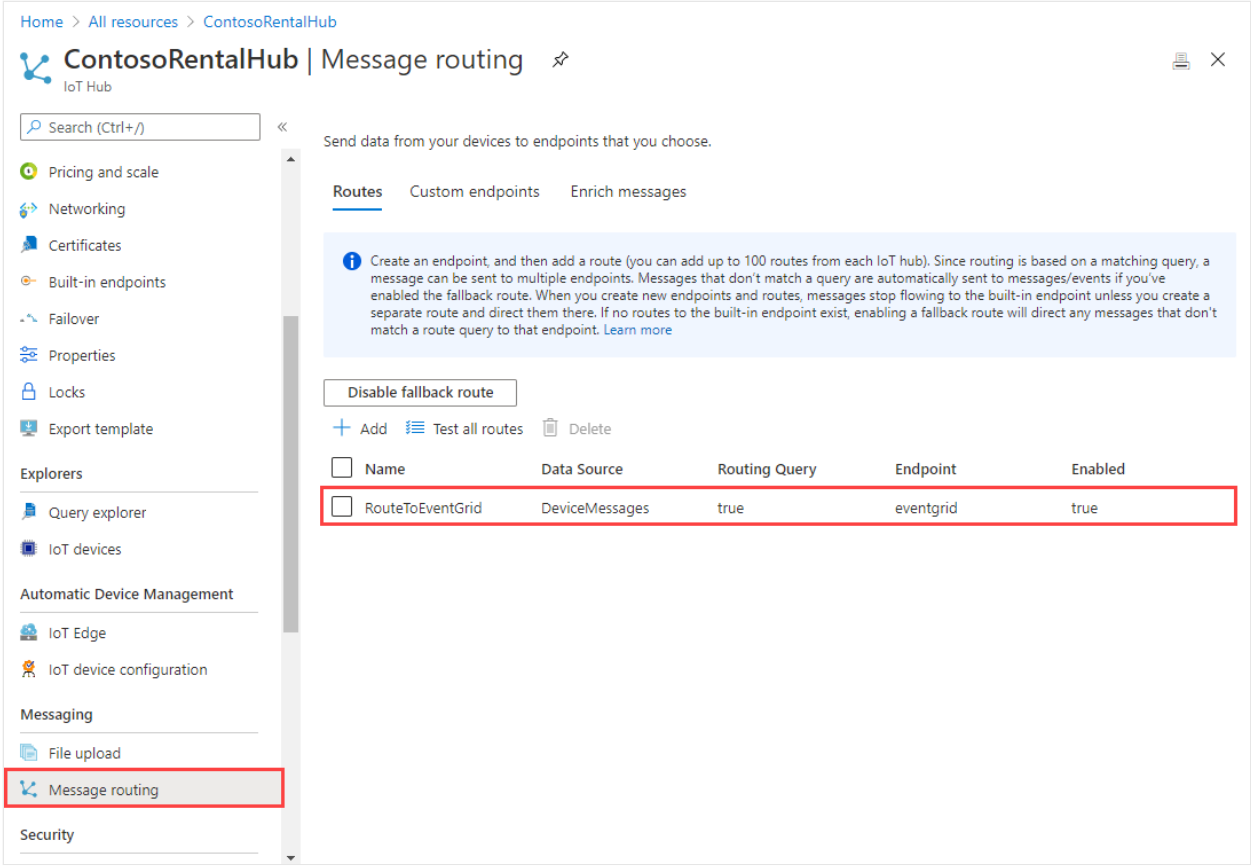
12. Review your settings. Make sure that the endpoint specifies the function you created in the beginning of this section. Select **Create**.



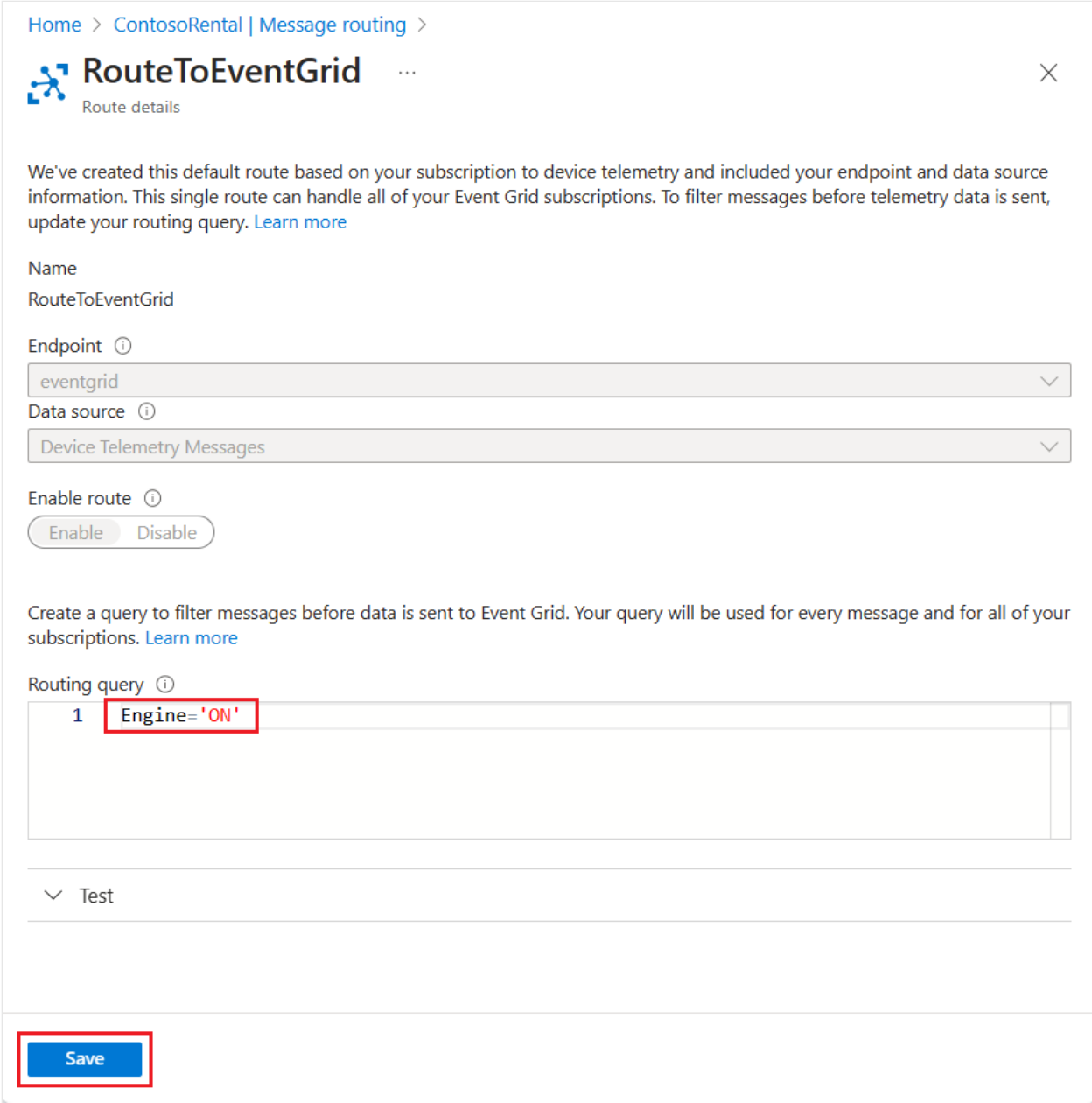
13. Now you're back at the **Edit Trigger** panel. Select **Save**.

Filter events by using IoT Hub message routing

When you add an Event Grid subscription to the Azure function, a messaging route is automatically created in the specified IoT hub. Message routing allows you to route different data types to various endpoints. For example, you can route device telemetry messages, device life-cycle events, and device twin change events. For more information, see Use IoT Hub message routing.



In your example scenario, you only want to receive messages when the rental car is moving. Create a routing query to filter the events where the `Engine` property equals **"ON"**. To create a routing query, select the **RouteToEventGrid** route and replace the **Routing query** with **"Engine='ON'"**. Then select **Save**. Now the IoT hub only publishes device telemetry where the engine is on.



> **Tip**
>
> There are various ways to query IoT device-to-cloud messages. To learn more about message routing syntax, see **IoT Hub message routing**.

Send telemetry data to IoT Hub

When your Azure function is running, you can now send telemetry data to the IoT hub, which routes it to Event Grid. Use a C# application to simulate location data for an in-vehicle device of a rental car. To run the application, you need .NET SDK 6.0 on your development computer. Follow these steps to send simulated telemetry data to the IoT hub:

1. If you haven't done so already, download the rentalCarSimulation C# project.

2. Open the `simulatedCar.cs` file in a text editor of your choice, and replace the value of the `connectionString` with the one you saved when you registered the device. Save changes to the file.

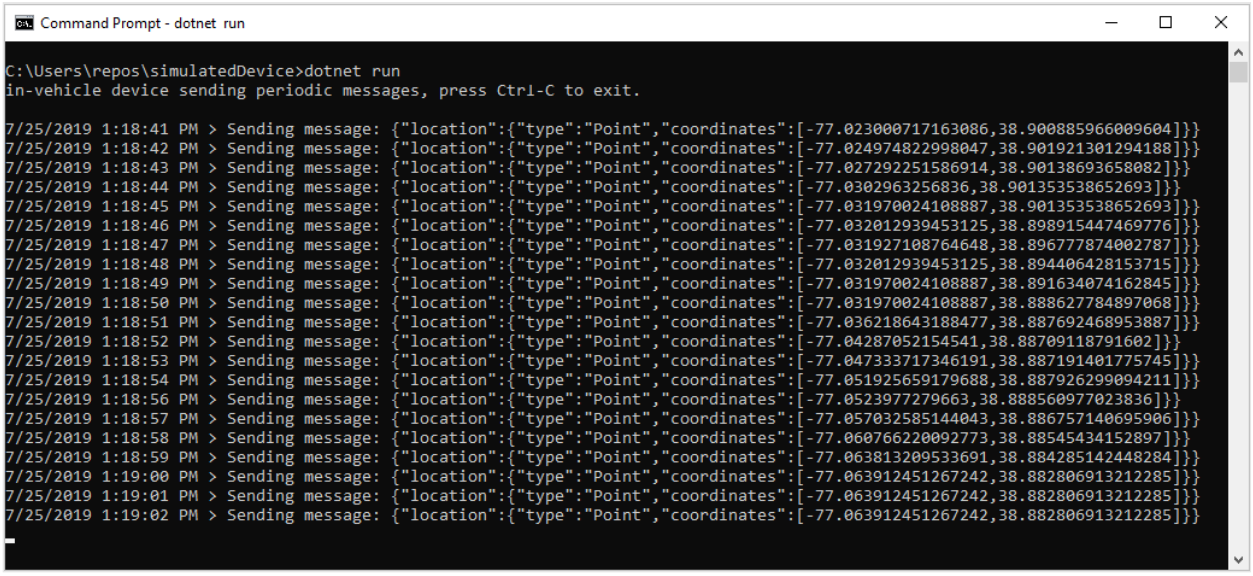
3. Make sure you have the ASP.NET Core Runtime installed on your machine. In your local terminal window, go to the root folder of the C# project and run the following command to install the required packages for simulated device application:

   cmd/sh
   ```
   dotnet restore
   ```

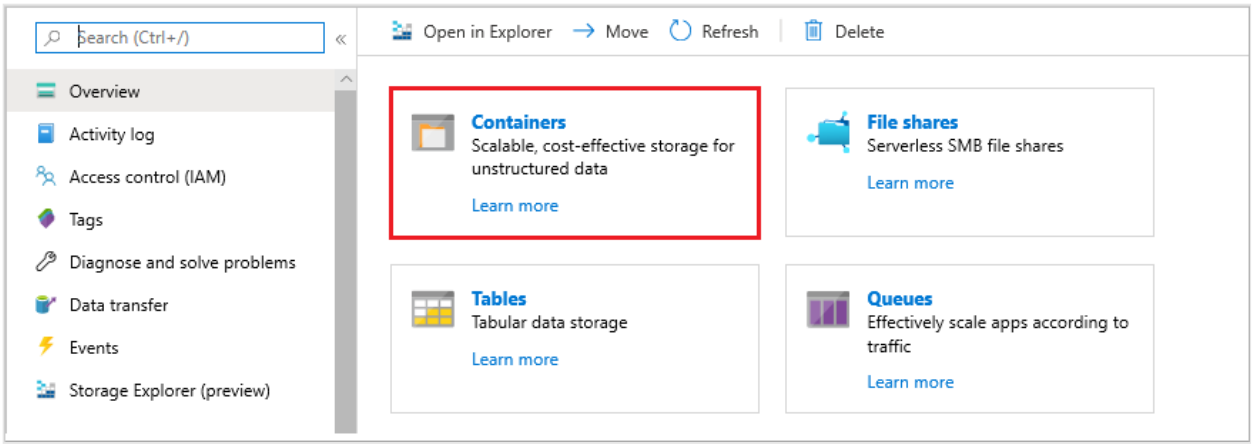
4. In the same terminal, run the following command to build and run the rental car simulation application:

   cmd/sh
   ```
   dotnet run
   ```

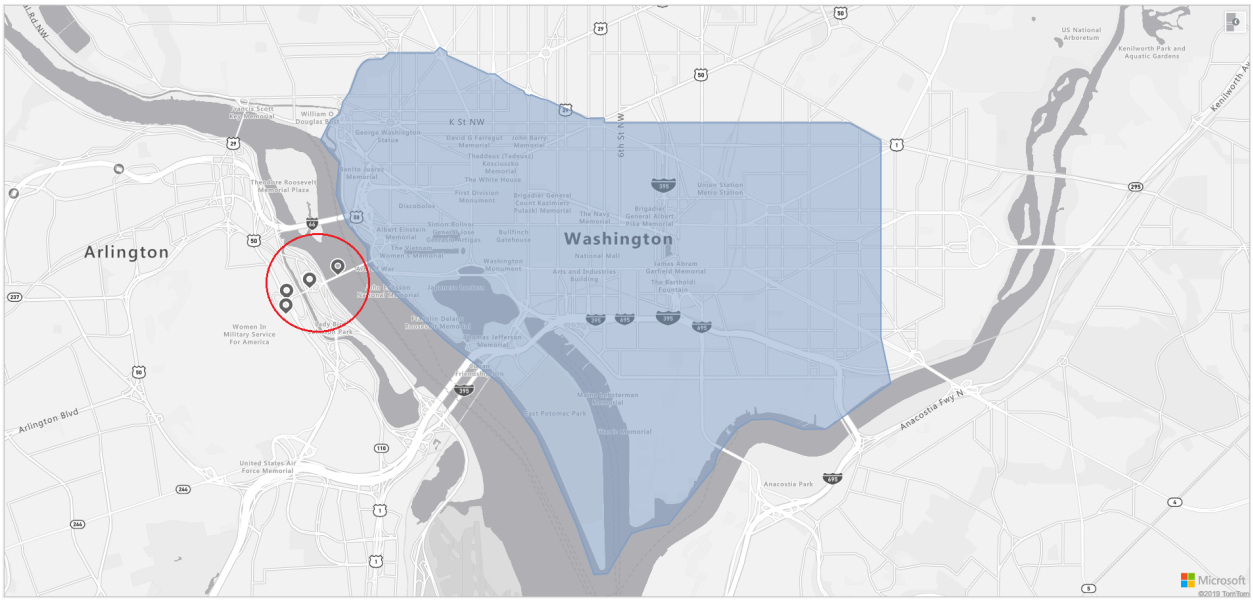
Your local terminal should look like the following screenshot.



If you open the blob storage container now, you can see four blobs for locations where the vehicle was outside the geofence.



The following map shows four vehicle location points outside the geofence. Each location was logged at regular time intervals.



Explore Azure Maps and IoT

To explore the Azure Maps APIs used in this tutorial, see:

- Get Search Address Reverse
- Get Geofence

For a complete list of Azure Maps REST APIs, see:

- Azure Maps REST APIs

- IoT Plug and Play

To get a list of devices that are Azure certified for IoT, visit:

- Azure certified devices

Clean up resources

There are no resources that require cleanup.

Next steps

To learn more about how to send device-to-cloud telemetry, and the other way around, see:

Send telemetry from a device

Observer design pattern

Article • 05/25/2023

The observer design pattern enables a subscriber to register with and receive notifications from a provider. It's suitable for any scenario that requires push-based notification. The pattern defines a *provider* (also known as a *subject* or an *observable*) and zero, one, or more *observers*. Observers register with the provider, and whenever a predefined condition, event, or state change occurs, the provider automatically notifies all observers by invoking a delegate. In this method call, the provider can also provide current state information to observers. In .NET, the observer design pattern is applied by implementing the generic System.IObservable<T> and System.IObserver<T> interfaces. The generic type parameter represents the type that provides notification information.

When to apply the pattern

The observer design pattern is suitable for distributed push-based notifications, because it supports a clean separation between two different components or application layers, such as a data source (business logic) layer and a user interface (display) layer. The pattern can be implemented whenever a provider uses callbacks to supply its clients with current information.

Implementing the pattern requires that you provide the following details:

- A provider or subject, which is the object that sends notifications to observers. A provider is a class or structure that implements the IObservable<T> interface. The provider must implement a single method, IObservable<T>.Subscribe, which is called by observers that wish to receive notifications from the provider.

- An observer, which is an object that receives notifications from a provider. An observer is a class or structure that implements the IObserver<T> interface. The observer must implement three methods, all of which are called by the provider:
  - IObserver<T>.OnNext, which supplies the observer with new or current information.
  - IObserver<T>.OnError, which informs the observer that an error has occurred.
  - IObserver<T>.OnCompleted, which indicates that the provider has finished sending notifications.

- A mechanism that allows the provider to keep track of observers. Typically, the provider uses a container object, such as a System.Collections.Generic.List<T> object, to hold references to the IObserver<T> implementations that have subscribed to notifications. Using a storage container for this purpose enables the

provider to handle zero to an unlimited number of observers. The order in which observers receive notifications isn't defined; the provider is free to use any method to determine the order.

- An IDisposable implementation that enables the provider to remove observers when notification is complete. Observers receive a reference to the IDisposable implementation from the Subscribe method, so they can also call the IDisposable.Dispose method to unsubscribe before the provider has finished sending notifications.

- An object that contains the data that the provider sends to its observers. The type of this object corresponds to the generic type parameter of the IObservable<T> and IObserver<T> interfaces. Although this object can be the same as the IObservable<T> implementation, most commonly it's a separate type.

> **Note**
>
> In addition to implementing the observer design pattern, you may be interested in exploring libraries that are built using the **IObservable<T>** and **IObserver<T>** interfaces. For example, **Reactive Extensions for .NET (Rx)** consist of a set of extension methods and LINQ standard sequence operators to support asynchronous programming.

Implement the pattern

The following example uses the observer design pattern to implement an airport baggage claim information system. A `BaggageInfo` class provides information about arriving flights and the carousels where baggage from each flight is available for pickup. It's shown in the following example.

C#

```
namespace Observables.Example;

public readonly record struct BaggageInfo(
    int FlightNumber,
    string From,
    int Carousel);
```

A `BaggageHandler` class is responsible for receiving information about arriving flights and baggage claim carousels. Internally, it maintains two collections:

- `_observers`: A collection of clients that observe updated information.
- `_flights`: A collection of flights and their assigned carousels.

The source code for the `BaggageHandler` class is shown in the following example.

C#

```
namespace Observables.Example;

public sealed class BaggageHandler : IObservable<BaggageInfo>
{
    private readonly HashSet<IObserver<BaggageInfo>> _observers = new();
    private readonly HashSet<BaggageInfo> _flights = new();

    public IDisposable Subscribe(IObserver<BaggageInfo> observer)
    {
        // Check whether observer is already registered. If not, add it.
        if (_observers.Add(observer))
        {
            // Provide observer with existing data.
            foreach (BaggageInfo item in _flights)
            {
                observer.OnNext(item);
            }
        }

        return new Unsubscriber<BaggageInfo>(_observers, observer);
    }

    // Called to indicate all baggage is now unloaded.
    public void BaggageStatus(int flightNumber) =>
        BaggageStatus(flightNumber, string.Empty, 0);

    public void BaggageStatus(int flightNumber, string from, int carousel)
    {
        var info = new BaggageInfo(flightNumber, from, carousel);

        // Carousel is assigned, so add new info object to list.
        if (carousel > 0 && _flights.Add(info))
        {
            foreach (IObserver<BaggageInfo> observer in _observers)
            {
                observer.OnNext(info);
            }
        }
        else if (carousel is 0)
        {
            // Baggage claim for flight is done.
            if (_flights.RemoveWhere(
                flight => flight.FlightNumber == info.FlightNumber) > 0)
            {
                foreach (IObserver<BaggageInfo> observer in _observers)
                {
```

```
                    observer.OnNext(info);
                }
            }
        }
    }

    public void LastBaggageClaimed()
    {
        foreach (IObserver<BaggageInfo> observer in _observers)
        {
            observer.OnCompleted();
        }

        _observers.Clear();
    }
}
```

Clients that wish to receive updated information call the `BaggageHandler.Subscribe` method. If the client hasn't previously subscribed to notifications, a reference to the client's IObserver<T> implementation is added to the `_observers` collection.

The overloaded `BaggageHandler.BaggageStatus` method can be called to indicate that baggage from a flight either is being unloaded or is no longer being unloaded. In the first case, the method is passed a flight number, the airport from which the flight originated, and the carousel where baggage is being unloaded. In the second case, the method is passed only a flight number. For baggage that is being unloaded, the method checks whether the `BaggageInfo` information passed to the method exists in the `_flights` collection. If it doesn't, the method adds the information and calls each observer's `OnNext` method. For flights whose baggage is no longer being unloaded, the method checks whether information on that flight is stored in the `_flights` collection. If it is, the method calls each observer's `OnNext` method and removes the `BaggageInfo` object from the `_flights` collection.

When the last flight of the day has landed and its baggage has been processed, the `BaggageHandler.LastBaggageClaimed` method is called. This method calls each observer's `OnCompleted` method to indicate that all notifications have completed, and then clears the `_observers` collection.

The provider's Subscribe method returns an IDisposable implementation that enables observers to stop receiving notifications before the OnCompleted method is called. The source code for this `Unsubscriber(Of BaggageInfo)` class is shown in the following example. When the class is instantiated in the `BaggageHandler.Subscribe` method, it's passed a reference to the `_observers` collection and a reference to the observer that is added to the collection. These references are assigned to local variables. When the

object's `Dispose` method is called, it checks whether the observer still exists in the `_observers` collection, and, if it does, removes the observer.

C#

```
namespace Observables.Example;

internal sealed class Unsubscriber<BaggageInfo> : IDisposable
{
    private readonly ISet<IObserver<BaggageInfo>> _observers;
    private readonly IObserver<BaggageInfo> _observer;

    internal Unsubscriber(
        ISet<IObserver<BaggageInfo>> observers,
        IObserver<BaggageInfo> observer) => (_observers, _observer) =
(observers, observer);

    public void Dispose() => _observers.Remove(_observer);
}
```

The following example provides an IObserver<T> implementation named `ArrivalsMonitor`, which is a base class that displays baggage claim information. The information is displayed alphabetically, by the name of the originating city. The methods of `ArrivalsMonitor` are marked as `overridable` (in Visual Basic) or `virtual` (in C#), so they can be overridden in a derived class.

C#

```
namespace Observables.Example;

public class ArrivalsMonitor : IObserver<BaggageInfo>
{
    private readonly string _name;
    private readonly List<string> _flights = new();
    private readonly string _format = "{0,-20} {1,5}  {2, 3}";
    private IDisposable? _cancellation;

    public ArrivalsMonitor(string name)
    {
        ArgumentException.ThrowIfNullOrEmpty(name);
        _name = name;
    }

    public virtual void Subscribe(BaggageHandler provider) =>
        _cancellation = provider.Subscribe(this);

    public virtual void Unsubscribe()
    {
        _cancellation?.Dispose();
        _flights.Clear();
```

```
    }

    public virtual void OnCompleted() => _flights.Clear();

    // No implementation needed: Method is not called by the BaggageHandler
class.
    public virtual void OnError(Exception e)
    {
        // No implementation.
    }

    // Update information.
    public virtual void OnNext(BaggageInfo info)
    {
        bool updated = false;

        // Flight has unloaded its baggage; remove from the monitor.
        if (info.Carousel is 0)
        {
            string flightNumber = string.Format("{0,5}", info.FlightNumber);
            for (int index = _flights.Count - 1; index >= 0; index--)
            {
                string flightInfo = _flights[index];
                if (flightInfo.Substring(21, 5).Equals(flightNumber))
                {
                    updated = true;
                    _flights.RemoveAt(index);
                }
            }
        }
        else
        {
            // Add flight if it doesn't exist in the collection.
            string flightInfo = string.Format(_format, info.From,
info.FlightNumber, info.Carousel);
            if (_flights.Contains(flightInfo) is false)
            {
                _flights.Add(flightInfo);
                updated = true;
            }
        }

        if (updated)
        {
            _flights.Sort();
            Console.WriteLine($"Arrivals information from {_name}");
            foreach (string flightInfo in _flights)
            {
                Console.WriteLine(flightInfo);
            }

            Console.WriteLine();
        }
    }
}
```

The `ArrivalsMonitor` class includes the `Subscribe` and `Unsubscribe` methods. The `Subscribe` method enables the class to save the IDisposable implementation returned by the call to Subscribe to a private variable. The `Unsubscribe` method enables the class to unsubscribe from notifications by calling the provider's Dispose implementation. `ArrivalsMonitor` also provides implementations of the OnNext, OnError, and OnCompleted methods. Only the OnNext implementation contains a significant amount of code. The method works with a private, sorted, generic List<T> object that maintains information about the airports of origin for arriving flights and the carousels on which their baggage is available. If the `BaggageHandler` class reports a new flight arrival, the OnNext method implementation adds information about that flight to the list. If the `BaggageHandler` class reports that the flight's baggage has been unloaded, the OnNext method removes that flight from the list. Whenever a change is made, the list is sorted and displayed to the console.

The following example contains the application entry point that instantiates the `BaggageHandler` class and two instances of the `ArrivalsMonitor` class, and uses the `BaggageHandler.BaggageStatus` method to add and remove information about arriving flights. In each case, the observers receive updates and correctly display baggage claim information.

C#

```
using Observables.Example;

BaggageHandler provider = new();
ArrivalsMonitor observer1 = new("BaggageClaimMonitor1");
ArrivalsMonitor observer2 = new("SecurityExit");

provider.BaggageStatus(712, "Detroit", 3);
observer1.Subscribe(provider);

provider.BaggageStatus(712, "Kalamazoo", 3);
provider.BaggageStatus(400, "New York-Kennedy", 1);
provider.BaggageStatus(712, "Detroit", 3);
observer2.Subscribe(provider);

provider.BaggageStatus(511, "San Francisco", 2);
provider.BaggageStatus(712);
observer2.Unsubscribe();

provider.BaggageStatus(400);
provider.LastBaggageClaimed();

// Sample output:
//   Arrivals information from BaggageClaimMonitor1
//   Detroit                712    3
```

```
//
//   Arrivals information from BaggageClaimMonitor1
//   Detroit                712    3
//   Kalamazoo              712    3
//
//   Arrivals information from BaggageClaimMonitor1
//   Detroit                712    3
//   Kalamazoo              712    3
//   New York-Kennedy       400    1
//
//   Arrivals information from SecurityExit
//   Detroit                712    3
//
//   Arrivals information from SecurityExit
//   Detroit                712    3
//   Kalamazoo              712    3
//
//   Arrivals information from SecurityExit
//   Detroit                712    3
//   Kalamazoo              712    3
//   New York-Kennedy       400    1
//
//   Arrivals information from BaggageClaimMonitor1
//   Detroit                712    3
//   Kalamazoo              712    3
//   New York-Kennedy       400    1
//   San Francisco          511    2
//
//   Arrivals information from SecurityExit
//   Detroit                712    3
//   Kalamazoo              712    3
//   New York-Kennedy       400    1
//   San Francisco          511    2
//
//   Arrivals information from BaggageClaimMonitor1
//   New York-Kennedy       400    1
//   San Francisco          511    2
//
//   Arrivals information from SecurityExit
//   New York-Kennedy       400    1
//   San Francisco          511    2
//
//   Arrivals information from BaggageClaimMonitor1
//   San Francisco          511    2
```

Related articles

Title	Description
Observer Design Pattern Best Practices	Describes best practices to adopt when developing applications that implement the observer design pattern.

Title	Description
How to: Implement a Provider	Provides a step-by-step implementation of a provider for a temperature monitoring application.
How to: Implement an Observer	Provides a step-by-step implementation of an observer for a temperature monitoring application.

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 feedback

The .NET documentation is open source. Provide feedback here.

Open a documentation issue

Provide product feedback

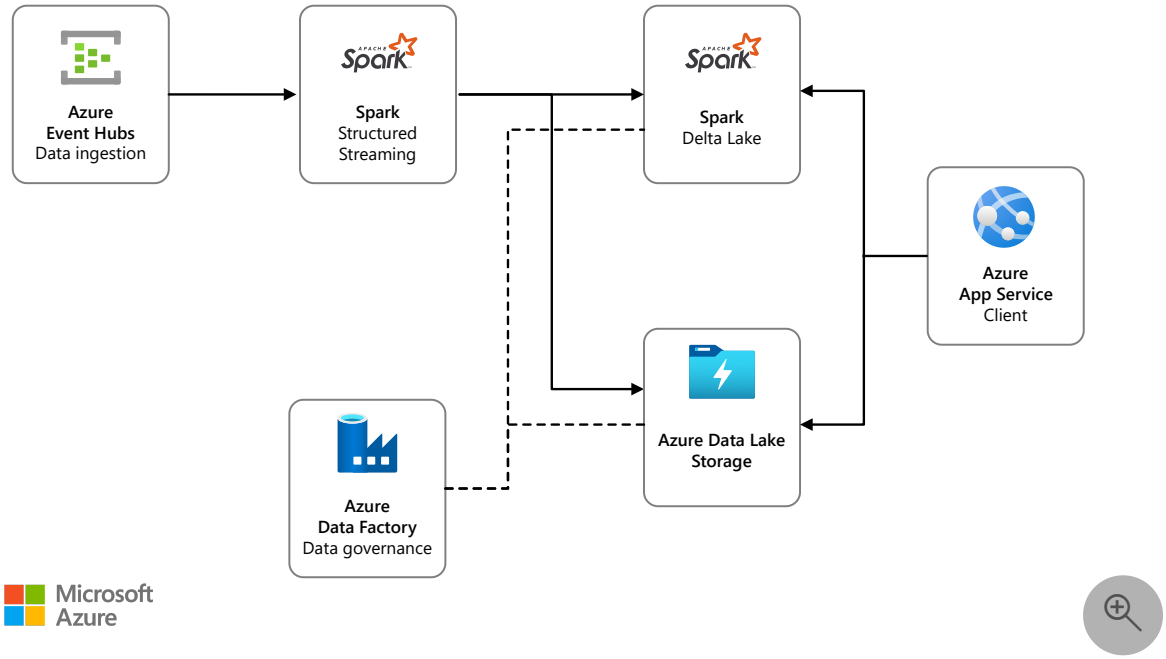
Build a delta lake to support ad hoc queries in online leisure and travel booking

Azure Event Hubs | Azure Data Lake Storage | Azure Databricks | Azure Synapse Analytics

This architecture provides an example delta lake for travel booking, where large amounts of raw documents are generated at a high frequency.

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.

Leisure and travel booking scenarios can generate large amounts of raw documents at a high frequency. However, you may not need to index the entire contents of these documents. For example, users may need to search by a known transaction ID, or by a customer name on a certain date, to retrieve a set of documents that is interesting to them.

Dataflow

The concept behind this architecture consists in decoupling the metadata useful for searching from bare data:

- Only metadata gets indexed within a queryable service (such as Spark), while the actual data is stored in a data lake.
- Raw documents in a data lake are linked to indexed metadata by their path.
- When querying for documents, the service searches the documents' metadata, and in turn the actual documents will be retrieved from the data lake by their path.

This solution dramatically lowers costs and increases performance, as metadata comprises a fraction of the entire data estate (for instance, petabytes of raw documents can be described by tens of gigabytes of concise metadata).

In addition, managing the blending of historical depth and real-time requirements into a uniform, easy-to-maintain, high-performance system is a typical challenge of this type of scenario. The Delta Lake architecture answers this challenge.

Components

Azure App Service is a platform as a service (PaaS) for building and hosting apps in managed virtual machines. App Service manages the underlying compute infrastructure on which your apps run, and it provides monitoring of resource usage quotas and app metrics, logging of diagnostic information, and alerts based on metrics.

Azure Data Factory is Azure's cloud extract, transform, and load (ETL) service for scale-out serverless data integration and data transformation. It offers a code-free UI for intuitive authoring and single-pane-of-glass monitoring and management. You can also lift and shift existing SQL Server Integration Services (SSIS) packages to Azure and run them with full compatibility in Azure Data Factory.

Azure Data Lake Storage Gen2 is a set of capabilities dedicated to big data analytics, built on Azure Blob Storage. Data Lake Storage Gen2 converges the capabilities of Azure Data Lake Storage Gen1 with Azure Blob Storage. For example, Data Lake Storage Gen2 provides file system semantics, file-level security, and scale. Because these capabilities are built on Blob Storage, you also get low-cost, tiered storage, with high availability/disaster recovery capabilities.

Azure Event Hubs is a fully managed, real-time data ingestion service that's simple, trusted, and scalable. Stream millions of events per second from any source to build dynamic data pipelines and immediately respond to business challenges.

Azure Databricks is an Apache Spark–based data analytics platform optimized for Microsoft Azure Cloud Services. Azure Databricks offers three environments for developing data intensive applications: Databricks SQL, Databricks Data Science & Engineering, and Databricks Machine Learning.

Alternatives

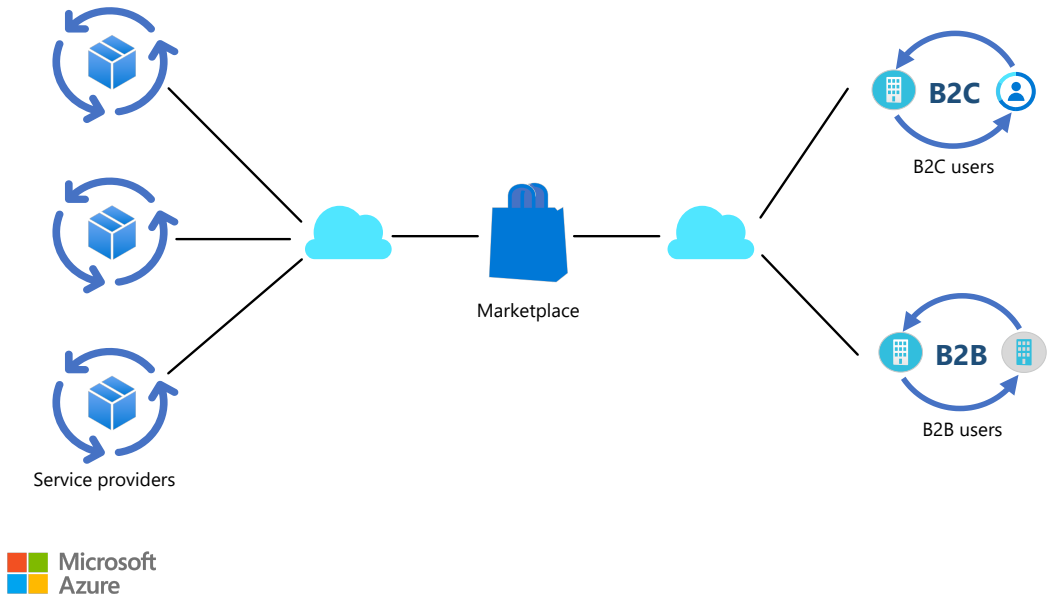
As an alternative to only indexing metadata, you could index all raw data in a service that offers query capabilities, such as Azure Databricks, Azure Synapse Analytics, Azure Cognitive Search, or Azure Data Explorer. This approach is more immediate, but pay attention to the combined effect of data size, performance requirements, and update frequency, especially from a cost perspective.

Contrary to using a delta lake, using a Lambda architecture keeps real-time data in a different repository than historical data, and your client runs the logic to make heterogeneous queries transparent to the user. The advantage of this solution is the larger set of services that you can use (such as Azure Stream Analytics and Azure SQL Database), but the architecture becomes more complex and the code base more expensive to maintain.

Spark is distributed with Azure Databricks, Azure Synapse Analytics, and Azure HDInsight. Hence, this architecture could be implemented with any of these Azure data services, preferably with a recent Spark version supporting Delta Lake 0.8 or 1.0.

Scenario details

The visibility of raw data in leisure and travel booking scenarios is important to multiple actors. Technical support teams oversee real-time diagnostics to continuously monitor transaction processing and rapidly react to undesired issues. Data engineers oversee exporting data for stakeholder review and to feed analytics in real time. Customer support teams require historical and recent data to handle customer inquiries and complaints. Finally, legal teams ensure compliance duties are respected and legal actions performed. These types of requirements are typical in marketplaces that aggregate external providers and manage user purchases. For example, leisure and travel booking systems disintermediate users and services providers for searching services, aggregating meaningful offers from providers, and managing user reservations.



Potential use cases

This architecture is ideal for the travel and hospitality industries. It's applicable to the following scenarios:

- Quickly retrieving either real-time (for example, for diagnostics) or historical (for compliance) raw documents in their original format.
- Managing petabytes of data.
- Guaranteeing seconds-range performance for real-time diagnostics.
- Achieving a unified approach to real-time diagnostics, historical queries, and feeding analytics.
- Feeding downstream real-time analytics.
- Controlling costs.
- Insourcing data as raw documents (for instance, as json, xml, or csv files).
- When a fraction of data is sufficient to describe queries.
- When users want to retrieve full raw documents.
- When the total data size would require scaling the system above your target price.

This architecture might not be suitable when:

- Data is insourced as recordsets.
- Users are required to run analytics.
- Users are willing to use their own packaged BI tool.
- The size of data is not a challenge from a cost perspective.

Raw documents are not necessarily required.

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.

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.

Users will perform a double hop to access data. They'll query metadata first, and then retrieve the desired set of documents. It might be difficult to reuse existing or packaged client assets.

Azure Data Lake Storage Gen2 provides three access tiers: hot, cool, and archive. In scenarios where documents are occasionally retrieved, the cool performance tier should guarantee similar performance to the hot performance tier but with the advantage of lower costs. In scenarios where the probability of retrieval is higher with newer data, consider blending the cool and hot tiers. Using archive tier storage could also provide an alternative to hard deletion, as well as reduce the size of data by keeping only meaningful information or more aggregated data.

The data lake will potentially manage petabytes of data, so data retention policies generally apply. Data governance solutions should be employed to manage data lifecycle, such as when to move old data between hot and cool storage tiers, when to delete or archive old data, and when to aggregate information into a downstream analytics solution.

Consider how this approach might work with downstream analytics scenarios. Although this example workload is not meant for analytics, it is appropriate for feeding downstream real-time analytics, while batch scenarios could be fed from the data lake instead.

Scalability

Azure Event Hubs is highly versatile when it comes to decoupling a transactional system that generates raw documents from a diagnostics and compliance system; is easy to implement in already-established architectures; and, ultimately, is easy to use. However, the transactional system might already use the streaming pattern to process incoming

documents. In that case, you would likely need to integrate logic for managing diagnostics and compliance into the streaming application as a substream.

DevOps

For deploying the used services in this example workload automatically, it's best to use continuous integration and continuous deployment (CI/CD) processes. Consider using a solution such as Azure DevOps or GitHub Actions.

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.

In general, use the Azure pricing calculator to estimate costs. See the cost section in Microsoft Azure Well-Architected Framework to learn about other considerations.

Deploy this scenario

In the following example architecture, we assume that one or more Azure Event Hubs namespaces will contain structured raw documents (such as json or xml files). However, the actual type and format of documents and source services, and their type of integration, is highly dependent on the specific scenario and architecture.

Streaming

With Spark Structured Streaming, raw data is pulled, decompressed, parsed, and translated to tabular data in a streaming DataFrame.

The following PySpark code snippet is used to load a streaming DataFrame from Event Hubs:

```
# Code tested in Databricks with Delta Lake 1.0
eh_connstr = <your_conn_str>
eh_consumergroup = <your_consumer_group>
ehConf = {}
ehConf['eventhubs.connectionString'] =
sc._jvm.org.apache.spark.eventhubs.EventHubsUtils.encrypt(eh_conn
str)
ehConf['eventhubs.consumerGroup'] = eh_consumergroup
```

```
streaming_df = spark \
  .readStream \
  .format("eventhubs") \
  .options(**ehConf) \
  .load()
```

The following code snippet is used to process the streaming DataFrame. It first decompresses the Event Hubs message if necessary, and then parses its json structure to a tabular format. This code is an example and should be adapted to your specific scenario:

```
# Code tested in Databricks with Delta Lake 1.0

# defines an UDF to unzip the Event Hubs Body field, assuming it
is gzipped

import zlib
def DecompressFunction(data):
  decoded_data = zlib.decompress(bytes(data), 15+32)
  return decoded_data.decode()

Decompress = udf(lambda body: DecompressFunction(body),
StringType())
decoded_body_df = streaming_df.withColumn("DecodedBody",
Decompress(col("body"))).select("DecodedBody")

# Parse json message from Event Hubs body, assuming the raw
document is stored in the data field, and the others fields hold
some metadata about it

schema = StructType([ \
    StructField("transactionId", LongType(),True), \
    StructField("timestamp",TimestampType(),True), \
    StructField("providerName", StringType(),True), \
    StructField("document", StringType(),True), \
    StructField("documentType", StringType(),True)
  ])

parsed_body_df = decoded_body_df.withColumn("jsonBody",
from_json(col("DecodedBody"), schema)).select("jsonBody")
```

Actual data processing consists of two steps. The first is to extract metadata to assist searching the raw documents after processing. Actual metadata depends on the use case, but generalizable examples would be relevant dates and identifiers, document types, source service, and any type of category:

```
# Code tested in Databricks with Delta Lake 1.0

df = parsed_body_df \
    .withColumn("transactionId",
parsed_body_df.jsonBody.transactionId) \
    .withColumn("timestamp", parsed_body_df.jsonBody.timestamp) \
    .withColumn("providerName",
parsed_body_df.jsonBody.providerName) \
    .withColumn("data", parsed_body_df.jsonBody.data)
    .withColumn("documentType",
parsed_body_df.jsonBody.documentType)
```

The second processing step is to generate a path to Azure Data Lake Storage Gen2, where you'll store raw documents:

```
# Code tested in Databricks with Delta Lake 1.0

# A function to generate a path
def GetPathFunction(timeStamp, transactionId, providerName,
Suffix='', Extension=".gz"):
  yy = timeStamp.year
  mm = timeStamp.month
  dd = timeStamp.day
  hh = timeStamp.hour
  mn = timeStamp.minute
  Suffix = f"{Suffix}_" if Suffix != '' else ''
  Name = f"{Suffix}{providerName}{Extension}"
  path = f"/{yy}/{mm}/{dd}/{hh}/{mn}/{transactionId}/{Name}"
  return path

GetPath = udf(lambda timestamp, transactionId, providerName,
suffix, extension: GetPathFunction(timestamp, transactionId,
providerName, suffix, extension), StringType())

df = df.withColumn("path", GetPath(col("timestamp"),
col("transactionId"), col("providerName"), col('documentType')))
```

Metadata ingestion in a delta lake

Metadata is written to a delta table that enables real-time query capabilities. Writes are streamed in a buffer, and queries to the table can merge results from the buffer with those from the historical portion of the table.

The following code snippet shows how to define a delta table in the metastore and partition it by date:

```
# Code tested in Databricks with Delta Lake 1.0

DeltaTable.create(spark) \
   .tableName("metadata") \
   .addColumn("transactionId", LongType()) \
   .addColumn("date", TimestampType()) \
   .addColumn("providerName", StringType()) \
   .addColumn("documentType", StringType()) \
   .addColumn("path", StringType()) \
   .partitionedBy("date") \
   .execute()
```

Note that the transactionId field is numeric. Typical messages passing distributed systems might use GUIDs to uniquely identify transactions instead. However, numeric data types enable greater query performance in most data platforms.

Assigning a unique transaction identifier might be challenging given the distributed nature of cloud data platforms (such as Spark). A useful approach is to base such a transaction identifier on a partition identifier (like the Event Hubs partition number) and a within-partition incremental number. An example of this approach is monotonically_increasing_id() in Azure Databricks.

The following code snippet shows how to append the stream with metadata of raw documents to the delta table:

```
# Code tested in Databricks with Delta Lake 1.0

df.withColumn("date", col("timeStamp").cast(DateType())) \
    .select("transactionId", "date", "providerName",
"documentType", "path") \
    .writeStream.format("delta") \
    .outputMode("append") \
    .option("checkpointLocation",
"/delta/metadata/_checkpoints/metadata_checkpoint") \
    .table("metadata")
```

Note that partitioning is managed while writing the stream according to the table schema.

Data ingestion in a data lake

Actual raw documents are written to an appropriate storage performance tier in Azure Data Lake Gen2.

The following code snippet shows a simple function to upload a file to Azure Data Lake Store Gen2; using a *foreach* method in the `DataStreamWriter` class allows you to upload the file hosted in each record of the streaming DataFrame:

```
# Code tested in Databricks with Delta Lake 1.0

from azure.storage.filedatalake import DataLakeServiceClient

def upload_data(storage_account_name, storage_account_key,
file_system_name, file_path, data):

  service_client =
DataLakeServiceClient(account_url="{}://{}.dfs.core.windows.net".
format("https", storage_account_name),
credential=storage_account_key)

  file_system_client =
service_client.get_file_system_client(file_system_name)
  file_client =
service_client.get_file_client(file_system_client.file_system_nam
e, file_path)

  if not file_client.exists:
    file_client.create_file()

  file_client.upload_data(data, overwrite=True)

# Process a row to upload data to ADLS
def Row2ADLS(row):
  upload_data(adls_name, adls_key, adls_container, row['path'],
row['data'])

df.writeStream.foreach(Row2ADLS).start()
```

Client

The client can be a custom web application that uses metadata to retrieve document paths from the delta table with standard SQL statements and, in turn, the actual document from the data lake with standard Azure Data Lake Storage Gen2 APIs.

The following code snippet, for example, shows how to retrieve the paths of all the documents in a certain transaction:

```
select * from metadata where transactionId = '123456'
```

Next steps

- Delta Lake
- Delta Lake in Azure Synapse Analytics
- Delta Lake in Azure Databricks
- Spark Structured Streaming
- Azure Synapse Analytics
- What is Delta Lake in Azure Synapse Analytics
- Azure Databricks Delta Lake and Delta Engine guide
- Describe Azure Databricks Delta Lake architecture

Related resources

See the related architectural guidance:

- Big data architectures
- Data lakes
- Choose an analytical data store in Azure

See these related architectures:

- Modern analytics architecture with Azure Databricks
- Modern data warehouse for small and medium business
- Ingestion, ETL, and stream processing pipelines with Azure Databricks
- Data science and machine learning with Azure Databricks

Custom business processes

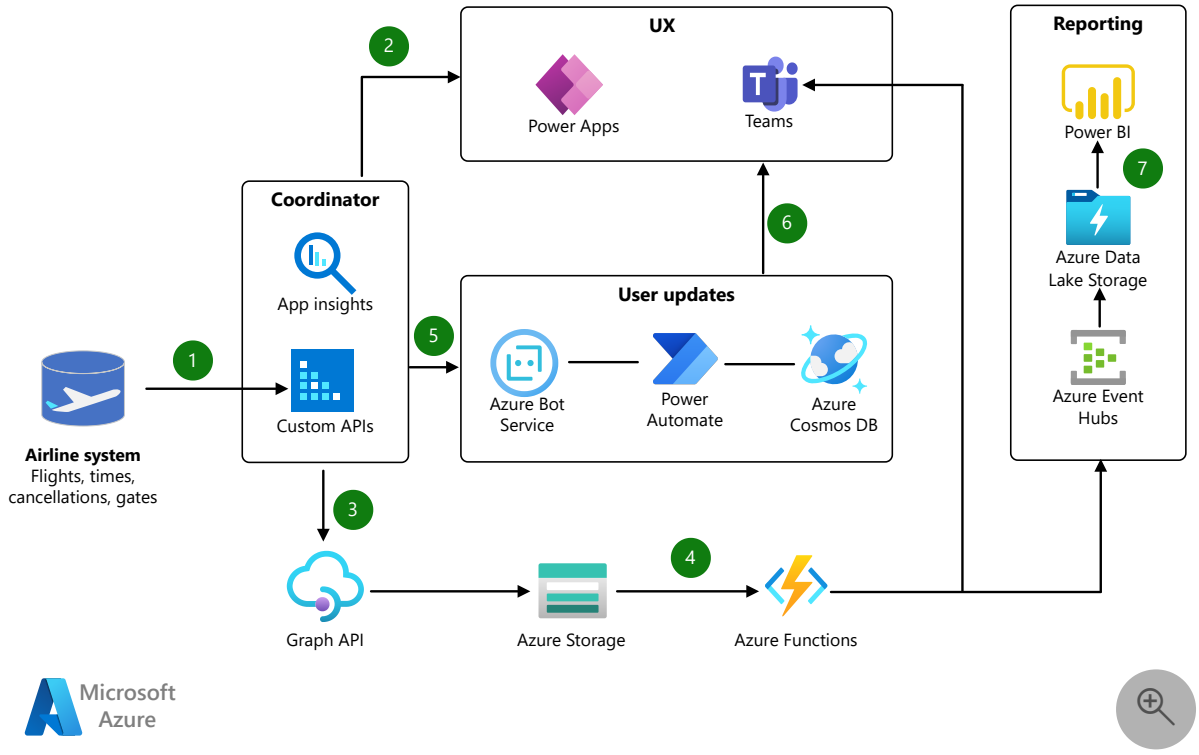
Azure API Management | Azure AI Bot Service | Azure Functions | Power Apps | Power BI | Power Automate

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 demonstrates how you can deploy portals that automate manual or paper-based processes and surface a rich user experience. Employ Azure API management and Azure Functions to connect custom APIs, which tap into your legacy systems. By using Azure-managed databases and a low-code approach to automating tasks, you can lower the overall solution costs while quickly building apps that are real-time, resilient, and scalable by default.

Architecture



Download a Visio file of this architecture.

Dataflow

The data flows through the solution as follows:

1. The airline system communicates with a custom API hosted in Azure API Management.
2. A custom API coordinator receives notifications and handles incoming messages from the airline system. It sends them to Power Apps, where flights are assigned to Microsoft Teams channels.
3. When a user selects a flight to monitor, or when the system assigns the user to a flight, the system queues a Graph API call in an Azure Storage Account queue for further processing.
4. Azure Functions runs the Graph API calls based on the incoming messages in the storage queue, sending notifications to Teams, and also streams all events to an Azure Event Hubs for further analytics.
5. The airline's notification system is managed by a custom bot messaging service that employs Azure Bot Service.
6. Custom bots send flight updates to users in Teams.
7. An Azure Data Lake storage offers long-term retention and micro-batch processing of events from Event Hubs, ultimately generating insightful reports with Power BI.

Components

Data is loaded from these different data sources using several Azure components:

- Power Apps: Increase agility across your organization by rapidly building low-code apps that modernize processes and solve tough challenges.
- Azure Functions: Accelerate and simplify application development with serverless compute
- Azure API management: Hybrid, multicloud management platform for APIs across all environments
- Azure SQL Database: Build apps that scale with the pace of your business with managed and intelligent SQL in the cloud
- Azure Cosmos DB: Fast NoSQL database with open APIs for any scale
- Azure Storage: Massively scalable, secure cloud storage for your data, apps, and workloads
- Azure Event Hubs: Simple, secure, and scalable real-time data ingestion
- Azure Data Lake Storage: Massively scalable and secure data lake for your high-performance analytics workloads
- Azure Bot Service: A managed service purpose-built for bot development

- Azure Data Factory: Hybrid data integration service that simplifies ETL at scale
- Power Automate: Streamline repetitive tasks and paperless processes
- Power BI is a suite of business analytics tools to analyze data and share insights.

Scenario details

Potential use cases

The use of digital workflows isn't limited to any one industry. Document-based digital workflows use the same components but arrange them differently to meet the requirements of a process. Examples of industries that can benefit from automated processes include, but are not limited to:

- Manufacturing
- Healthcare
- Education
- Finance
- Law firms
- Airlines (aerospace)

Next steps

- What is Power BI?
- Introduction to Azure Functions
- About Azure API Management
- What is Azure SQL Database?
- Welcome to Azure Cosmos DB
- What is Azure Event Hubs?
- Introduction to Azure Data Lake Storage
- Azure Bot Service
- What is Azure Data Factory?
- Introduction to Power Apps

Related resources

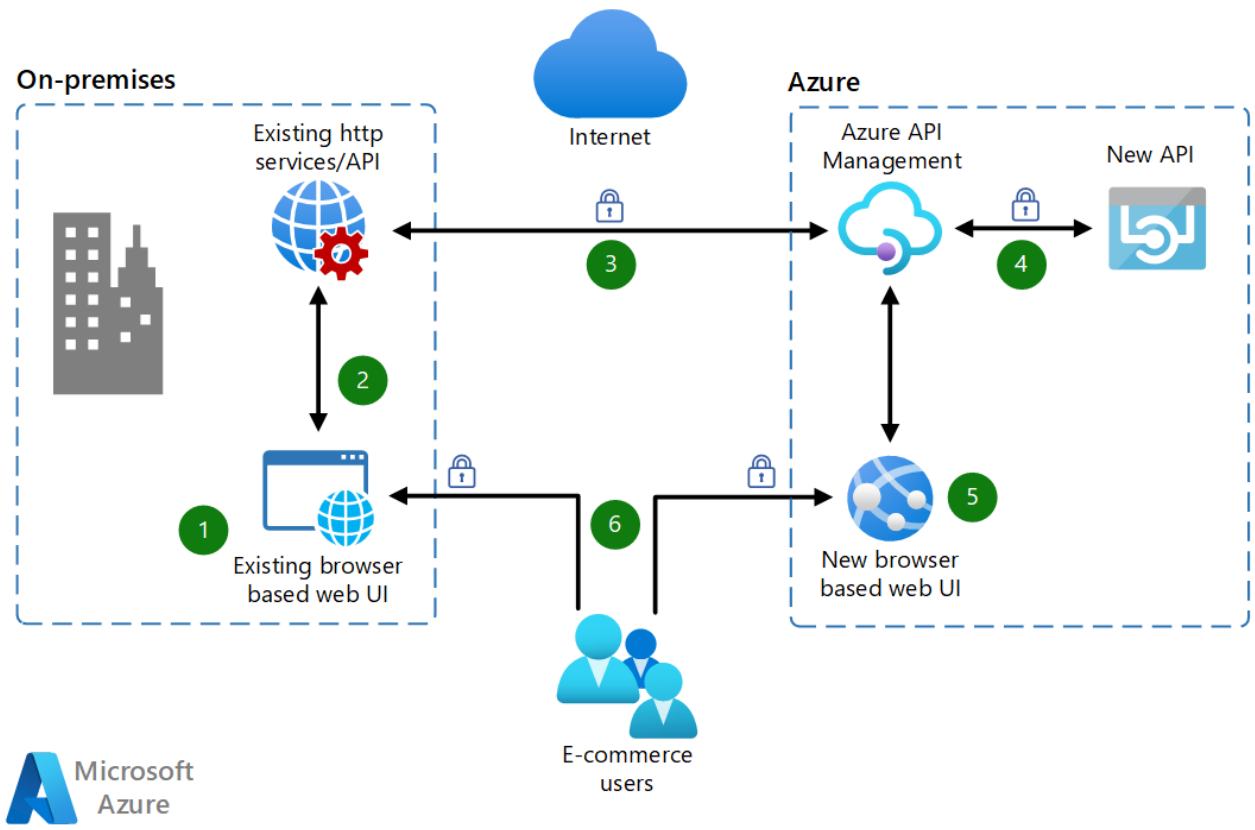
- Design great API developer experiences using API Management and GitHub

Migrate a web app by using Azure API Management

Azure API Management | Azure Monitor | Azure App Service

In this scenario, an e-commerce company in the travel industry migrates a legacy web application by using Azure API Management. The new UI will be hosted as a platform as a service (PaaS) application on Azure, and it will depend on both existing and new HTTP APIs. These APIs will ship with a better-designed set of interfaces, which will enable better performance, easier integration, and future extensibility.

Architecture



Download a Visio file of this architecture.

Workflow

1. The existing on-premises web application continues to directly consume the existing on-premises web services.
2. Calls from the existing web app to the existing HTTP services remain unchanged. These calls are internal to the corporate network.

3. Inbound calls are made from Azure to the existing internal services:

    - The security team allows traffic from the API Management instance to pass through the corporate firewall to the existing on-premises services by using secure transport (HTTPS or SSL).
    - The operations team allows inbound calls to the services only from the API Management instance. It meets this requirement by adding the IP address of the API Management instance to the allowlist within the corporate network perimeter.
    - A new module is configured into the on-premises request pipeline for HTTP services (to act on *only* connections that originate externally). The pipeline validates a certificate that API Management provides.

4. The new API:

    - Is surfaced only through the API Management instance, which provides the API facade. The new API isn't accessed directly.
    - Is developed and published as an Azure PaaS Web API app.
    - Is configured (via settings for the Web Apps feature of Azure App Service) to accept only the API Management virtual IP.
    - Is hosted in Web Apps with secure transport (HTTPS or SSL) turned on.
    - Has authorization enabled, provided by Azure App Service via Microsoft Entra ID and OAuth 2.

5. The new browser-based web application depends on the Azure API Management instance for *both* the existing HTTP API and the new API.

The API Management instance is configured to map the legacy HTTP services to a new API contract. In this configuration, the new Web UI is unaware of the integration with a set of legacy services/APIs and new APIs.

In the future, the project team will gradually port functionality to the new APIs and retire the original services. The team will handle these changes within API Management configuration, leaving the front-end UI unaffected and avoiding redevelopment work.

Components

- Azure API Management
- Azure App Service

Alternatives

- If the organization plans to move its infrastructure entirely to Azure, including the virtual machines (VMs) that host the legacy applications, API Management is still a great option because it can act as a facade for any addressable HTTP endpoint.

- If the organization had decided to keep the existing endpoints private and not expose them publicly, the organization's API Management instance could be linked to an Azure virtual network:
  - In an Azure "lift and shift" scenario linked to a deployed Azure virtual network, the organization could directly address the back-end service through private IP addresses.
  - In the on-premises scenario, the API Management instance could reach back to the internal service privately via an Azure VPN gateway and site-to-site IPSec VPN connection or Azure ExpressRoute. This scenario would then become a hybrid of Azure and on-premises.

- The organization can keep the API Management instance private by deploying it in internal mode. The organization can then use deployment with Azure Application Gateway to enable public access for some APIs while others remain internal. For more information, see Integrate API Management in an internal virtual network with Application Gateway.

- The organization might decide to host its APIs on-premises. One reason for this change might be that the organization couldn't move downstream database dependencies that are in scope for this project to the cloud. If that's the case, the organization can still take advantage of API Management locally by using a self-hosted gateway.

  The self-hosted gateway is a containerized deployment of the API Management gateway that connects back to Azure on an outbound socket. The first prerequisite is that self-hosted gateways can't be deployed without a parent resource in Azure, which carries an additional charge. Second, the Premium tier of API Management is required.

> **Note**
>
> For general information on connecting API Management to a virtual network, see **this article**.

Scenario details

An e-commerce company in the travel industry is modernizing its legacy browser-based software stack. Although the existing stack is mostly monolithic, some SOAP-based HTTP services exist from a recent project. The company is considering the creation of additional revenue streams to monetize some of the internal intellectual property that it has developed.

Goals for the project include addressing technical debt, improving ongoing maintenance, and accelerating feature development with fewer regression bugs. The project will use an iterative process to avoid risk, with some steps performed in parallel:

- The development team will modernize the application's back end, which consists of relational databases hosted on VMs.
- The in-house development team will write new business functionality that will be exposed over new HTTP APIs.
- A contract development team will build a new browser-based UI, which will be hosted in Azure.

New application features will be delivered in stages. These features will gradually replace the existing browser-based client/server UI functionality (hosted on-premises) that now powers the company's e-commerce business.

Members of the management team don't want to modernize unnecessarily. They also want to maintain control of scope and costs. To do this, they've decided to preserve their existing SOAP HTTP services. They also intend to minimize changes to the existing UI. They can use Azure API Management to address many of the project's requirements and constraints.

Potential use cases

This scenario highlights modernizing legacy browser-based software stacks.

You can use this scenario to:

- See how your business can benefit from using the Azure ecosystem.
- Plan for migrating services to Azure.
- Learn how a shift to Azure would affect existing APIs.

Considerations

These considerations implement the pillars of the Azure Well-Architected Framework, which is a set of guiding tenets that help improve the quality of a workload. For more information, see Microsoft Azure Well-Architected Framework.

Availability and scalability

- You can scale out Azure API Management by choosing a pricing tier and then adding units.
- Scaling can also happen automatically with autoscaling.
- Deploying across multiple regions enables failover options and can be done in the Premium tier.
- Consider Integrating with Azure Application Insights, which also surfaces metrics through Azure Monitor for monitoring.

Cost optimization

Cost optimization is about finding ways to reduce unnecessary expenses and improve operational efficiencies. For more information, see Overview of the cost optimization pillar.

API Management is offered in four tiers: Developer, Basic, Standard, and Premium. For detailed guidance on the differences in these tiers, see the Azure API Management pricing guidance.

You can scale API Management by adding and removing units. Each unit has capacity that depends on its tier.

> **Note**
>
> You can use the Developer tier for evaluation of the API Management features. Don't use it for production.

To view projected costs and customize to your deployment needs, you can modify the number of scale units and App Service instances in the Azure pricing calculator.

Deploy this scenario

To get started, create an Azure API Management instance in the portal.

Alternatively, you can choose from an existing Azure Resource Manager quickstart template that aligns to your specific use case.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- Ben Gimblett | Senior Customer Engineer

To see non-public LinkedIn profiles, sign in to LinkedIn.

Next steps

Product documentation:

- Azure App Service overview
- Azure API Management overview

Learn modules:

- Explore Azure App Service
- Deploy a website to Azure with Azure App Service
- Protect your APIs on Azure API Management

Related resources

- Architect scalable e-commerce web apps
- Design great API developer experiences using API Management and GitHub
- DevTest and DevOps for PaaS solutions

Predictive aircraft engine monitoring

Azure Data Factory | Azure Event Hubs | Azure HDInsight | Azure Machine Learning | Azure Stream Analytics | Azure Monitor

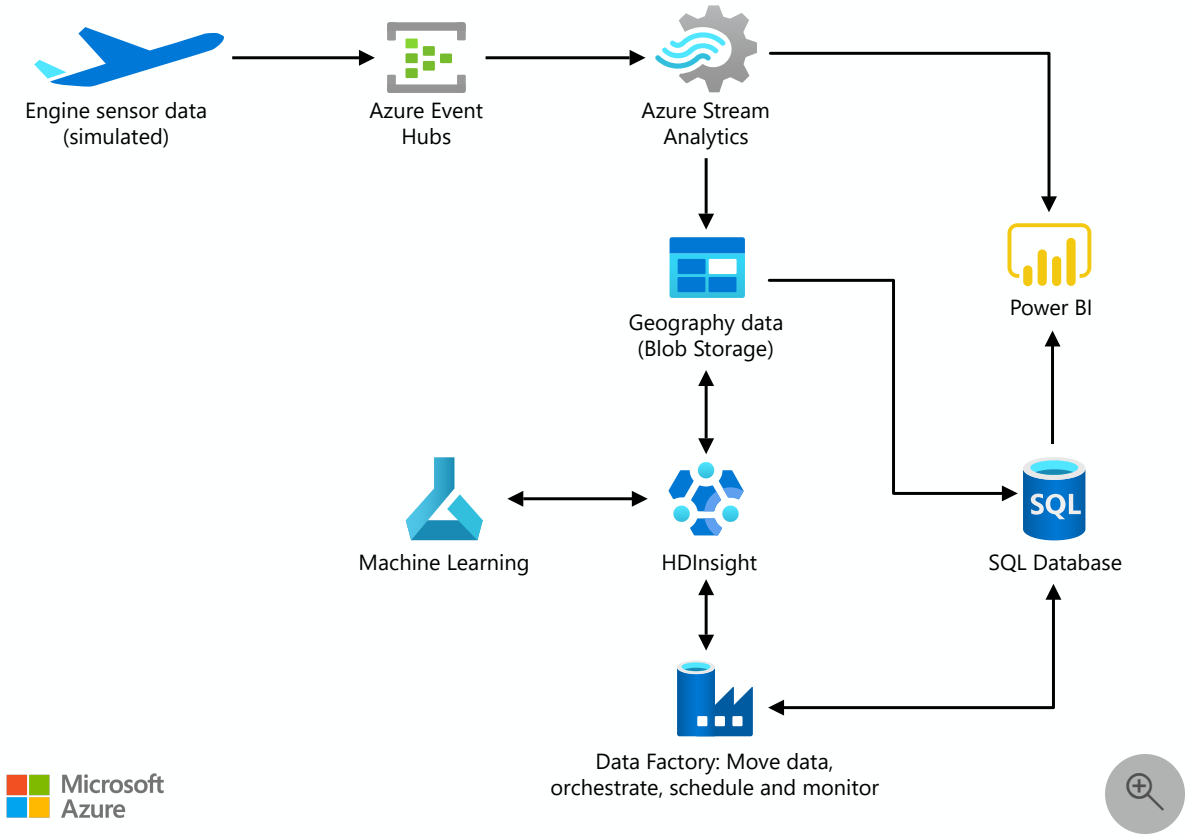
Solution ideas

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**.

Microsoft Azure's Predictive Maintenance solution demonstrates how to combine real-time aircraft data with analytics to monitor aircraft health.

This solution is built with Azure Stream Analytics, Event Hubs, Azure Machine Learning, HDInsight, Azure SQL Database, Data Factory, and Power BI. These services run in a high-availability environment, patched and supported, allowing you to focus on your solution instead of the environment they run in.

Architecture



Download a Visio file of this architecture.

Components

- Azure Stream Analytics provides near real-time analytics on the input stream from Azure Event Hubs. Input data is filtered and passed to a Machine Learning endpoint, finally sending the results to the Power BI dashboard.
- Event Hubs ingests raw assembly-line data and passes it on to Stream Analytics.
- Azure Machine Learning predicts potential failures based on real-time assembly-line data from Stream Analytics.
- HDInsight runs Hive scripts to provide aggregations on the raw events that were archived by Stream Analytics.
- Azure SQL Database stores prediction results received from Machine Learning and publishes data to Power BI.
- Data Factory handles orchestration, scheduling, and monitoring of the batch processing pipeline.
- Power BI enables visualization of real-time assembly-line data from Stream Analytics and the predicted failures and alerts from Data Warehouse.

Scenario details

Potential use cases

This solution is ideal for the aircraft and aerospace industries.

With the right information, it's possible to determine the condition of equipment in order to predict when maintenance should be performed. Predictive maintenance can be used for the following items:

- Real-time diagnostics.
- Real-time flight assistance.
- Prognostics.
- Cost reduction.

Next steps

See product documentation:

- Stream Analytics
- Event Hubs
- Azure Machine Learning
- HDInsight
- SQL Database
- Azure Data Factory
- Power BI

Related resources

Read other Azure Architecture Center articles about predictive maintenance and prediction with machine learning:

- Predictive maintenance
- Predictive maintenance for industrial IoT
- Predict length of stay and patient flow

Public Holidays

Article • 12/05/2023

Worldwide public holiday data sourced from PyPI holidays package and Wikipedia, covering 38 countries or regions from 1970 to 2099.

Each row indicates the holiday info for a specific date, country or region, and whether most people have paid time off.

> **Note**
>
> Microsoft provides Azure Open Datasets on an “as is” basis. Microsoft makes no warranties, express or implied, guarantees or conditions with respect to your use of the datasets. To the extent permitted under your local law, Microsoft disclaims all liability for any damages or losses, including direct, consequential, special, indirect, incidental or punitive, resulting from your use of the datasets.
>
> This dataset is provided under the original terms that Microsoft received source data. The dataset may include data sourced from Microsoft.

Volume and retention

This dataset is stored in Parquet format. It's a snapshot with holiday information from January 1, 1970 to January 1, 2099. The data size is about 500KB.

Storage location

This dataset is stored in the East US Azure region. We recommend locating compute resources in East US for affinity.

Additional information

This dataset combines data sourced from Wikipedia (WikiMedia Foundation Inc) and PyPI holidays package.

- Wikipedia: original source, original license
- PyPI holidays: original source, original license

The combined dataset is provided under the Creative Commons Attribution-ShareAlike 3.0 Unported License.

Email aod@microsoft.com if you have any questions about the data source.

Columns

Expand table

Name	Data type	Unique	Values (sample)	Description
countryOrRegion	string	38	Sweden Norway	Country or region full name.
countryRegionCode	string	35	SE NO	Country or region code following the format here.
date	timestamp	20,665	2074-01-01 00:00:00 2025-12-25 00:00:00	Date of the holiday.
holidayName	string	483	Søndag Söndag	Full name of the holiday.
isPaidTimeOff	boolean	3	True	Indicate whether most people have paid time off on this date (only available for US, GB, and India now). If it is NULL, it means unknown.
normalizeHolidayName	string	438	Søndag Söndag	Normalized name of the holiday.

Preview

Expand table

countryOrRegion	holidayName	normalizeHolidayName	countryRegionCode	date
Norway	Søndag	Søndag	NO	12/28/2098 12:00:00 AM
Sweden	Söndag	Söndag	SE	12/28/2098 12:00:00

countryOrRegion	holidayName	normalizeHolidayName	countryRegionCode	date
				AM
Australia	Boxing Day	Boxing Day	AU	12/26/2098 12:00:00 AM
Hungary	Karácsony másnapja	Karácsony másnapja	HU	12/26/2098 12:00:00 AM
Austria	Stefanitag	Stefanitag	AT	12/26/2098 12:00:00 AM
Canada	Boxing Day	Boxing Day	CA	12/26/2098 12:00:00 AM
Croatia	Sveti Stjepan	Sveti Stjepan	HR	12/26/2098 12:00:00 AM
Czech	2. svátek vánoční	2. svátek vánoční	CZ	12/26/2098 12:00:00 AM

Data access

Azure Notebooks

azureml-opendatasets

```
# This is a package in preview.
from azureml.opendatasets import PublicHolidays

from datetime import datetime
from dateutil import parser
from dateutil.relativedelta import relativedelta


end_date = datetime.today()
start_date = datetime.today() - relativedelta(months=1)
```

```
hol = PublicHolidays(start_date=start_date, end_date=end_date)
hol_df = hol.to_pandas_dataframe()
```

```

hol_df.info()
```

Azure Databricks

azureml-opendatasets

```

# This is a package in preview.
# You need to pip install azureml-opendatasets in Databricks cluster.
https://learn.microsoft.com/azure/data-explorer/connect-from-
databricks#install-the-python-library-on-your-azure-databricks-cluster
from azureml.opendatasets import PublicHolidays

from datetime import datetime
from dateutil import parser
from dateutil.relativedelta import relativedelta


end_date = datetime.today()
start_date = datetime.today() - relativedelta(months=1)
hol = PublicHolidays(start_date=start_date, end_date=end_date)
hol_df = hol.to_spark_dataframe()
```

```

display(hol_df.limit(5))
```

Azure Synapse

azureml-opendatasets

Python

```
# This is a package in preview.
from azureml.opendatasets import PublicHolidays

from datetime import datetime
```

```python
from dateutil import parser
from dateutil.relativedelta import relativedelta


end_date = datetime.today()
start_date = datetime.today() - relativedelta(months=1)
hol = PublicHolidays(start_date=start_date, end_date=end_date)
hol_df = hol.to_spark_dataframe()
```

Python

```python
# Display top 5 rows
display(hol_df.limit(5))
```

Next steps

View the rest of the datasets in the Open Datasets catalog.

Russian Open Speech To Text

Article • 04/19/2022

A collection of speech samples derived from various audio sources. The dataset contains short audio clips in Russian.

> **Note**
>
> Microsoft provides Azure Open Datasets on an "as is" basis. Microsoft makes no warranties, express or implied, guarantees or conditions with respect to your use of the datasets. To the extent permitted under your local law, Microsoft disclaims all liability for any damages or losses, including direct, consequential, special, indirect, incidental or punitive, resulting from your use of the datasets.
>
> This dataset is provided under the original terms that Microsoft received source data. The dataset may include data sourced from Microsoft.

This Russian speech to text (STT) dataset includes:

- ~16 million utterances
- ~20,000 hours
- 2.3 TB (uncompressed in .wav format in int16), 356G in opus
- All files were transformed to opus, except for validation datasets

The main purpose of the dataset is to train speech-to-text models.

Dataset composition

Dataset size is given for .wav files.

DATASET	UTTERANCES	HOURS	GB	SECS/CHARS	COMMENT	ANNOTATION	QUALITY/NOISE
radio_v4 (*)	7,603,192	10,430	1,195	5s / 68	Radio	Align	95% / crisp
public_speech (*)	1,700,060	2,709	301	6s / 79	Public speech	Align	95% / crisp
audiobook_2	1,149,404	1,511	162	5s / 56	Books	Align	95% / crisp
radio_2	651,645	1,439	154	8s / 110	Radio	Align	95% / crisp
public_youtube1120	1,410,979	1,104	237	3s / 34	Youtube	Subtitles	95% / ~crisp
public_youtube700	759,483	701	75	3s / 43	Youtube	Subtitles	95% / ~crisp
tts_russian_addresses	1,741,838	754	81	2s / 20	Addresses	TTS 4 voices	100% / crisp
asr_public_phone_calls_2	603,797	601	66	4s / 37	Phone calls	ASR	70% / noisy
public_youtube1120_hq	369,245	291	31	3s / 37	YouTube HQ	Subtitles	95% / ~crisp
asr_public_phone_calls_1	233,868	211	23	3s / 29	Phone calls	ASR	70% / noisy
radio_v4_add (*)	92,679	157	18	6s / 80	Radio	Align	95% / crisp

DATASET	UTTERANCES	HOURS	GB	SECS/CHARS	COMMENT	ANNOTATION	QUALITY/NOISE
asr_public_stories_2	78,186	78	9	4s / 43	Books	ASR	80% / crisp
asr_public_stories_1	46,142	38	4	3s / 30	Books	ASR	80% / crisp
public_series_1	20,243	17	2	3s / 38	Youtube	Subtitles	95% / ~crisp
asr_calls_2_val	12,950	7,7	2	2s / 34	Phone calls	Manual annotation	99% / crisp
public_lecture_1	6,803	6	1	3s / 47	Lectures	Subtitles	95% / crisp
buriy_audiobooks_2_val	7,850	4,9	1	2s / 31	Books	Manual annotation	99% / crisp
public_youtube700_val	7,311	4,5	1	2s / 35	Youtube	Manual annotation	99% / crisp

(*) Only a sample of data is provided with txt files.

Annotation methodology

The dataset is compiled using open sources. Long sequences are split into audio chunks using voice activity detection and alignment. Some audio types are annotated automatically and verified statistically by using heuristics.

Data volumes and update frequency

The total size of the dataset is 350 GB. The total size of the dataset with publicly shared labels is 130 GB.

The dataset itself is not likely to be updated for backwards compatibility. Follow the original repository for benchmarks and exclude files.

New domains and languages may be added in future.

Audio normalization

All files are normalized for easier and faster runtime augmentations. Processing is as follows:

- Converted to mono, if necessary;
- Converted to 16-kHz sampling rate, if necessary;
- Stored as 16-bit integers;
- Converted to OPUS;

On disk DB methodology

Each audio file (wav, binary) is hashed. The hash is used to create a folder hierarchy for more optimal fs operation.

```
Python
```

```
target_format = 'wav'
wavb = wav.tobytes()

f_hash = hashlib.sha1(wavb).hexdigest()

store_path = Path(root_folder,
                  f_hash[0],
                  f_hash[1:3],
                  f_hash[3:15] + '.' + target_format)
```

Downloads

The dataset is provided in two forms:

- Archives available via Azure blob storage and/or direct links;
- Original files available via Azure blob storage; Everything is stored in 'https://azureopendatastorage.blob.core.windows.net/openstt/'

Folder structure:

```
└── ru_open_stt_opus                                        <= archived folders
│   │
│   ├── archives
│   │    ├── asr_calls_2_val.tar.gz                         <= tar.gz archives with opus
and wav files
│   │    │   ...                                            <= see the below table for
enumeration
│   │    └── tts_russian_addresses_rhvoice_4voices.tar.gz
│   │
│   └── manifests
│        ├── asr_calls_2_val.csv                            <= csv files with wav_path,
text_path, duration (see notebooks)
│        │   ...
│        └── tts_russian_addresses_rhvoice_4voices.csv
│
└── ru_open_stt_opus_unpacked                               <= a separate folder for each
uploaded domain
    ├── public_youtube1120
    │    ├── 0                                              <= see "On disk DB
methodology" for details
    │    ├── 1
    │    │   ├── 00
    │    │   │  ...
    │    │   └── ff
    │    │        ├── *.opus                               <= actual files
    │    │        └── *.txt
    │    │   ...
    │    └── f
    │
    ├── public_youtube1120_hq
    ├── public_youtube700_val
    ├── asr_calls_2_val
    ├── radio_2
    ├── private_buriy_audiobooks_2
    ├── asr_public_phone_calls_2
    ├── asr_public_stories_2
    ├── asr_public_stories_1
    ├── public_lecture_1
```

```
    ├── asr_public_phone_calls_1
    ├── public_series_1
    └── public_youtube700
```

DATASET	GB, WAV	GB, ARCHIVE	ARCHIVE	SOURCE	MANIFEST
Train					
Radio and public speech sample	-	11.4	opus+txt	-	manifest
audiobook_2	162	25.8	opus+txt	Internet + alignment	manifest
radio_2	154	24.6	opus+txt	Radio	manifest
public_youtube1120	237	19.0	opus+txt	YouTube videos	manifest
asr_public_phone_calls_2	66	9.4	opus+txt	Internet + ASR	manifest
public_youtube1120_hq	31	4.9	opus+txt	YouTube videos	manifest
asr_public_stories_2	9	1.4	opus+txt	Internet + alignment	manifest
tts_russian_addresses_rhvoice_4voices	80.9	12.9	opus+txt	TTS	manifest
public_youtube700	75.0	12.2	opus+txt	YouTube videos	manifest
asr_public_phone_calls_1	22.7	3.2	opus+txt	Internet + ASR	manifest
asr_public_stories_1	4.1	0.7	opus+txt	Public stories	manifest
public_series_1	1.9	0.3	opus+txt	Public series	manifest
public_lecture_1	0.7	0.1	opus+txt	Internet + manual	manifest
Val					
asr_calls_2_val	2	0.8	wav+txt	Internet	manifest
buriy_audiobooks_2_val	1	0.5	wav+txt	Books + manual	manifest
public_youtube700_val	2	0.13	wav+txt	YouTube videos + manual	manifest

Download instructions

Direct download

For instructions on how to download the dataset directly, see the GitHub download instructions page.

Additional information

For help or questions about the data contact the data author(s) at aveysov@gmail.com

This license allows users to distribute, remix, adapt, and build upon the material in any medium or format for noncommercial purposes only, and only so long as attribution is given to the creator. It includes the following elements:

- BY – Credit must be given to the creator
- NC – Only noncommercial uses of the work are permitted

CC-BY-NC and commercial usage available after agreement with dataset authors.

Data access

Azure Notebooks

azure-storage

> **Tip**
>
> Download the notebook instead.

Helper functions / dependencies

Building libsndfile

An efficient way to read opus files in Python that doesn't incur significant overhead is to use pysoundfile (a Python CFFI wrapper around libsoundfile).

Opus support has been implemented upstream, but it has not been properly released. Therefore, we opted for custom build + monkey patching.

Typically, you need to run this in your shell with sudo access:

```bash
apt-get update
apt-get install cmake autoconf autogen automake build-essential libasound2-dev \
libflac-dev libogg-dev libtool libvorbis-dev libopus-dev pkg-config -y

cd /usr/local/lib
git clone https://github.com/erikd/libsndfile.git
cd libsndfile
git reset --hard 49b7d61
mkdir -p build && cd build

cmake .. -DBUILD_SHARED_LIBS=ON
make && make install
cmake --build .
```

Helper functions / dependencies

Install the following libraries:

```
pandas
numpy
```

```
scipy
tqdm
soundfile
librosa
```

Manifests are csv files with the following columns:

- Path to audio
- Path to text file
- Duration

They proved to be the simplest format of accessing data.

For ease of use, all the manifests are already rerooted. All paths in them are relative, you need to provide a root folder.

Python

```python
# manifest utils
import os
import numpy as np
import pandas as pd
from tqdm import tqdm
from urllib.request import urlopen


def reroot_manifest(manifest_df,
                    source_path,
                    target_path):
    if source_path != '':
        manifest_df.wav_path = manifest_df.wav_path.apply(lambda x: x.replace(source_path,

target_path))
        manifest_df.text_path = manifest_df.text_path.apply(lambda x:
x.replace(source_path,

target_path))
    else:
        manifest_df.wav_path = manifest_df.wav_path.apply(lambda x:
os.path.join(target_path, x))
        manifest_df.text_path = manifest_df.text_path.apply(lambda x:
os.path.join(target_path, x))
    return manifest_df


def save_manifest(manifest_df,
                  path,
                  domain=False):
    if domain:
        assert list(manifest_df.columns) == ['wav_path', 'text_path', 'duration',
'domain']
    else:
        assert list(manifest_df.columns) == ['wav_path', 'text_path', 'duration']

    manifest_df.reset_index(drop=True).sort_values(by='duration',
                                                   ascending=True).to_csv(path,
                                                                          sep=',',
                                                                          header=False,
                                                                          index=False)
    return True
```

```python
def read_manifest(manifest_path,
                  domain=False):
    if domain:
        return pd.read_csv(manifest_path,
                           names=['wav_path',
                                  'text_path',
                                  'duration',
                                  'domain'])
    else:
        return pd.read_csv(manifest_path,
                           names=['wav_path',
                                  'text_path',
                                  'duration'])


def check_files(manifest_df,
                domain=False):
    orig_len = len(manifest_df)
    if domain:
        assert list(manifest_df.columns) == ['wav_path', 'text_path', 'duration']
    else:
        assert list(manifest_df.columns) == ['wav_path', 'text_path', 'duration',
'domain']
    wav_paths = list(manifest_df.wav_path.values)
    text_path = list(manifest_df.text_path.values)

    omitted_wavs = []
    omitted_txts = []

    for wav_path, text_path in zip(wav_paths, text_path):
        if not os.path.exists(wav_path):
            print('Dropping {}'.format(wav_path))
            omitted_wavs.append(wav_path)
        if not os.path.exists(text_path):
            print('Dropping {}'.format(text_path))
            omitted_txts.append(text_path)

    manifest_df = manifest_df[~manifest_df.wav_path.isin(omitted_wavs)]
    manifest_df = manifest_df[~manifest_df.text_path.isin(omitted_txts)]
    final_len = len(manifest_df)

    if final_len != orig_len:
        print('Removed {} lines'.format(orig_len-final_len))
    return manifest_df


def plain_merge_manifests(manifest_paths,
                          MIN_DURATION=0.1,
                          MAX_DURATION=100):

    manifest_df = pd.concat([read_manifest(_)
                             for _ in manifest_paths])
    manifest_df = check_files(manifest_df)

    manifest_df_fit = manifest_df[(manifest_df.duration>=MIN_DURATION) &
                                  (manifest_df.duration<=MAX_DURATION)]

    manifest_df_non_fit = manifest_df[(manifest_df.duration<MIN_DURATION) |
                                      (manifest_df.duration>MAX_DURATION)]

    print(f'Good hours: {manifest_df_fit.duration.sum() / 3600:.2f}')
    print(f'Bad hours: {manifest_df_non_fit.duration.sum() / 3600:.2f}')

    return manifest_df_fit
```

```
def save_txt_file(wav_path, text):
    txt_path = wav_path.replace('.wav','.txt')
    with open(txt_path, "w") as text_file:
        print(text, file=text_file)
    return txt_path


def read_txt_file(text_path):
    #with open(text_path, 'r') as file:
    response = urlopen(text_path)
    file = response.readlines()
    for i in range(len(file)):
        file[i] = file[i].decode('utf8')
    return file

def create_manifest_from_df(df, domain=False):
    if domain:
        columns = ['wav_path', 'text_path', 'duration', 'domain']
    else:
        columns = ['wav_path', 'text_path', 'duration']
    manifest = df[columns]
    return manifest


def create_txt_files(manifest_df):
    assert 'text' in manifest_df.columns
    assert 'wav_path' in manifest_df.columns
    wav_paths, texts = list(manifest_df['wav_path'].values),
list(manifest_df['text'].values)
    # not using multiprocessing for simplicity
    txt_paths = [save_txt_file(*_) for _ in tqdm(zip(wav_paths, texts),
total=len(wav_paths))]
    manifest_df['text_path'] = txt_paths
    return manifest_df


def replace_encoded(text):
    text = text.lower()
    if '2' in text:
        text = list(text)
        _text = []
        for i,char in enumerate(text):
            if char=='2':
                try:
                    _text.extend([_text[-1]])
                except:
                    print(''.join(text))
            else:
                _text.extend([char])
        text = ''.join(_text)
    return text
```

Python

```
# reading opus files
import os
import soundfile as sf



# Fx for soundfile read/write functions
def fx_seek(self, frames, whence=os.SEEK_SET):
    self._check_if_closed()
    position = sf._snd.sf_seek(self._file, frames, whence)
```

```
    return position


def fx_get_format_from_filename(file, mode):
    format = ''
    file = getattr(file, 'name', file)
    try:
        format = os.path.splitext(file)[-1][1:]
        format = format.decode('utf-8', 'replace')
    except Exception:
        pass
    if format == 'opus':
        return 'OGG'
    if format.upper() not in sf._formats and 'r' not in mode:
        raise TypeError("No format specified and unable to get format from "
                        "file extension: {0!r}".format(file))
    return format


#sf._snd = sf._ffi.dlopen('/usr/local/lib/libsndfile/build/libsndfile.so.1.0.29')
sf._subtypes['OPUS'] = 0x0064
sf.SoundFile.seek = fx_seek
sf._get_format_from_filename = fx_get_format_from_filename


def read(file, **kwargs):
    return sf.read(file, **kwargs)


def write(file, data, samplerate, **kwargs):
    return sf.write(file, data, samplerate, **kwargs)
```

Python

```
# display utils
import gc
from IPython.display import HTML, Audio, display_html
pd.set_option('display.max_colwidth', 3000)
#Prepend_path is set to read directly from Azure. To read from local replace below string
with path to the downloaded dataset files
prepend_path =
'https://azureopendatastorage.blob.core.windows.net/openstt/ru_open_stt_opus_unpacked/'


def audio_player(audio_path):
    return '<audio preload="none" controls="controls"><source src="{}" type="audio/wav">
</audio>'.format(audio_path)

def display_manifest(manifest_df):
    display_df = manifest_df
    display_df['wav'] = [audio_player(prepend_path+path) for path in display_df.wav_path]
    display_df['txt'] = [read_txt_file(prepend_path+path) for path in
tqdm(display_df.text_path)]
    audio_style = '<style>audio {height:44px;border:0;padding:0 20px 0px;margin:-10px
-20px -20px;}</style>'
    display_df = display_df[['wav','txt', 'duration']]
    display(HTML(audio_style + display_df.to_html(escape=False)))
    del display_df
    gc.collect()
```

Play with a dataset

Play a sample of files

Most platforms browsers support native audio playback. So we can use HTML5 audio players to view our data.

Python

```
manifest_df = read_manifest(prepend_path +'/manifests/public_series_1.csv')
#manifest_df = reroot_manifest(manifest_df,
                               #source_path='',
                               #target_path='../../../../../nvme/stt/data/ru_open_stt/')
```

Python

```
sample = manifest_df.sample(n=20)
display_manifest(sample)
```

Read a file

Python

```
!ls ru_open_stt_opus/manifests/*.csv
```

Some examples showing how to best read wav and opus files.

Scipy is the fastest for wav. Pysoundfile is the best overall for opus.

Python

```
%matplotlib inline

import librosa
from scipy.io import wavfile
from librosa import display as ldisplay
from matplotlib import pyplot as plt
```

Read a wav

Python

```
manifest_df = read_manifest(prepend_path +'manifests/asr_calls_2_val.csv')
#manifest_df = reroot_manifest(manifest_df,
                               #source_path='',
                               #target_path='../../../../../nvme/stt/data/ru_open_stt/')
```

Python

```
sample = manifest_df.sample(n=5)
display_manifest(sample)
```

Python

```
from io import BytesIO

wav_path = sample.iloc[0].wav_path
response = urlopen(prepend_path+wav_path)
data = response.read()
sr, wav = wavfile.read(BytesIO(data))
wav.astype('float32')
absmax = np.max(np.abs(wav))
wav =  wav / absmax
```

Python

```
# shortest way to plot a spectrogram
D = librosa.amplitude_to_db(np.abs(librosa.stft(wav)), ref=np.max)
plt.figure(figsize=(12, 6))
ldisplay.specshow(D, y_axis='log')
plt.colorbar(format='%+2.0f dB')
plt.title('Log-frequency power spectrogram')
# shortest way to plot an envelope
plt.figure(figsize=(12, 6))
ldisplay.waveplot(wav, sr=sr, max_points=50000.0, x_axis='time', offset=0.0, max_sr=1000,
ax=None)
```

Read opus

Python

```
manifest_df = read_manifest(prepend_path +'manifests/asr_public_phone_calls_2.csv')
#manifest_df = reroot_manifest(manifest_df,
                              #source_path='',
                              #target_path='../../../../../nvme/stt/data/ru_open_stt/')
```

Python

```
sample = manifest_df.sample(n=5)
display_manifest(sample)
```

Python

```
opus_path = sample.iloc[0].wav_path
response = urlopen(prepend_path+opus_path)
data = response.read()
wav, sr = sf.read(BytesIO(data))
wav.astype('float32')
absmax = np.max(np.abs(wav))
wav =  wav / absmax
```

Python

```
# shortest way to plot a spectrogram
D = librosa.amplitude_to_db(np.abs(librosa.stft(wav)), ref=np.max)
plt.figure(figsize=(12, 6))
ldisplay.specshow(D, y_axis='log')
plt.colorbar(format='%+2.0f dB')
plt.title('Log-frequency power spectrogram')
# shortest way to plot an envelope
```

```
plt.figure(figsize=(12, 6))
ldisplay.waveplot(wav, sr=sr, max_points=50000.0, x_axis='time', offset=0.0, max_sr=1000,
ax=None)
```

Next steps

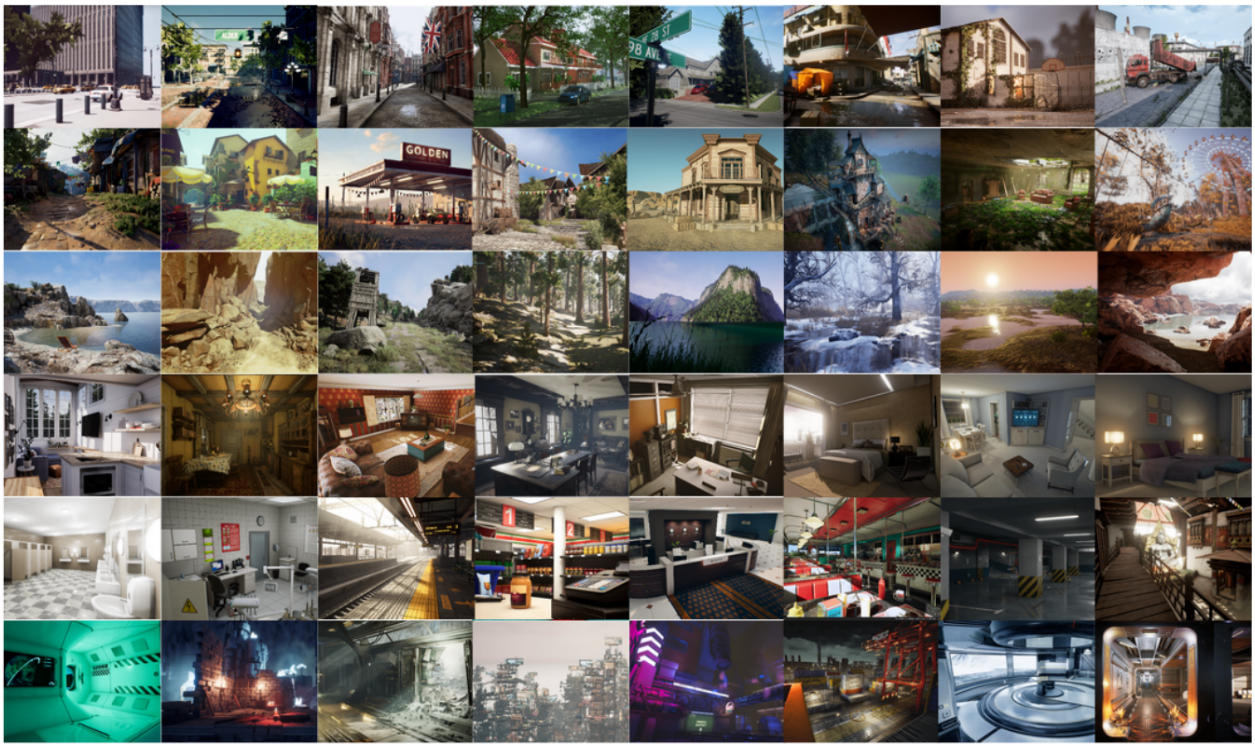
View the rest of the datasets in the Open Datasets catalog.

TartanAir: AirSim simulation dataset for simultaneous localization and mapping (SLAM)

Article • 04/19/2022

Simultaneous Localization and Mapping (SLAM) is one of the most fundamental capabilities necessary for robots. Due to the ubiquitous availability of images, Visual SLAM (V-SLAM) has become an important component of many autonomous systems. Impressive progress has been made with both geometric-based methods and learning-based methods. However, developing robust and reliable SLAM methods for real-world applications is still a challenging problem. Real-life environments are full of difficult cases such as light changes or lack of illumination, dynamic objects, and texture-less scenes. This dataset takes advantages of the advancing computer graphics technology, and aims to cover diverse scenarios with challenging features in simulation.

> **Note**
>
> Microsoft provides Azure Open Datasets on an “as is” basis. Microsoft makes no warranties, express or implied, guarantees or conditions with respect to your use of the datasets. To the extent permitted under your local law, Microsoft disclaims all liability for any damages or losses, including direct, consequential, special, indirect, incidental or punitive, resulting from your use of the datasets.
>
> This dataset is provided under the original terms that Microsoft received source data. The dataset may include data sourced from Microsoft.



The data is collected in photo-realistic simulation environments in the presence of various light conditions, weather, and moving objects. By collecting data in simulation, we are able to obtain multi-modal sensor data and precise ground truth labels, including the stereo RGB image, depth image, segmentation, optical flow, and camera poses. We set up a large number of environments with various styles and scenes, covering challenging viewpoints and diverse motion patterns, which are difficult to achieve by using physical data collection platforms. The four most important features of our dataset are: 1) Large-size diverse realistic data; 2) Multimodal ground truth labels; 3) Diversity of motion patterns; 4) Challenging Scenes.

This dataset provides five types of data:

- Stereo images: image type (PNG)
- Depth file: numpy type (NPY)
- Segmentation file: numpy type (NPY)
- Optical flow file: numpy type (NPY)
- Camera pose file: text type (TXT)

It is collected from different environments, contains hundreds of trajectories (3 TB) in total as of 2019.

Challenging visual effects

In some simulations, The dataset simulates multiple types of challenging visual effects.

- Hard lighting conditions. Day-night alternating. Low-lighting. Rapidly changing illuminations.
- Weather effects. Clear, raining, snowing, windy, and fog.
- Seasonal change.

Storage location

This dataset is stored in the East US Azure region. Allocating compute resources in East US is recommended for affinity.

License terms

This project is released under the MIT License. Review the License file for more details.

Additional information

View the official TartanAir website or view the original research paper.

Email tartanair@hotmail.com if you have any questions about the data source. You can also reach out to contributers on the associated GitHub.

Citation More technical details are available in AirSim paper (FSR 2017 Conference). Cite this as:

```
@article{tartanair2020arxiv,
  title =   {TartanAir: A Dataset to Push the Limits of Visual SLAM},
  author =  {Wenshan Wang, Delong Zhu, Xiangwei Wang, Yaoyu Hu, Yuheng Qiu,
Chen Wang, Yafei Hu, Ashish Kapoor, Sebastian Scherer},
  journal = {arXiv preprint arXiv:2003.14338},
  year =    {2020},
  url = {https://arxiv.org/abs/2003.14338}
}
```

```
@inproceedings{airsim2017fsr,
  author = {Shital Shah and Debadeepta Dey and Chris Lovett and Ashish
Kapoor},
  title = {AirSim: High-Fidelity Visual and Physical Simulation for
Autonomous Vehicles},
  year = {2017},
  booktitle = {Field and Service Robotics},
  eprint = {arXiv:1705.05065},
```

```
  url = {https://arxiv.org/abs/1705.05065}
}
```

Data access

Use the following code sample to access the data in a Python notebook.

Dependencies

```bash
pip install numpy
pip install azure-storage-blob
pip install opencv-python
```

Imports and container client

```python
from azure.storage.blob import ContainerClient
import numpy as np
import io
import cv2
import time
import matplotlib.pyplot as plt
%matplotlib inline

# Dataset website: http://theairlab.org/tartanair-dataset/
account_url = 'https://tartanair.blob.core.windows.net/'
container_name = 'tartanair-release1'

container_client = ContainerClient(account_url=account_url,
                                   container_name=container_name,
                                   credential=None)
```

Environments and trajectories

```python
def get_environment_list():
    '''
    List all the environments shown in the root directory
    '''
    env_gen = container_client.walk_blobs()
    envlist = []
```

```
    for env in env_gen:
        envlist.append(env.name)
    return envlist

def get_trajectory_list(envname, easy_hard = 'Easy'):
    '''
    List all the trajectory folders, which is named as 'P0XX'
    '''
    assert(easy_hard=='Easy' or easy_hard=='Hard')
    traj_gen = container_client.walk_blobs(name_starts_with=envname + '/' +
easy_hard+'/')
    trajlist = []
    for traj in traj_gen:
        trajname = traj.name
        trajname_split = trajname.split('/')
        trajname_split = [tt for tt in trajname_split if len(tt)>0]
        if trajname_split[-1][0] == 'P':
            trajlist.append(trajname)
    return trajlist

def _list_blobs_in_folder(folder_name):
    """
    List all blobs in a virtual folder in an Azure blob container
    """

    files = []
    generator = container_client.list_blobs(name_starts_with=folder_name)
    for blob in generator:
        files.append(blob.name)
    return files

def get_image_list(trajdir, left_right = 'left'):
    assert(left_right == 'left' or left_right == 'right')
    files = _list_blobs_in_folder(trajdir + '/image_' + left_right + '/')
    files = [fn for fn in files if fn.endswith('.png')]
    return files

def get_depth_list(trajdir, left_right = 'left'):
    assert(left_right == 'left' or left_right == 'right')
    files = _list_blobs_in_folder(trajdir + '/depth_' + left_right + '/')
    files = [fn for fn in files if fn.endswith('.npy')]
    return files

def get_flow_list(trajdir, ):
    files = _list_blobs_in_folder(trajdir + '/flow/')
    files = [fn for fn in files if fn.endswith('flow.npy')]
    return files

def get_flow_mask_list(trajdir, ):
    files = _list_blobs_in_folder(trajdir + '/flow/')
    files = [fn for fn in files if fn.endswith('mask.npy')]
    return files

def get_posefile(trajdir, left_right = 'left'):
    assert(left_right == 'left' or left_right == 'right')
```

```
    return trajdir + '/pose_' + left_right + '.txt'

def get_seg_list(trajdir, left_right = 'left'):
    assert(left_right == 'left' or left_right == 'right')
    files = _list_blobs_in_folder(trajdir + '/seg_' + left_right + '/')
    files = [fn for fn in files if fn.endswith('.npy')]
    return files
```

List environments

Python

```
envlist = get_environment_list()
print('Find {} environments..'.format(len(envlist)))
print(envlist)
```

List 'Easy' trajectories in the first environment

Python

```
diff_level = 'Easy'
env_ind = 0
trajlist = get_trajectory_list(envlist[env_ind], easy_hard = diff_level)
print('Find {} trajectories in {}'.format(len(trajlist),
envlist[env_ind]+diff_level))
print(trajlist)
```

List all the data files in one trajectory

Python

```
traj_ind = 1
traj_dir = trajlist[traj_ind]

left_img_list = get_image_list(traj_dir, left_right = 'left')
print('Find {} left images in {}'.format(len(left_img_list), traj_dir))

right_img_list = get_image_list(traj_dir, left_right = 'right')
print('Find {} right images in {}'.format(len(right_img_list), traj_dir))

left_depth_list = get_depth_list(traj_dir, left_right = 'left')
print('Find {} left depth files in {}'.format(len(left_depth_list),
traj_dir))

right_depth_list = get_depth_list(traj_dir, left_right = 'right')
print('Find {} right depth files in {}'.format(len(right_depth_list),
traj_dir))
```

```
left_seg_list = get_seg_list(traj_dir, left_right = 'left')
print('Find {} left segmentation files in {}'.format(len(left_seg_list),
traj_dir))

right_seg_list = get_seg_list(traj_dir, left_right = 'left')
print('Find {} right segmentation files in {}'.format(len(right_seg_list),
traj_dir))

flow_list = get_flow_list(traj_dir)
print('Find {} flow files in {}'.format(len(flow_list), traj_dir))

flow_mask_list = get_flow_mask_list(traj_dir)
print('Find {} flow mask files in {}'.format(len(flow_mask_list), traj_dir))

left_pose_file = get_posefile(traj_dir, left_right = 'left')
print('Left pose file: {}'.format(left_pose_file))

right_pose_file = get_posefile(traj_dir, left_right = 'right')
print('Right pose file: {}'.format(right_pose_file))
```

Data download functions

Python

```python
def read_numpy_file(numpy_file,):
    '''
    return a numpy array given the file path
    '''
    bc = container_client.get_blob_client(blob=numpy_file)
    data = bc.download_blob()
    ee = io.BytesIO(data.content_as_bytes())
    ff = np.load(ee)
    return ff


def read_image_file(image_file,):
    '''
    return a uint8 numpy array given the file path
    '''
    bc = container_client.get_blob_client(blob=image_file)
    data = bc.download_blob()
    ee = io.BytesIO(data.content_as_bytes())
    img=cv2.imdecode(np.asarray(bytearray(ee.read()),dtype=np.uint8),
cv2.IMREAD_COLOR)
    im_rgb = img[:, :, [2, 1, 0]] # BGR2RGB
    return im_rgb
```

Data visualization functions

Python

```python
def depth2vis(depth, maxthresh = 50):
    depthvis = np.clip(depth,0,maxthresh)
    depthvis = depthvis/maxthresh*255
    depthvis = depthvis.astype(np.uint8)
    depthvis = np.tile(depthvis.reshape(depthvis.shape+(1,)), (1,1,3))

    return depthvis

def seg2vis(segnp):
    colors = [(205, 92, 92), (0, 255, 0), (199, 21, 133), (32, 178, 170),
(233, 150, 122), (0, 0, 255), (128, 0, 0), (255, 0, 0), (255, 0, 255), (176,
196, 222), (139, 0, 139), (102, 205, 170), (128, 0, 128), (0, 255, 255), (0,
255, 255), (127, 255, 212), (222, 184, 135), (128, 128, 0), (255, 99, 71),
(0, 128, 0), (218, 165, 32), (100, 149, 237), (30, 144, 255), (255, 0, 255),
(112, 128, 144), (72, 61, 139), (165, 42, 42), (0, 128, 128), (255, 255, 0),
(255, 182, 193), (107, 142, 35), (0, 0, 128), (135, 206, 235), (128, 0, 0),
(0, 0, 255), (160, 82, 45), (0, 128, 128), (128, 128, 0), (25, 25, 112),
(255, 215, 0), (154, 205, 50), (205, 133, 63), (255, 140, 0), (220, 20, 60),
(255, 20, 147), (95, 158, 160), (138, 43, 226), (127, 255, 0), (123, 104,
238), (255, 160, 122), (92, 205, 92),]
    segvis = np.zeros(segnp.shape+(3,), dtype=np.uint8)

    for k in range(256):
        mask = segnp==k
        colorind = k % len(colors)
        if np.sum(mask)>0:
            segvis[mask,:] = colors[colorind]

    return segvis

def _calculate_angle_distance_from_du_dv(du, dv, flagDegree=False):
    a = np.arctan2( dv, du )

    angleShift = np.pi

    if ( True == flagDegree ):
        a = a / np.pi * 180
        angleShift = 180
        # print("Convert angle from radian to degree as demanded by the
input file.")

    d = np.sqrt( du * du + dv * dv )

    return a, d, angleShift

def flow2vis(flownp, maxF=500.0, n=8, mask=None, hueMax=179, angShift=0.0):
    """
    Show a optical flow field as the KITTI dataset does.
    Some parts of this function is the transform of the original MATLAB code
flow_to_color.m.
    """
```

```
    ang, mag, _ = _calculate_angle_distance_from_du_dv( flownp[:, :, 0], flownp[:, :, 1], flagDegree=False )

    # Use Hue, Saturation, Value colour model
    hsv = np.zeros( ( ang.shape[0], ang.shape[1], 3 ) , dtype=np.float32)

    am = ang < 0
    ang[am] = ang[am] + np.pi * 2

    hsv[ :, :, 0 ] = np.remainder( ( ang + angShift ) / (2*np.pi), 1 )
    hsv[ :, :, 1 ] = mag / maxF * n
    hsv[ :, :, 2 ] = (n - hsv[:, :, 1])/n

    hsv[:, :, 0] = np.clip( hsv[:, :, 0], 0, 1 ) * hueMax
    hsv[:, :, 1:3] = np.clip( hsv[:, :, 1:3], 0, 1 ) * 255
    hsv = hsv.astype(np.uint8)

    rgb = cv2.cvtColor(hsv, cv2.COLOR_HSV2RGB)

    if ( mask is not None ):
        mask = mask > 0
        rgb[mask] = np.array([0, 0 ,0], dtype=np.uint8)

    return rgb
```

Download and visualize

Python

```
data_ind = 173 # randomly select one frame (data_ind < TRAJ_LEN)

left_img = read_image_file(left_img_list[data_ind])
right_img = read_image_file(right_img_list[data_ind])

# Visualize the left and right RGB images
plt.figure(figsize=(12, 5))
plt.subplot(121)
plt.imshow(left_img)
plt.title('Left Image')
plt.subplot(122)
plt.imshow(right_img)
plt.title('Right Image')
plt.show()

# Visualize the left and right depth files
left_depth = read_numpy_file(left_depth_list[data_ind])
left_depth_vis = depth2vis(left_depth)

right_depth = read_numpy_file(right_depth_list[data_ind])
right_depth_vis = depth2vis(right_depth)

plt.figure(figsize=(12, 5))
```

```
plt.subplot(121)
plt.imshow(left_depth_vis)
plt.title('Left Depth')
plt.subplot(122)
plt.imshow(right_depth_vis)
plt.title('Right Depth')
plt.show()

# Visualize the left and right segmentation files
left_seg = read_numpy_file(left_seg_list[data_ind])
left_seg_vis = seg2vis(left_seg)

right_seg = read_numpy_file(right_seg_list[data_ind])
right_seg_vis = seg2vis(right_seg)

plt.figure(figsize=(12, 5))
plt.subplot(121)
plt.imshow(left_seg_vis)
plt.title('Left Segmentation')
plt.subplot(122)
plt.imshow(right_seg_vis)
plt.title('Right Segmentation')
plt.show()

# Visualize the flow and mask files

flow = read_numpy_file(flow_list[data_ind])
flow_vis = flow2vis(flow)

flow_mask = read_numpy_file(flow_mask_list[data_ind])
flow_vis_w_mask = flow2vis(flow, mask = flow_mask)

plt.figure(figsize=(12, 5))
plt.subplot(121)
plt.imshow(flow_vis)
plt.title('Optical Flow')
plt.subplot(122)
plt.imshow(flow_vis_w_mask)
plt.title('Optical Flow w/ Mask')
plt.show()
```

Next steps

View the rest of the datasets in the Open Datasets catalog.

Solutions for the facilities and real estate industries

Article • 12/16/2022

IoT, AI, machine learning, and other Azure technologies can help building owners, operators, and occupants save money and live better lives.

Smart buildings can heat and cool themselves, sending energy to the right places at the right times. They can even greet visitors and help them find their way. Predictive maintenance can help building operators know when something is likely to need attention.

Machine learning, Azure IoT Edge, and other technologies can help create safe indoor environments during challenging times. And solar-powered IoT devices combined with Azure services can provide clean, low-cost power, light, and internet service to remote customers.

Watch this short video to get a sense of some of the things smart buildings can do:

https://www.youtube-nocookie.com/embed/CEAyzkuuAQ8

Architectures for facilities and real estate

The following articles provide detailed analysis of architectures created and recommended for the facilities and real-estate industries.

Architecture	Summary	Technology focus
Azure IoT reference architecture	Learn about a recommended architecture for IoT applications that use Azure platform as a service (PaaS) components. Apply this architecture to smart buildings and other facilities scenarios.	IoT
Buy online, pick up in store (BOPIS)	Learn how to use Azure IoT to enhance a solution for stores that implement buy online, pick up in store solutions.	IoT
Cognizant Safe Buildings with IoT and Azure	Deploy Cognizant Safe Buildings, IoT, and Azure services to help protect buildings from COVID-19 outbreaks.	IoT

Architecture	Summary	Technology focus
Connected factory signal pipeline	Learn about the connected factory signal pipeline architecture, which provides a common configuration interface to connect brownfield IoT devices through an OPC Unified Architecture (UA) gateway. Apply this architecture to predictive maintenance scenarios and other scenarios that are related to facilities and real estate.	IoT
COVID-19 safe solutions with IoT Edge	Create an environment that monitors social distance, mask/PPE use, and occupancy requirements by using CCTVs and Azure IoT Edge, Azure Stream Analytics, and Azure Machine Learning.	Containers
Create smart places by using Azure Digital Twins	Use Azure Digital Twins to create models of smart places from IoT device data. View and control products, systems, environments, and experiences.	IoT
Digital image-based modeling on Azure	Learn how to perform image-based modeling on Azure infrastructure as a service (IaaS). You can use this technology to model and measure buildings.	Compute
Facilities management powered by mixed reality and IoT	Improve uptime and operations with mixed reality and IoT. Visualize a virtual replica of your physical space with real-time data in the context of your environment.	Mixed reality
IoT connected light, power, and internet for emerging markets	View an architecture that uses solar-powered IoT devices with Azure services to provide clean, low-cost power, light, and internet service to remote customers.	IoT
IoT device connectivity for healthcare facilities	Learn how to reliably connect building and campus IoT devices to the cloud with improved security and scalability.	Networking
IoT measure and control loops	Learn how an IoT measure and control loop keeps an IoT device within the tolerable range of setpoint configuration. This technology applies to smoke sensors, thermostats, and solar panels.	IoT

Architecture	Summary	Technology focus
IoT monitor and manage loops	Learn about an IoT monitor and manage loop—a supervisory system that continually monitors a physical system that's controlled by a set of networked IoT devices. This technology applies to smart campuses and smart metering.	IoT
UVEN smart and secure disinfection and lighting	Learn about a system that uses IoT and Azure Sphere to provide smart virus disinfection and healthy, human-optimized lighting.	IoT

Solution ideas for facilities and real estate

The following are other ideas that you can use as a starting point for your facilities and real estate solution.

- Azure digital twins builder
- Contactless IoT interfaces with Azure intelligent edge
- Control IoT devices using a voice assistant
- Environment monitoring and supply chain optimization with IoT
- IoT Connected Platform for COVID-19 detection and prevention
- IoT monitor and manage loops
- Low-latency network connections for industry
- Predict length of stay and patient flow

Adopting a DTDL industry ontology

Article • 01/18/2024

Microsoft has partnered with domain experts to create DTDL model sets based on industry standards, to help minimize reinvention and simplify solutions. This article presents the DTDL industry ontologies that are currently available.

List of ontologies

Expand table

Industry	Ontology repository	Description	Learn more
Smart buildings	Digital Twins Definition Language-based RealEstateCore ontology for smart buildings	Microsoft has partnered with RealEstateCore to deliver this open-source DTDL ontology for the real estate industry. RealEstateCore is a consortium of real estate owners, software vendors, and research institutions. This smart buildings ontology provides common ground for modeling smart buildings, using industry standards (like BRICK Schema or W3C Building Topology Ontology) to avoid reinvention. The ontology also comes with best practices for how to consume and properly extend it.	You can read more about the partnership with RealEstateCore and goals for this initiative in the following blog post and embedded video: RealEstateCore, a smart building ontology for digital twins, is now available.
Smart cities	Digital Twins Definition Language (DTDL) ontology for Smart Cities	Microsoft has collaborated with Open Agile Smart Cities (OASC) and Sirus to provide a DTDL-based ontology for smart cities, starting with ETSI CIM NGSI-LD.	You can also read more about the partnerships and approach for smart cities in the following blog post and embedded video: Smart Cities Ontology for Digital Twins.
Energy grids	Digital Twins Definition Language (DTDL)	This ontology was created to help solution providers accelerate development of digital twin solutions for energy use cases like	You can also read more about the partnerships and approach for energy

Industry	Ontology repository	Description	Learn more
	ontology for Energy Grid	monitoring grid assets, outage and impact analysis, simulation, and predictive maintenance. Additionally, the ontology can be used to enable the digital transformation and modernization of the energy grid. It's adapted from the Common Information Model (CIM), a global standard for energy grid assets management, power system operations modeling, and physical energy commodity market.	grids in the following blog post: Energy Grid Ontology for Digital Twins.
Manufacturing	Manufacturing Ontologies	These ontologies were created to help solution providers accelerate development of digital twin solutions for manufacturing use cases like asset condition monitoring, simulation, OEE calculation, and predictive maintenance. Additionally, the ontologies can be used to enable the digital transformation and modernization of factories and plants. They are adapted from OPC UA, ISA95 and the Asset Administration Shell, three global standards widely used in the manufacturing space.	Visit the repository to read more about this ontology and explore a sample solution for ingesting OPC UA data into Azure Digital Twins.

Each ontology is focused on an initial set of models. You can contribute to the ontologies by suggesting extensions or other improvements through the GitHub contribution process in each ontology repository.

Next steps

Learn about extending existing industry-standard ontologies for your specific solution: Extending industry ontologies.

Or, continue on the path for developing models based on ontologies: Full model development path.

Creator for indoor maps

Article • 03/17/2023

This article introduces concepts and tools that apply to Azure Maps Creator. We recommend that you read this article before you begin to use the Azure Maps Creator API and SDK.

You can use Creator to develop applications with map features that are based on indoor map data. This article describes the process of uploading, converting, creating, and using your map data. Typically, the workflow is completed by two different personas with distinct areas of expertise and responsibility:

- Map maker: responsible for curating and preparing the map data.
- Creator map data user: uses customer map data in applications.

The following diagram illustrates the entire workflow.

Create Azure Maps Creator

To use Creator services, an Azure Maps Creator resource must be created and associated to an Azure Maps account with the Gen 2 pricing tier. For information about how to create an Azure Maps Creator resource in Azure, see Manage Azure Maps Creator.

> **Tip**
>
> For pricing information see the *Creator* section in **Azure Maps pricing**.

Creator authentication

Creator inherits Azure Maps Access Control (IAM) settings. All API calls for data access must be sent with authentication and authorization rules.

Creator usage data is incorporated in your Azure Maps usage charts and activity log. For more information, see Manage authentication in Azure Maps.

> **Important**
>
> We recommend using:
>
> - Azure Active Directory (Azure AD) in all solutions that are built with an Azure Maps account using Creator services. For more information about Azure AD, see **Azure AD authentication**.
> - Role-based access control settings. Using these settings, map makers can act as the Azure Maps Data Contributor role, and Creator map data users can act as the Azure Maps Data Reader role. For more information, see **Authorization with role-based access control**.

Creator data item types

Creator services create, store, and use various data types that are defined and discussed in the following sections. A creator data item can be of the following types:

- Converted data
- Dataset
- Tileset
- style
- Map configuration
- Feature stateset
- Routeset

Upload a drawing package

Creator collects indoor map data by converting an uploaded drawing package. The drawing package represents a constructed or remodeled facility. For information about drawing package requirements, see Drawing package requirements.

Use Data Upload to upload a drawing package. After the Drawing packing is uploaded, the Data Upload API returns a user data identifier (`udid`). The `udid` can then be used to convert the uploaded package into indoor map data.

Convert a drawing package

The Conversion service converts an uploaded drawing package into indoor map data. The Conversion service also validates the package. Validation issues are classified into two types:

- Errors: If any errors are detected, the conversion process fails. When an error occurs, the Conversion service provides a link to the Azure Maps Drawing Error Visualizer stand-alone web application. You can use the Drawing Error Visualizer to inspect Drawing package warnings and errors that occurred during the conversion process. After you fix the errors, you can attempt to upload and convert the package.
- Warnings: If any warnings are detected, the conversion succeeds. However, we recommend that you review and resolve all warnings. A warning means that part of the conversion was ignored or automatically fixed. Failing to resolve the warnings could result in errors in later processes. For more information, see Drawing package warnings and errors.

Create indoor map data

Azure Maps Creator provides the following services that support map creation:

- Dataset service.
- Tileset service. Use the Tileset service to create a vector-based representation of a dataset. Applications can use a tileset to present a visual tile-based view of the dataset.
- Custom styling service. Use the style service or visual style editor to customize the visual elements of an indoor map.
- Feature State service. Use the Feature State service to support dynamic map styling. Applications can use dynamic map styling to reflect real-time events on spaces provided by the IoT system.
- Wayfinding service. Use the wayfinding API to generate a path between two points within a facility. Use the routeset API to create the data that the wayfinding service needs to generate paths.

Datasets

A dataset is a collection of indoor map features. The indoor map features represent facilities that are defined in a converted drawing package. After you create a dataset with the Dataset service, you can create any number of tilesets or feature statesets.

At any time, developers can use the Dataset service to add or remove facilities to an existing dataset. For more information about how to update an existing dataset using the API, see the append options in Dataset service. For an example of how to update a dataset, see Data maintenance.

Tilesets

A tileset is a collection of vector data that represents a set of uniform grid tiles. Developers can use the Tileset service to create tilesets from a dataset.

To reflect different content stages, you can create multiple tilesets from the same dataset. For example, you can make one tileset with furniture and equipment, and another tileset without furniture and equipment. You might choose to generate one tileset with the most recent data updates, and another tileset without the most recent data updates.

In addition to the vector data, the tileset provides metadata for map rendering optimization. For example, tileset metadata contains a minimum and maximum zoom level for the tileset. The metadata also provides a bounding box that defines the geographic extent of the tileset. An application can use a bounding box to programmatically set the correct center point. For more information about tileset metadata, see Tileset List.

After a tileset is created, it's retrieved using the Render service.

If a tileset becomes outdated and is no longer useful, you can delete the tileset. For information about how to delete tilesets, see Data maintenance.

> **Note**
>
> A tileset is independent of the dataset from which it was created. If you create tilesets from a dataset, and then subsequently update that dataset, the tilesets isn't updated.
>
> To reflect changes in a dataset, you must create new tilesets. Similarly, if you delete a tileset, the dataset isn't affected.

Custom styling (preview)

A style defines the visual appearance of a map. It defines what data to draw, the order to draw it in, and how to style the data when drawing it. Azure Maps Creator styles support the MapLibre standard for style layers and sprites.

When you convert a drawing package after uploading it to your Azure Maps account, default styles are applied to the elements of your map. The custom styling service enables you to customize the visual appearance of your map. You can do this by manually editing the style JSON and importing it into your Azure Maps account using the Style - Create HTTP request, however the recommended approach is to use the visual style editor. For more information, see Create custom styles for indoor maps.

Example layer in the style.json file:

JSON

```
{
    "id": "indoor_unit_gym_label",
    "type": "symbol",
    "filter": ["all", ["has","floor0"], ["any", ["==", "categoryName",
"room.gym"]]],
    "layout": {
        "visibility": "none",
        "icon-image": "gym",
        "icon-size": {"stops": [[17.5, 0.7], [21, 1.1]]},
        "symbol-avoid-edges": true,
        "symbol-placement": "point",
        "text-anchor": "top",
        "text-field": "{name}",
        "text-font": ["SegoeFrutigerHelveticaMYingHei-Medium"],
        "text-keep-upright": true,
        "text-letter-spacing": 0.1,
        "text-offset": [0, 1.05],
        "text-size": {"stops": [[18, 5], [18.5, 6.5], [19, 8], [19.5, 9.5],
[20, 11]]}
    },
    "metadata": {"microsoft.maps:layerGroup": "labels_indoor"},
    "minzoom": 17.5,
    "paint": {
        "text-color": "rgba(0, 0, 0, 1)",
        "text-halo-blur": 0.5,
        "text-halo-color": "rgba(255, 255, 255, 1)",
        "text-halo-width": 1,
        "text-opacity": ["step", ["zoom"], 0, 18, 1]
    },
    "source-layer": "Indoor unit"
},
```

Layer Properties	Description
id	The name of the layer

Layer Properties	Description
type	The rendering type for this layer. Some of the more common types include: **fill**: A filled polygon with an optional stroked border. **Line**: A stroked line. **Symbol**: An icon or a text label. **fill-extrusion**: An extruded (3D) polygon.
filter	Only features that match the filter criteria are displayed.
layout	Layout properties for the layer.
minzoom	A number between 0 and 24 that represents the minimum zoom level for the layer. At zoom levels less than the minzoom, the layer is hidden.
paint	Default paint properties for this layer.
source-layer	A source supplies the data, from a vector tile source, displayed on a map. Required for vector tile sources; prohibited for all other source types, including GeoJSON sources.

Map configuration

The map configuration is an array of configurations. Each configuration consists of a basemap and one or more layers, each layer consisting of a style + tileset tuple.

The map configuration is used when you Instantiate the Indoor Manager of a Map object when developing applications in Azure Maps. It's referenced using the `mapConfigurationId` or `alias`. Map configurations are immutable. When making changes to an existing map configuration, a new map configuration is created, resulting in a different `mapConfingurationId`. Anytime you create a map configuration using an alias already used by an existing map configuration, it points to the new map configuration.

The following JSON is an example of a default map configuration. See the following table for a description of each element of the file:

JSON

```
{
    "version": 1.0,
    "description": "This is the default Azure Maps map configuration for
facility ontology tilesets.",
    "defaultConfiguration": "indoor_light",
    "configurations": [
```

```
        {
            "name": "indoor_light",
            "displayName": "Indoor light",
            "description": "A base style for Azure Maps.",
            "thumbnail": "indoor_2022-01-01.png",
            "baseMap": "microsoft_light",
            "layers": [
                {
                    "tilesetId": "fa37d225-924e-3f32-8441-6128d9e5519a",
                    "styleId": "microsoft-maps:indoor_2022-01-01"
                }
            ]
        },
        {
            "name": "indoor_dark",
            "displayName": "Indoor dark",
            "description": "A base style for Azure Maps.",
            "thumbnail": "indoor_dark_2022-01-01.png",
            "baseMap": "microsoft_dark",
            "layers": [
                {
                    "tilesetId": "fa37d225-924e-3f32-8441-6128d9e5519a",
                    "styleId": "microsoft-maps:indoor_dark_2022-01-01"
                }
            ]
        }
    ]
}
```

Style Object Properties	Description
Name	The name of the style.
displayName	The display name of the style.
description	The user defined description of the style.
thumbnail	Use to specify the thumbnail used in the style picker for this style. For more information, see the style picker control.
baseMap	Use to Set the base map style.
layers	The layers array consists of one or more *tileset* + *Style* tuples, each being a layer of the map. This enables multiple buildings on a map, each building represented in its own tileset.

Additional information

- For more information how to modify styles using the style editor, see Create custom styles for indoor maps.

- For more information on style Rest API, see style in the Maps Creator Rest API reference.
- For more information on the map configuration Rest API, see Creator - map configuration Rest API.

Feature statesets

Feature statesets are collections of dynamic properties (*states*) that are assigned to dataset features, such as rooms or equipment. An example of a *state* can be temperature or occupancy. Each *state* is a key/value pair that contains the name of the property, the value, and the timestamp of the last update.

You can use the Feature State service to create and manage a feature stateset for a dataset. The stateset is defined by one or more *states*. Each feature, such as a room, can have one *state* attached to it.

The value of each *state* in a stateset is updated or retrieved by IoT devices or other applications. For example, using the Feature State Update API, devices measuring space occupancy can systematically post the state change of a room.

An application can use a feature stateset to dynamically render features in a facility according to their current state and respective map style. For more information about using feature statesets to style features in a rendering map, see Indoor Maps module.

> **Note**
>
> Like tilesets, changing a dataset doesn't affect the existing feature stateset, and deleting a feature stateset doesn't affect the dataset to which it's attached.

Wayfinding (preview)

The Wayfinding service enables you to provide your customers with the shortest path between two points within a facility. Once you've imported your indoor map data and created your dataset, you can use that to create a routeset. The routeset provides the data required to generate paths between two points. The wayfinding service takes into account things such as the minimum width of openings and may optionally exclude elevators or stairs when navigating between levels as a result.

Creator wayfinding is powered by Havok.

Wayfinding paths

When a wayfinding path is successfully generated, it finds the shortest path between two points in the specified facility. Each floor in the journey is represented as a separate leg, as are any stairs or elevators used to move between floors.

For example, the first leg of the path might be from the origin to the elevator on that floor. The next leg is the elevator, and then the final leg is the path from the elevator to the destination. The estimated travel time is also calculated and returned in the HTTP response JSON.

Structure

For wayfinding to work, the facility data must contain a structure. The wayfinding service calculates the shortest path between two selected points in a facility. The service creates the path by navigating around structures, such as walls and any other impermeable structures.

Vertical penetration

If the selected origin and destination are on different floors, the wayfinding service determines what verticalPenetration objects such as stairs or elevators, are available as possible pathways for navigating vertically between levels. By default, the option that results in the shortest path is used.

The Wayfinding service includes stairs or elevators in a path based on the value of the vertical penetration's `direction` property. For more information on the direction property, see verticalPenetration in the Facility Ontology article. See the `avoidFeatures` and `minWidth` properties in the wayfinding API documentation to learn about other factors that can affect the path selection between floor levels.

For more information, see the Indoor maps wayfinding service how-to article.

Using indoor maps

Render V2-Get Map Tile API

The Azure Maps Render V2-Get Map Tile API has been extended to support Creator tilesets.

Applications can use the Render V2-Get Map Tile API to request tilesets. The tilesets can then be integrated into a map control or SDK. For an example of a map control that uses the Render V2 service, see Indoor Maps Module.

Web Feature service API

You can use the Web Feature service (WFS) to query datasets. WFS follows the Open Geospatial Consortium API Features. You can use the WFS API to query features within the dataset itself. For example, you can use WFS to find all mid-size meeting rooms of a specific facility and floor level.

Alias API

Creator services such as Conversion, Dataset, Tileset and Feature State return an identifier for each resource that's created from the APIs. The Alias API allows you to assign an alias to reference a resource identifier.

Indoor Maps module

The Azure Maps Web SDK includes the Indoor Maps module. This module offers extended functionalities to the Azure Maps *Map Control* library. The Indoor Maps module renders indoor maps created in Creator. It integrates widgets such as *floor picker* that help users to visualize the different floors.

You can use the Indoor Maps module to create web applications that integrate indoor map data with other Azure Maps services. The most common application setups include adding knowledge from other maps - such as road, imagery, weather, and transit - to indoor maps.

The Indoor Maps module also supports dynamic map styling. For a step-by-step walkthrough to implement feature stateset dynamic styling in an application, see Use the Indoor Map module.

Azure Maps integration

As you begin to develop solutions for indoor maps, you can discover ways to integrate existing Azure Maps capabilities. For example, you can implement asset tracking or safety scenarios by using the Geofence service with Creator indoor maps. For example, you can use the Geofence API to determine whether a worker enters or leaves specific indoor areas. For more information about how to connect Azure Maps with IoT telemetry, see Tutorial: Implement IoT spatial analytics by using Azure Maps.

Data maintenance

You can use the Azure Maps Creator List, Update, and Delete API to list, update, and delete your datasets, tilesets, and feature statesets.

> **Note**
>
> When you review a list of items to determine whether to delete them, consider the impact of that deletion on all dependent API or applications. For example, if you delete a tileset that's being used by an application by means of the **Render V2-Get Map Tile API**, the application fails to render that tileset.

Example: Updating a dataset

The following example shows how to update a dataset, create a new tileset, and delete an old tileset:

1. Follow steps in the Upload a drawing package and Convert a drawing package sections to upload and convert the new drawing package.
2. Use Dataset Create to append the converted data to the existing dataset.
3. Use Tileset Create to generate a new tileset out of the updated dataset.
4. Save the new **tilesetId** for the next step.
5. To enable the visualization of the updated campus dataset, update the tileset identifier in your application. If the old tileset is no longer used, you can delete it.

Next steps

Tutorial: Creating a Creator indoor map

Tutorial: Use Azure Maps Creator to create indoor maps

Article • 10/12/2023

This tutorial describes how to create indoor maps for use in Microsoft Azure Maps. This tutorial demonstrates how to:

- ✓ Upload your drawing package for indoor maps.
- ✓ Convert your drawing package into map data.
- ✓ Create a dataset from your map data.
- ✓ Create a tileset from the data in your dataset.
- ✓ Get the default map configuration ID from your tileset.

You can also create a dataset from a GeoJSON package. For more information, see Create a dataset using a GeoJSON package (preview).

Prerequisites

- An Azure Maps account
- A subscription key
- A Creator resource
- An Azure storage account
- The sample drawing package downloaded

This tutorial uses the Postman application, but you can use a different API development environment.

> **Important**
>
> - This article uses the `us.atlas.microsoft.com` geographical URL. If your Creator service wasn't created in the United States, you must use a different geographical URL. For more information, see **Access to Creator services**.
> - In the URL examples, replace `{Your-Azure-Maps-Subscription-key}` with your Azure Maps subscription key.

Upload a drawing package

Follow the steps outlined in the How to create data registry article to upload the GeoJSON package into your Azure storage account then register it in your Azure Maps account.

> **Important**
>
> Make sure to make a note of the unique identifier (`udid`) value, you will need it. The `udid` is is how you reference the GeoJSON package you uploaded into your Azure storage account from your source code and HTTP requests.

Convert a drawing package

Now that the drawing package is uploaded, you use the `udid` value for the uploaded package to convert the package into map data. The Conversion API uses a long-running transaction that implements the pattern defined in the Creator Long-Running Operation article.

To convert a drawing package:

1. In the Postman app, select **New**.
2. In the **Create New** window, select **HTTP Request**.
3. For **Request name**, enter a name for the request, such as **POST Convert Drawing Package**.
4. Select the **POST** HTTP method.
5. Enter the following URL to the Conversion service. Replace `{Your-Azure-Maps-Subscription-key}` with your Azure Maps subscription key. Replace `udid` with the `udid` value of the uploaded package.

   ```HTTP
   https://us.atlas.microsoft.com/conversions?subscription-key={Your-Azure-Maps-Subscription-key}&api-version=2023-03-01-preview&udid={udid}&inputType=DWG&dwgPackageVersion=2.0
   ```

6. Select **Send**.
7. In the response window, select the **Headers** tab.

8. Copy the value of the **Operation-Location** key. It contains the status URL that you use to check the status of the conversion.

Body Cookies Headers (9) Test Results Status: 202 Accepted Time: 14

KEY	VALUE
Content-Type	application/json
x-ms-azuremaps-region	West US 2
X-Content-Type-Options	nosniff
Strict-Transport-Security	max-age=31536000; includeSubDomains
Operation-Location	https://us.atlas.microsoft.com/conversions/operations/c1414013-4677-4d6f-bce4-eb65f14c237f?api-version=2.0
X-Cache	CONFIG_NOCACHE
X-MSEdge-Ref	Ref A: B5090DE2CEB84A2CA9EEE2EC07537853 Ref B: WSTEDGE1116 Ref C: 2021-05-19T18:24:27Z
Date	Wed, 19 May 2021 18:24:28 GMT
Content-Length	0

Check the status of the drawing package conversion

After the conversion operation finishes, it returns a `conversionId` value. You can access the `conversionId` value by checking the status of the drawing package's conversion process. You can then use the `conversionId` value to access the converted data.

To check the status of the conversion process and retrieve the `conversionId` value:

1. In the Postman app, select **New**.
2. In the **Create New** window, select **HTTP Request**.
3. For **Request name**, enter a name for the request, such as **GET Conversion Status**.
4. Select the **GET** HTTP method.
5. Enter the status URL that you copied in the Convert a drawing package section. The request should look like the following URL:

```HTTP
https://us.atlas.microsoft.com/conversions/operations/{operationId}?
api-version=2.0&subscription-key={Your-Azure-Maps-Subscription-key}
```

6. Select **Send**.
7. In the response window, select the **Headers** tab.
8. Copy the value of the **Resource-Location** key, which is the resource location URL. The resource location URL contains the unique identifier `conversionId`, which other APIs use to access the converted map data.

Cookies Headers (9) Test Results Status: 200 OK Time: 656 ms Size: 1.07 KB Save Response

KEY	VALUE
Content-Length	593
Content-Type	application/json; charset=utf-8
x-ms-azuremaps-regi...	West US 2
X-Content-Type-Opti...	nosniff
Strict-Transport-Secu...	max-age=31536000; includeSubDomains
Resource-Location	https://us.atlas.microsoft.com/conversions/5dd13c86-5dbd-b058-1024-e6f392f40baa?api-version=2.0
X-Cache	CONFIG_NOCACHE
X-MSEdge-Ref	Ref A: 26478BDDA9534CB9ABE456DB40020736 Ref B: WSTEDGE1012 Ref C: 2021-05-20T17:15:50Z
Date	Thu, 20 May 2021 17:15:50 GMT

The sample drawing package should be converted without errors or warnings. But if you receive errors or warnings from your own drawing package, the JSON response includes a link to the Drawing Error Visualizer. You can use the Drawing Error Visualizer to inspect the details of errors and warnings. To get recommendations for resolving conversion errors and warnings, see Drawing conversion errors and warnings.

The following JSON fragment displays a sample conversion warning:

JSON

```
{
    "operationId": "{operationId}",
    "created": "2021-05-19T18:24:28.7922905+00:00",
    "status": "Succeeded",
     "warning": {
        "code": "dwgConversionProblem",
        "details": [
            {
                "code": "warning",
                "details": [
                    {
                        "code": "manifestWarning",
                        "message": "Ignoring unexpected JSON property:
unitProperties[0].nonWheelchairAccessible with value False"
                    }
                ]
            }
        ]
    },
    "properties": {
        "diagnosticPackageLocation":
"https://atlas.microsoft.com/mapData/ce61c3c1-faa8-75b7-349f-d863f6523748?
api-version=1.0"
    }
}
```

Create a dataset

A dataset is a collection of map features, such as buildings, levels, and rooms. To create a dataset, use the Dataset Create API. The Dataset Create API takes the `conversionId` value for the converted drawing package and returns a `datasetId` value for the created dataset.

To create a dataset:

1. In the Postman app, select **New**.
2. In the **Create New** window, select **HTTP Request**.
3. For **Request name**, enter a name for the request, such as **POST Dataset Create**.
4. Select the **POST** HTTP method.
5. Enter the following URL to the Dataset service. Replace `{conversionId}` with the `conversionId` value that you obtained in Check the status of the drawing package conversion.

   HTTP

   ```
   https://us.atlas.microsoft.com/datasets?api-version=2023-03-01-preview&conversionId={conversionId}&subscription-key={Your-Azure-Maps-Subscription-key}
   ```

6. Select **Send**.
7. In the response window, select the **Headers** tab.
8. Copy the value of the **Operation-Location** key. It contains the status URL that you use to check the status of the dataset.

Cookies Headers (9) Test Results Status: 202

KEY	VALUE
Content-Type	application/json
x-ms-azuremaps-region	West US 2
X-Content-Type-Options	nosniff
Strict-Transport-Security	max-age=31536000; includeSubDomains
Operation-Location	https://us.atlas.microsoft.com/datasets/operations/92267b58-1078-4212-8fe2-026bfb3e9bd8?api-version=2.0
X-Cache	CONFIG_NOCACHE
X-MSEdge-Ref	Ref A: 9E9CBE907A544F8D890B44497656B2F8 Ref B: WSTEDGE0920 Ref C: 2021-05-19T19:19:21Z
Date	Wed, 19 May 2021 19:19:21 GMT
Content-Length	0

Check the dataset creation status

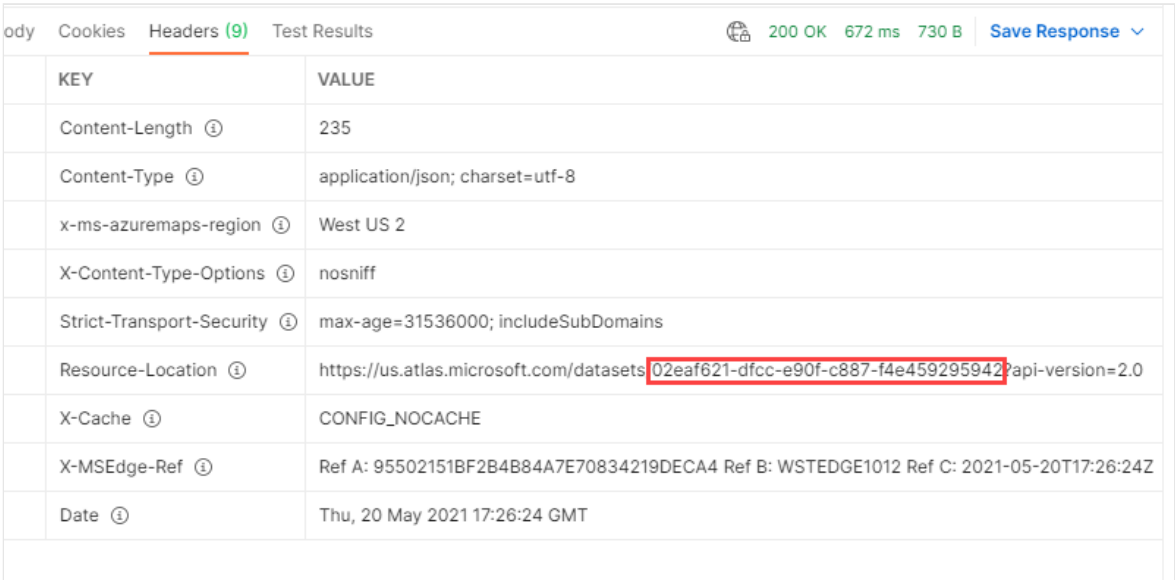
To check the status of the dataset creation process and retrieve the `datasetId` value:

1. In the Postman app, select **New**.
2. In the **Create New** window, select **HTTP Request**.
3. For **Request name**, enter a name for the request, such as **GET Dataset Status**.
4. Select the **GET** HTTP method.
5. Enter the status URL that you copied in the Create a dataset section. The request should look like the following URL:

   HTTP

   ```
   https://us.atlas.microsoft.com/datasets/operations/{operationId}?api-version=2023-03-01-preview&subscription-key={Your-Azure-Maps-Subscription-key}
   ```

6. Select **Send**.
7. In the response window, select the **Headers** tab. The value of the **Resource-Location** key is the resource location URL. The resource location URL contains the unique identifier (`datasetId`) of the dataset.
8. Save the `datasetId` value, because you'll use it in the next tutorial.



Create a tileset

A tileset is a set of vector tiles that render on the map. Tilesets are created from existing datasets. However, a tileset is independent from the dataset that it comes from. If the

dataset is deleted, the tileset continues to exist.

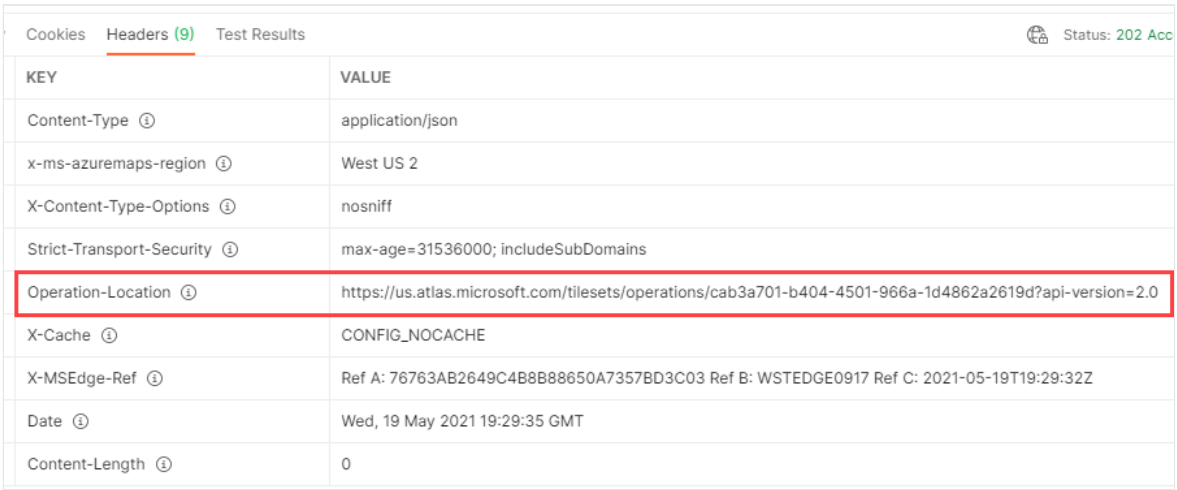
To create a tileset:

1. In the Postman app, select **New**.
2. In the **Create New** window, select **HTTP Request**.
3. For **Request name**, enter a name for the request, such as **POST Tileset Create**.
4. Select the **POST** HTTP method.
5. Enter the following URL to the Tileset service. Replace `{datasetId}` with the `datasetId` value that you obtained in the Check the dataset creation status section.

   HTTP

   ```
   https://us.atlas.microsoft.com/tilesets?api-version=2023-03-01-preview&datasetID={datasetId}&subscription-key={Your-Azure-Maps-Primary-Subscription-key}
   ```

6. Select **Send**.
7. In the response window, select the **Headers** tab.
8. Copy the value of the **Operation-Location** key. It contains the status URL, which you use to check the status of the tileset.



KEY	VALUE
Content-Type	application/json
x-ms-azuremaps-region	West US 2
X-Content-Type-Options	nosniff
Strict-Transport-Security	max-age=31536000; includeSubDomains
Operation-Location	https://us.atlas.microsoft.com/tilesets/operations/cab3a701-b404-4501-966a-1d4862a2619d?api-version=2.0
X-Cache	CONFIG_NOCACHE
X-MSEdge-Ref	Ref A: 76763AB2649C4B8B88650A7357BD3C03 Ref B: WSTEDGE0917 Ref C: 2021-05-19T19:29:32Z
Date	Wed, 19 May 2021 19:29:35 GMT
Content-Length	0

Check the status of tileset creation

To check the status of the tileset creation process and retrieve the `tilesetId` value:

1. In the Postman app, select **New**.

2. In the **Create New** window, select **HTTP Request**.

3. For **Request name**, enter a name for the request, such as **GET Tileset Status**.

4. Select the **GET** HTTP method.

5. Enter the status URL that you copied in the Create a tileset section. The request should look like the following URL:

```
HTTP

https://us.atlas.microsoft.com/tilesets/operations/{operationId}?api-version=2023-03-01-preview&subscription-key={Your-Azure-Maps-Subscription-key}
```

6. Select **Send**.

7. In the response window, select the **Headers** tab. The value of the **Resource-Location** key is the resource location URL. The resource location URL contains the unique identifier (`tilesetId`) of the dataset.

Cookies Headers (9) Test Results Status: 200 OK Time: 410 ms Size: 730 B Save Response

KEY	VALUE
Content-Length	235
Content-Type	application/json; charset=utf-8
x-ms-azuremaps-region	West US 2
X-Content-Type-Options	nosniff
Strict-Transport-Security	max-age=31536000; includeSubDomains
Resource-Location	https://us.atlas.microsoft.com/tilesets/d8eed547-dcc7-bf52-5b75-082bcd7c5ac1?api-version=2.0
X-Cache	CONFIG_NOCACHE
X-MSEdge-Ref	Ref A: D4117D7DDCEA4490975B653F92CE235F Ref B: WSTEDGE0621 Ref C: 2021-05-20T17:29:38Z
Date	Thu, 20 May 2021 17:29:38 GMT

Get the map configuration (preview)

After you create a tileset, you can get the `mapConfigurationId` value by using the tileset get HTTP request:

1. In the Postman app, select **New**.

2. In the **Create New** window, select **HTTP Request**.

3. For **Request name**, enter a name for the request, such as **GET mapConfigurationId from Tileset**.

4. Select the **GET** HTTP method.

5. Enter the following URL to the Tileset service. Pass in the tileset ID that you obtained in the previous step.

```http
https://us.atlas.microsoft.com/tilesets/{tilesetId}?api-version=2023-03-01-preview&subscription-key={Your-Azure-Maps-Subscription-key}
```

6. Select **Send**.

7. The tileset JSON appears in the body of the response. Scroll down to see the `mapConfigurationId` value:

```json
"defaultMapConfigurationId": "5906cd57-2dba-389b-3313-ce6b549d4396"
```

For more information, see Map configuration in the article about indoor map concepts.

Next steps

Use the Azure Maps Indoor Maps module with custom styles

Use the Azure Maps Indoor Maps module with custom styles (preview)

Article • 06/30/2023

The Azure Maps Web SDK includes an Indoor Maps module, enabling you to render indoor maps created in Azure Maps Creator services.

When you create an indoor map using Azure Maps Creator, default styles are applied. Azure Maps Creator now also supports customizing the styles of the different elements of your indoor maps using the Style Rest API, or the visual style editor.

Prerequisites

- Azure Maps account
- Azure Maps Creator resource
- Subscription key
- A map configuration alias or ID. For more information, see map configuration API.

> **Tip**
>
> If you have never used Azure Maps Creator to create an indoor map, you might find the **Use Creator to create indoor maps** tutorial helpful.

The map configuration `alias` (or `mapConfigurationId`) is required to render indoor maps with custom styles via the Azure Maps Indoor Maps module.

Embed the Indoor Maps module

You can install and embed the *Azure Maps Indoor* module in one of two ways.

To use the globally hosted Azure Content Delivery Network version of the *Azure Maps Indoor* module, reference the following `script` and `stylesheet` references in the `<head>` element of the HTML file:

HTML

```
<link rel="stylesheet"
href="https://atlas.microsoft.com/sdk/javascript/indoor/0.2/atlas-
indoor.min.css" type="text/css"/>
```

```
<script src="https://atlas.microsoft.com/sdk/javascript/indoor/0.2/atlas-indoor.min.js"></script>
```

Or, you can download the *Azure Maps Indoor* module. The *Azure Maps Indoor* module contains a client library for accessing Azure Maps services. The following steps demonstrate how to install and load the *Indoor* module into your web application.

1. Install the latest azure-maps-indoor package.

   PowerShell

   ```
   >npm install azure-maps-indoor
   ```

2. Import the *Azure Maps Indoor* module JavaScript in a source file:

   JavaScript

   ```
   import * as indoor from "azure-maps-indoor";
   ```

   You would also need to embed the CSS Style Sheet for various controls to display correctly. If you're using a JavaScript bundler to bundle the dependencies and package your code, refer to your bundler's documentation on how it's done. For Webpack, it's commonly done via a combination of `style-loader` and `css-loader` with documentation available at style-loader.

   To begin, install style-loader and css-loader:

   PowerShell

   ```
   npm install --save-dev style-loader css-loader
   ```

   Inside your source file, import atlas-indoor.min.css:

   JavaScript

   ```
   import "azure-maps-indoor/dist/atlas-indoor.min.css";
   ```

   Then add loaders to the module rules portion of the Webpack config:

   JavaScript

   ```
   module.exports = {
     module: {
       rules: [
   ```

```
        {
          test: /\.css$/i,
          use: ["style-loader", "css-loader"]
        }
      ]
    }
  };
```

Set the domain and instantiate the Map object

Set the map domain with a prefix matching the location of your Creator resource, `US` or `EU`, for example:

```
atlas.setDomain('us.atlas.microsoft.com');
```

For more information, see Azure Maps service geographic scope.

Next, instantiate a *Map object* with the map configuration object set to the `alias` or `mapConfigurationId` property of your map configuration, then set your `styleAPIVersion` to `2022-09-01-preview`.

The *Map object* will be used in the next step to instantiate the *Indoor Manager* object. The following code shows you how to instantiate the *Map object* with `mapConfiguration`, `styleAPIVersion` and map domain set:

JavaScript

```
const subscriptionKey = "<Your Azure Maps Subscription Key>";
const region = "<Your Creator resource region: us or eu>"
const mapConfiguration = "<map configuration alias or ID>"
atlas.setDomain(`${region}.atlas.microsoft.com`);

const map = new atlas.Map("map-id", {
  //use your facility's location
  center: [-122.13315, 47.63637],
  //or, you can use bounds: [# west, # south, # east, # north] and replace #
with your Map bounds
  authOptions: {
      authType: 'subscriptionKey',
      subscriptionKey: subscriptionKey
  },
  zoom: 19,

  mapConfiguration: mapConfiguration,
  styleAPIVersion: '2022-09-01-preview'
});
```

Instantiate the Indoor Manager

To load the indoor map style of the tiles, you must instantiate the *Indoor Manager*. Instantiate the *Indoor Manager* by providing the *Map object*. If you wish to support dynamic map styling, you must pass the `statesetId`. The `statesetId` variable name is case-sensitive. Your code should look like the following JavaScript code snippet:

javascriptf

```
const statesetId = "<statesetId>";

const indoorManager = new atlas.indoor.IndoorManager(map, {
  statesetId: statesetId // Optional
});
```

To enable polling of state data you provide, you must provide the `statesetId` and call `indoorManager.setDynamicStyling(true)`. Polling state data lets you dynamically update the state of dynamic properties or *states*. For example, a feature such as room can have a dynamic property (*state*) called `occupancy`. Your application may wish to poll for any *state* changes to reflect the change inside the visual map. The following code shows you how to enable state polling:

JavaScript

```
const statesetId = "<statesetId>";

const indoorManager = new atlas.indoor.IndoorManager(map, {
  statesetId: statesetId // Optional
});

if (statesetId.length > 0) {
    indoorManager.setDynamicStyling(true);
}
```

Indoor level picker control

The *Indoor Level Picker* control allows you to change the level of the rendered map. You can optionally initialize the *Indoor Level Picker* control via the *Indoor Manager*. Here's the code to initialize the level control picker:

JavaScript

```
const levelControl = new atlas.control.LevelControl({ position: "top-right"
});
```

```
indoorManager.setOptions({ levelControl });
```

Indoor events

The *Azure Maps Indoor* module supports *Map object* events. The *Map object* event listeners are invoked when a level or facility has changed. If you want to run code when a level or a facility have changed, place your code inside the event listener. The code below shows how event listeners can be added to the *Map object*.

JavaScript

```
map.events.add("levelchanged", indoorManager, (eventData) => {

  //code that you want to run after a level has been changed
  console.log("The level has changed: ", eventData);
});

map.events.add("facilitychanged", indoorManager, (eventData) => {

  //code that you want to run after a facility has been changed
  console.log("The facility has changed: ", eventData);
});
```

The `eventData` variable holds information about the level or facility that invoked the `levelchanged` or `facilitychanged` event, respectively. When a level changes, the `eventData` object contains the `facilityId`, the new `levelNumber`, and other metadata. When a facility changes, the `eventData` object contains the new `facilityId`, the new `levelNumber`, and other metadata.

Example: custom styling: consume map configuration in WebSDK (preview)

When you create an indoor map using Azure Maps Creator, default styles are applied. Azure Maps Creator now also supports customizing your indoor styles. For more information, see Create custom styles for indoor maps. Creator also offers a visual style editor.

1. Follow the Create custom styles for indoor maps how-to article to create your custom styles. Make a note of the map configuration alias after saving your changes.

2. Use the Azure Content Delivery Network option to install the *Azure Maps Indoor* module.

3. Create a new HTML file

4. In the HTML header, reference the *Azure Maps Indoor* module JavaScript and style sheet.

5. Set the map domain with a prefix matching a location of your Creator resource: `atlas.setDomain('us.atlas.microsoft.com');` if your Creator resource has been created in US region, or `atlas.setDomain('eu.atlas.microsoft.com');` if your Creator resource has been created in EU region.

6. Initialize a *Map object*. The *Map object* supports the following options:

   - `Subscription key` is your Azure Maps subscription key.
   - `center` defines a latitude and longitude for your indoor map center location. Provide a value for `center` if you don't want to provide a value for `bounds`. Format should appear as `center`: [-122.13315, 47.63637].
   - `bounds` is the smallest rectangular shape that encloses the tileset map data. Set a value for `bounds` if you don't want to set a value for `center`. You can find your map bounds by calling the Tileset List API. The Tileset List API returns the `bbox`, which you can parse and assign to `bounds`. Format should appear as `bounds`: [# west, # south, # east, # north].
   - `mapConfiguration` the ID or alias of the map configuration that defines the custom styles you want to display on the map, use the map configuration ID or alias from step 1.
   - `style` allows you to set the initial style from your map configuration that is displayed. If not set, the style matching map configuration's default configuration is used.
   - `zoom` allows you to specify the min and max zoom levels for your map.
   - `styleAPIVersion`: pass **'2022-09-01-preview'** (which is required while Custom Styling is in public preview)

7. Next, create the *Indoor Manager* module with *Indoor Level Picker* control instantiated as part of *Indoor Manager* options, optionally set the `statesetId` option.

8. Add *Map object* event listeners.

Tip

The map configuration is referenced using the `mapConfigurationId` or `alias` . Each time you edit or change a map configuration, its ID changes but its alias remains the same. It is recommended to reference the map configuration by its alias in your applications. For more information, See **map configuration** in the concepts article.

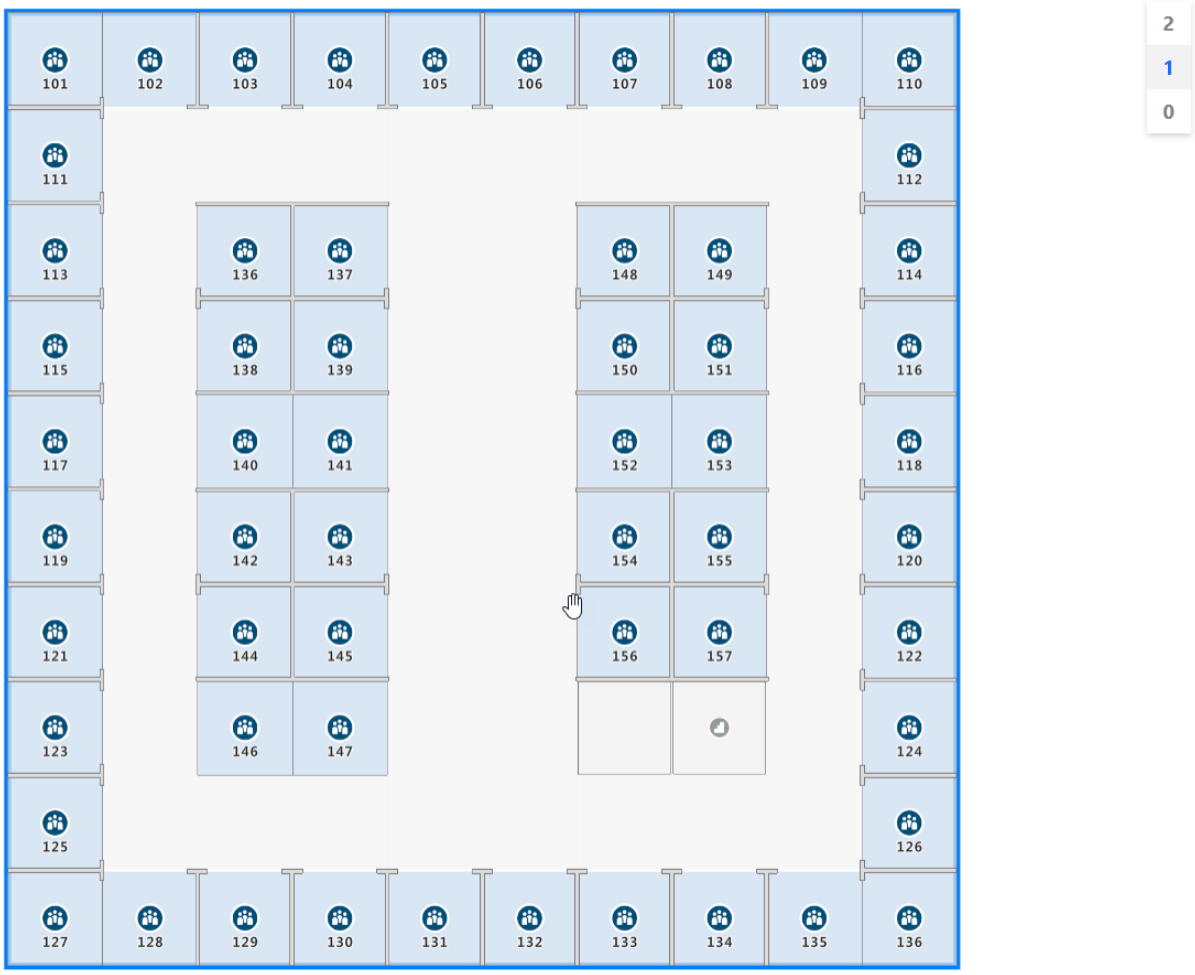
Your file should now look similar to the following HTML:

HTML

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, user-scalable=no" />
    <title>Indoor Maps App</title>

    <link rel="stylesheet"
href="https://atlas.microsoft.com/sdk/javascript/mapcontrol/2/atlas.min.css"
type="text/css" />
    <link rel="stylesheet"
href="https://atlas.microsoft.com/sdk/javascript/indoor/0.2/atlas-
indoor.min.css" type="text/css"/>

    <script
src="https://atlas.microsoft.com/sdk/javascript/mapcontrol/2/atlas.min.js">
</script>
    <script
src="https://atlas.microsoft.com/sdk/javascript/indoor/0.2/atlas-
indoor.min.js"></script>

    <style>
      html,
      body {
        width: 100%;
        height: 100%;
        padding: 0;
        margin: 0;
      }

      #map-id {
        width: 100%;
        height: 100%;
      }
    </style>
  </head>

  <body>
    <div id="map-id"></div>
    <script>
      const subscriptionKey = "<Your Azure Maps Subscription Key>";
      const mapConfig = "<Your map configuration id or alias>";
      const statesetId = "<Your statesetId>";
```

```
      const region = "<Your Creator resource region: us or eu>"
      atlas.setDomain(`${region}.atlas.microsoft.com`);

      const map = new atlas.Map("map-id", {
        //use your facility's location
        center: [-122.13315, 47.63637],
        //or, you can use bounds: [# west, # south, # east, # north] and
replace # with your Map bounds
        authOptions: {
            authType: 'subscriptionKey',
            subscriptionKey: subscriptionKey
        },
        zoom: 19,

        mapConfiguration: mapConfig,
        styleAPIVersion: '2022-09-01-preview'
      });

      const levelControl = new atlas.control.LevelControl({
        position: "top-right",
      });

      const indoorManager = new atlas.indoor.IndoorManager(map, {
        levelControl: levelControl, //level picker
        statesetId: statesetId // Optional
      });

      if (statesetId.length > 0) {
        indoorManager.setDynamicStyling(true);
      }

      map.events.add("levelchanged", indoorManager, (eventData) => {
        //put code that runs after a level has been changed
        console.log("The level has changed:", eventData);
      });

      map.events.add("facilitychanged", indoorManager, (eventData) => {
        //put code that runs after a facility has been changed
        console.log("The facility has changed:", eventData);
      });
    </script>
  </body>
</html>
```

To see your indoor map, load it into a web browser. It should appear like the following image. If you select the stairwell feature, the *level picker* appears in the upper right-hand corner.



For a live demo of an indoor map with available source code, see Creator Indoor Maps in the [Azure Maps Samples].

Next steps

Read about the APIs that are related to the *Azure Maps Indoor* module:

Drawing package requirements

Creator for indoor maps

Learn more about how to add more data to your map:

Indoor Maps dynamic styling

Code samples

Facility Ontology

Article • 04/13/2023

Facility ontology defines how Azure Maps Creator internally stores facility data in a Creator dataset. In addition to defining internal facility data structure, facility ontology is also exposed externally through the WFS API. When WFS API is used to query facility data in a dataset, the response format is defined by the ontology supplied to that dataset.

Changes and Revisions

The Facility 1.0 contains revisions for the Facility feature class definitions for Azure Maps services.

Major Changes

Fixed the following constraint validation checks:

- Constraint validation check for exclusivity of `isObstruction = true` *or* the presence of `obstructionArea` for `lineElement` and `areaElement` feature classes.
- Constraint validation check for exclusivity of `isRoutable = true` *or* the presence of `routeThroughBehavior` for the `category` feature class.

Feature collection

At a high level, the facility ontology consists of feature collections, each contains an array of feature objects. All feature objects have two fields in common, `ID` and `Geometry`. When importing a drawing package into Azure Maps Creator, these fields are automatically generated.

In addition to these common fields, each feature class defines a set of properties. Each property is defined by its data type and constraints. Some feature classes have properties that are dependent on other feature classes. Dependant properties evaluate to the `ID` of another feature class.

The remaining sections in this article define the different feature classes and their properties that make up the facility ontology in Microsoft Azure Maps Creator.

unit

The `unit` feature class defines a physical and non-overlapping area that can be occupied and traversed by a navigating agent. A `unit` can be a hallway, a room, a courtyard, and so on.

Geometry Type: Polygon

Property	Type	Required	Description
`originalId`	string	false	When the dataset is created through the conversion service, the original ID is automatically set to the Azure Maps internal ID. Maximum length allowed is 1,000 characters.
`externalId`	string	false	An ID used by the client to associate the feature with another feature in a different dataset, such as in an internal database. Maximum length allowed is 1,000 characters.
`categoryId`	category.Id	true	The ID of a category feature.
`isOpenArea`	boolean (Default value is `null`.)	false	Represents whether the unit is an open area. If set to `true`, structures don't surround the unit boundary, and a navigating agent can enter the `unit` without the need of an opening. By default, units are surrounded by physical barriers and are open only where an opening feature is placed on the boundary of the unit. If walls are needed in an open area unit, they can be represented as a lineElement or areaElement with an `isObstruction` property equal to `true`.
`navigableBy`	enum ["pedestrian", "wheelchair", "machine", "bicycle", "automobile", "hiredAuto", "bus", "railcar", "emergency", "ferry", "boat"]	false	Indicates the types of navigating agents that can traverse the unit. If unspecified, the unit is assumed to be traversable by any navigating agent.

Property	Type	Required	Description
`isRoutable`	boolean (Default value is `null`.)	false	Determines if the unit is part of the routing graph. If set to `true`, the unit can be used as source/destination or intermediate node in the routing experience.
`routeThroughBehavior`	enum ["disallowed", "allowed", "preferred"]	false	Determines if navigating through the unit is allowed. If unspecified, it inherits its value from the category feature referred to in the `categoryId` property. If specified, it overrides the value given in its category feature."
`nonPublic`	boolean	false	If `true`, the unit is navigable only by privileged users. Default value is `false`.
`levelId`	level.Id	true	The ID of a level feature.
`occupants`	array of directoryInfo.Id	false	The IDs of directoryInfo features. Used to represent one or many occupants in the feature.
`addressId`	directoryInfo.Id	false	The ID of a directoryInfo feature. Used to represent the address of the feature.
`addressRoomNumber`	directoryInfo.Id	true	Room/Unit/Apartment/Suite number of the unit.
`name`	string	false	Name of the feature in local language. Maximum length allowed is 1,000 characters.
`nameSubtitle`	string	false	Subtitle that shows up under the `name` of the feature. Can be used to display the name in a different language, and so on. Maximum length allowed is 1,000 characters.
`nameAlt`	string	false	Alternate name used for the feature. Maximum length allowed is 1,000 characters.
`anchorPoint`	Point	false	GeoJSON Point geometry that represents the feature as a point. Can be used to position the label of the feature.

zone

The `zone` feature class defines a virtual area, like a WiFi zone or emergency assembly area. Zones can be used as destinations but aren't meant for through traffic.

Geometry Type: Polygon

Property	Type	Required	Description
`originalId`	string	false	When the dataset is created through the conversion service, the original ID is automatically set to the Azure Maps internal ID. Maximum length allowed is 1,000 characters.
`externalId`	string	false	An ID used by the client to associate the feature with another feature in a different dataset, such as in an internal database. Maximum length allowed is 1,000 characters.
`categoryId`	category.Id	true	The ID of a category feature.
`setId`	string	true	Required for zone features that represent multi-level zones. The `setId` is the unique ID for a zone that spans multiple levels. The `setId` enables a zone with varying coverage on different floors to be represented with different geometry on different levels. The `setId` can be any string and is case-sensitive. It's recommended that the `setId` is a GUID. Maximum length allowed is 1,000 characters.
`levelId`	level.Id	true	The ID of a level feature.
`name`	string	false	Name of the feature in local language. Maximum length allowed is 1,000 characters.
`nameSubtitle`	string	false	Subtitle that shows up under the `name` of the feature. Can be used to display the name in a different language, and so on. Maximum length allowed is 1,000 characters.
`nameAlt`	string	false	Alternate name used for the feature. Maximum length allowed is 1,000 characters.
`anchorPoint`	Point	false	GeoJSON Point geometry that represents the feature as a point. Can be used to position the label of the feature.

level

The `level` class feature defines an area of a building at a set elevation. For example, the floor of a building, which contains a set of features, such as units.

Geometry Type: MultiPolygon

Property	Type	Required	Description
`originalId`	string	false	When the dataset is created through the conversion service, the original ID is automatically set to the Azure Maps internal ID. Maximum length allowed is 1,000 characters.
`externalId`	string	false	An ID used by the client to associate the feature with another feature in a different dataset, such as in an internal database. Maximum length allowed is 1,000 characters.
`facilityId`	facility.Id	true	The ID of a facility feature.
`ordinal`	integer	true	The level number. Used by the verticalPenetration feature to determine the relative order of the floors to help with travel direction. The general practice is to start with 0 for the ground floor. Add +1 for every floor upwards, and -1 for every floor going down. It can be modeled with any numbers, as long as the higher physical floors are represented by higher ordinal values.
`abbreviatedName`	string	false	A four-character abbreviated level name, like what would be found on an elevator button.
`heightAboveFacilityAnchor`	double	false	Vertical distance of the level's floor above facility.anchorHeightAboveSeaLevel, in meters.
`verticalExtent`	double	false	Vertical extent of the level, in meters. If not provided, defaults to facility.defaultLevelVerticalExtent.
`name`	string	false	Name of the feature in local language. Maximum length allowed is 1,000 characters.

Property	Type	Required	Description
`nameSubtitle`	string	false	Subtitle that shows up under the `name` of the feature. Can be used to display the name in a different language, and so on. Maximum length allowed is 1,000 characters.
`nameAlt`	string	false	Alternate name used for the feature. Maximum length allowed is 1,000 characters.
`anchorPoint`	Point	false	GeoJSON Point geometry that represents the feature as a point. Can be used to position the label of the feature.

facility

The `facility` feature class defines the area of the site, building footprint, and so on.

Geometry Type: MultiPolygon

Property	Type	Required	Description
`originalId`	string	false	When the dataset is created through the conversion service, the original ID is automatically set to the Azure Maps internal ID. Maximum length allowed is 1,000 characters.
`externalId`	string	false	An ID used by the client to associate the feature with another feature in a different dataset, such as in an internal database. Maximum length allowed is 1,000 characters.
`categoryId`	category.Id	true	The ID of a category feature.
`occupants`	array of directoryInfo.Id	false	The IDs of directoryInfo features. Used to represent one or many occupants in the feature.
`addressId`	directoryInfo.Id	true	The ID of a directoryInfo feature. Used to represent the address of the feature.

Property	Type	Required	Description
`name`	string	false	Name of the feature in local language. Maximum length allowed is 1,000 characters.
`nameSubtitle`	string	false	Subtitle that shows up under the `name` of the feature. Can be used to display the name in a different language, and so on. Maximum length allowed is 1,000 characters.
`nameAlt`	string	false	Alternate name used for the feature. Maximum length allowed is 1,000 characters.
`anchorPoint`	Point	false	GeoJSON Point geometry that represents the feature as a point. Can be used to position the label of the feature.
`anchorHeightAboveSeaLevel`	double	false	Height of anchor point above sea level, in meters. Sea level is defined by EGM 2008.
`defaultLevelVerticalExtent`	double	false	Default value for vertical extent of levels, in meters.

verticalPenetration

The `verticalPenetration` class feature defines an area that, when used in a set, represents a method of navigating vertically between levels. It can be used to model stairs, elevators, and so on. Geometry can overlap units and other vertical penetration features.

Geometry Type: Polygon

Property	Type	Required	Description
`originalId`	string	false	When the dataset is created through the conversion service, the original ID is automatically set to the Azure Maps internal ID. Maximum length allowed is 1,000 characters.

Property	Type	Required	Description
`externalId`	string	false	An ID used by the client to associate the feature with another feature in a different dataset, such as in an internal database. Maximum length allowed is 1,000 characters.
`categoryId`	category.Id	true	The ID of a category feature.
`setId`	string	true	Vertical penetration features must be used in sets to connect multiple levels. Vertical penetration features in the same set are considered to be the same. The `setId` can be any string, and is case-sensitive. Using a GUID as a `setId` is recommended. Maximum length allowed is 1,000 characters.
`levelId`	level.Id	true	The ID of a level feature.
`direction`	string enum ["both", "lowToHigh", "highToLow", "closed"]	false	Travel direction allowed on this feature. The ordinal attribute on the level feature is used to determine the low and high order.
`navigableBy`	enum ["pedestrian", "wheelchair", "machine", "bicycle", "automobile", "hiredAuto", "bus", "railcar", "emergency", "ferry", "boat"]	false	Indicates the types of navigating agents that can traverse the unit. If unspecified, the unit is traversable by any navigating agent.
`nonPublic`	boolean	false	If `true`, the unit is navigable only by privileged users. Default value is `false`.
`name`	string	false	Name of the feature in local language. Maximum length allowed is 1,000 characters.
`nameSubtitle`	string	false	Subtitle that shows up under the `name` of the feature. Can be used to display the name in a different language, and so on. Maximum length allowed is 1,000 characters.
`nameAlt`	string	false	Alternate name used for the feature. Maximum length allowed is 1,000 characters.

Property	Type	Required	Description
`anchorPoint`	Point	false	GeoJSON Point geometry that represents the feature as a point. Can be used to position the label of the feature.

opening

The `opening` class feature defines a traversable boundary between two units, or a `unit` and `verticalPenetration`.

Geometry Type: LineString

Property	Type	Required	Description
`originalId`	string	false	When the dataset is created through the conversion service, the original ID is automatically set to the Azure Maps internal ID. Maximum length allowed is 1,000 characters.
`externalId`	string	false	An ID used by the client to associate the feature with another feature in a different dataset, such as in an internal database. Maximum length allowed is 1,000 characters.
`categoryId`	category.Id	true	The ID of a category feature.
`levelId`	level.Id	true	The ID of a level feature.

Property	Type	Required	Description
`isConnectedToVerticalPenetration`	boolean	false	Whether or not this feature is connected to a `verticalPenetration` feature on one of its sides. Default value is `false`.
`navigableBy`	enum ["pedestrian", "wheelchair", "machine", "bicycle", "automobile", "hiredAuto", "bus", "railcar", "emergency", "ferry", "boat"]	false	Indicates the types of navigating agents that can traverse the unit. If unspecified, the unit is traversable by any navigating agent.
`accessRightToLeft`	enum ["prohibited", "digitalKey", "physicalKey", "keyPad", "guard", "ticket", "fingerprint", "retina", "voice", "face", "palm", "iris", "signature", "handGeometry", "time", "ticketChecker", "other"]	false	Method of access when passing through the opening from right to left. Left and right are determined by the vertices in the feature geometry, standing at the first vertex and facing the second vertex. Omitting this property means there are no access restrictions.
`accessLeftToRight`	enum ["prohibited", "digitalKey", "physicalKey", "keyPad", "guard", "ticket", "fingerprint", "retina", "voice", "face", "palm", "iris", "signature", "handGeometry", "time", "ticketChecker", "other"]	false	Method of access when passing through the opening from left to right. Left and right are determined by the vertices in the feature geometry, standing at the first vertex and facing the second vertex. Omitting this property means there are no access restrictions.

Property	Type	Required	Description
`isEmergency`	boolean	false	If `true`, the opening is navigable only during emergencies. Default value is `false`
`anchorPoint`	Point	false	GeoJSON Point geometry y that represents the feature as a point. Can be used to position the label of the feature.

directoryInfo

The `directoryInfo` object class feature defines the name, address, phone number, website, and hours of operation for a unit, facility, or an occupant of a unit or facility.

Geometry Type: None

Property	Type	Required	Description
`originalId`	string	false	When the dataset is created through the conversion service, the original ID is automatically set to the Azure Maps internal ID. Maximum length allowed is 1,000 characters.
`externalId`	string	false	An ID used by the client to associate the feature with another feature in a different dataset, such as in an internal database. Maximum length allowed is 1,000 characters.
`streetAddress`	string	false	Street address part of the address. Maximum length allowed is 1,000 characters.
`unit`	string	false	Unit number part of the address. Maximum length allowed is 1,000 characters.
`locality`	string	false	The locality of the address. For example: city, municipality, village. Maximum length allowed is 1,000 characters.
`adminDivisions`	array of strings	false	Administrative division part of the address, from smallest to largest (County, State, Country). For example: ["King", "Washington", "USA"] or ["West Godavari", "Andhra Pradesh", "IND"]. Maximum length allowed is 1,000 characters.

Property	Type	Required	Description
`postalCode`	string	false	Postal code part of the address. Maximum length allowed is 1,000 characters.
`name`	string	false	Name of the feature in local language. Maximum length allowed is 1,000 characters.
`nameSubtitle`	string	false	Subtitle that shows up under the `name` of the feature. Can be used to display the name in a different language, and so on. Maximum length allowed is 1,000 characters.
`nameAlt`	string	false	Alternate name used for the feature. Maximum length allowed is 1,000 characters.
`phoneNumber`	string	false	Phone number. Maximum length allowed is 1,000 characters.
`website`	string	false	Website URL. Maximum length allowed is 1,000 characters.
`hoursOfOperation`	string	false	Hours of operation as text, following the Open Street Map specification. Maximum length allowed is 1,000 characters.

pointElement

The `pointElement` is a class feature that defines a point feature in a unit, such as a first aid kit or a sprinkler head.

Geometry Type: MultiPoint

Property	Type	Required	Description
`originalId`	string	false	When the dataset is created through the conversion service, the original ID is automatically set to the Azure Maps internal ID. Maximum length allowed is 1,000 characters.
`externalId`	string	false	An ID used by the client to associate the feature with another feature in a different dataset, such as in an internal database. Maximum length allowed is 1,000 characters.
`categoryId`	category.Id	true	The ID of a category feature.

Property	Type	Required	Description
`unitId`	string	true	The ID of a unit feature containing this feature. Maximum length allowed is 1,000 characters.
`isObstruction`	boolean (Default value is `null`.)	false	If `true`, this feature represents an obstruction to be avoided while routing through the containing unit feature.
`name`	string	false	Name of the feature in local language. Maximum length allowed is 1,000 characters.
`nameSubtitle`	string	false	Subtitle that shows up under the `name` of the feature. Can be used to display the name in a different language, and so on. Maximum length allowed is 1,000 characters.
`nameAlt`	string	false	Alternate name used for the feature. Maximum length allowed is 1,000 characters.

lineElement

The `lineElement` is a class feature that defines a line feature in a unit, such as a dividing wall or window.

Geometry Type: LinearMultiString

Property	Type	Required	Description
`originalId`	string	false	When the dataset is created through the conversion service, the original ID is automatically set to the Azure Maps internal ID. Maximum length allowed is 1,000 characters.
`externalId`	string	false	An ID used by the client to associate the feature with another feature in a different dataset, such as in an internal database. Maximum length allowed is 1,000 characters.
`categoryId`	category.Id	true	The ID of a category feature.
`unitId`	unitId	true	The ID of a unit feature containing this feature.
`isObstruction`	boolean (Default value is `null`.)	false	If `true`, this feature represents an obstruction to be avoided while routing through the containing unit feature.

Property	Type	Required	Description
`name`	string	false	Name of the feature in local language. Maximum length allowed is 1,000 characters.
`nameSubtitle`	string	false	Subtitle that shows up under the `name` of the feature. Can be used to display the name in a different language, and so on. Maximum length allowed is 1,000 characters.
`nameAlt`	string	false	Alternate name used for the feature. Maximum length allowed is 1,000 characters.
`anchorPoint`	Point	false	GeoJSON Point geometry that represents the feature as a point. Can be used to position the label of the feature.
`obstructionArea`	Polygon or MultiPolygon	false	A simplified geometry (when the line geometry is complicated) of the feature that is to be avoided during routing. Requires `isObstruction` set to true.

areaElement

The `areaElement` is a class feature that defines a polygon feature in a unit, such as an area open to below, an obstruction like an island in a unit.

Geometry Type: MultiPolygon

Property	Type	Required	Description
`originalId`	string	false	When the dataset is created through the conversion service, the original ID is automatically set to the Azure Maps internal ID. Maximum length allowed is 1,000 characters.
`externalId`	string	false	An ID used by the client to associate the feature with another feature in a different dataset, such as in an internal database. Maximum length allowed is 1,000 characters.
`categoryId`	category.Id	true	The ID of a category feature.
`unitId`	unitId	true	The ID of a unit feature containing this feature.
`isObstruction`	boolean	false	If `true`, this feature represents an obstruction to be avoided while routing through the containing unit feature.

Property	Type	Required	Description
`obstructionArea`	Polygon or MultiPolygon	false	A simplified geometry (when the line geometry is complicated) of the feature that is to be avoided during routing. Requires `isObstruction` set to true.
`name`	string	false	Name of the feature in local language. Maximum length allowed is 1,000 characters.
`nameSubtitle`	string	false	Subtitle that shows up under the `name` of the feature. Can be used to display the name in a different language, and so on. Maximum length allowed is 1,000 characters.
`nameAlt`	string	false	Alternate name used for the feature. Maximum length allowed is 1,000 characters.
`anchorPoint`	Point	false	GeoJSON Point geometry that represents the feature as a point. Can be used to position the label of the feature.

category

The `category` class feature defines category names. For example: "room.conference".

Geometry Type: None

Property	Type	Required	Description
`originalId`	string	false	The category's original ID derived from client data. Maximum length allowed is 1,000 characters.
`externalId`	string	false	An ID used by the client to associate the category with another category in a different dataset, such as in an internal database. Maximum length allowed is 1,000 characters.
`name`	string	true	Name of the category. Suggested to use "." to represent hierarchy of categories. For example: "room.conference", "room.privateoffice". Maximum length allowed is 1,000 characters.
`routeThroughBehavior`	boolean	false	Determines whether a feature can be used for through traffic.

Property	Type	Required	Description
`isRoutable`	boolean (Default value is `null`.)	false	Determines if a feature should be part of the routing graph. If set to `true`, the unit can be used as source/destination or intermediate node in the routing experience.

Next steps

Learn more about Creator for indoor maps by reading:

Creator for indoor maps

Azure digital twins builder

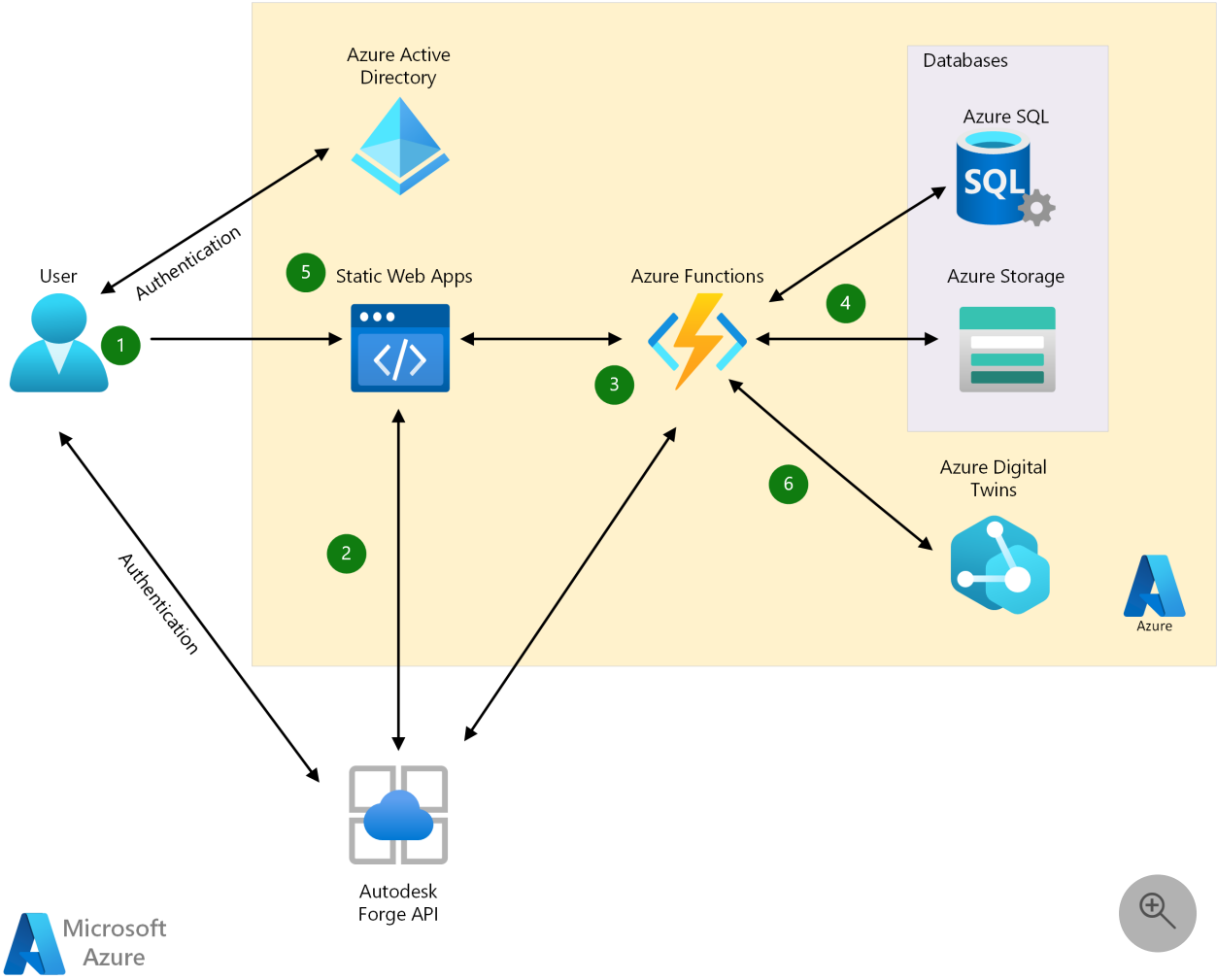
Azure Digital Twins | Azure Functions | Azure Static Web Apps

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 uses Azure Digital Twins and other Azure services to enable the process more effectively, from building information modeling (BIM) to digital twins (DT). The solution in this article uses Autodesk Forge data specifically, but using BIM data from other providers can work also. The idea is broadly applicable to the process of creating a DT from BIM data.

Architecture



Download a Visio file of this architecture.

Dataflow

1. A web app that was built by using Static Web Apps presents two buttons to the user. One of them initiates sign-in to Azure, the other initiates sign-in to Autodesk BIM 360. The user signs into both accounts.
2. The web app uses the Autodesk Forge API to build a list of the BIM models that are shared with the BIM 360 account. The user selects a model for the web app to display, which it does by once again making use of the Autodesk Forge API.
3. The user selects **Parse Model**, which triggers a request to a function that was built by using Azure Functions. The function uses the Autodesk Forge API to extract the metadata of the model that the user selected.
4. Data such as an entity's family type—for example, space, HVAC, mechanical—and its relationship to other entities—for example, the space where an asset is located—is stored by the function in an Azure SQL database. The function uses this data to map the entities to Digital Twin Definition Language (DTDL) models. The results are returned to the web app in a table for the user to view.
5. The user updates the table to correct any mapping mistakes, and to add additional initialization properties to the digital twins that are instantiated from this data. The user can also remove records and create additional assets like sensors or IoT devices.
6. When satisfied that the table is correct, the user selects **Upload to ADT** to cause the web app to load the table to Azure Digital Twins as a DT.

Components

- Azure Digital Twins stores digital representations of IoT devices and environments. You can use this data for data propagation or real-time analysis. Azure Digital Twins:
  - Models environments with DTDL.
  - Has a REST API for entering data.
  - Provides SDKs that support control and data plane operations for various languages.

  You can build ontologies by using DTDL. You can also start with an industry-supported model such as the Digital Twins Definition Language-based RealEstateCore ontology for smart buildings.

- Autodesk Forge is the API service for Autodesk and BIM 360. It provides the model viewer and BIM models metadata.

- Static Web Apps is an Azure service that, in this architecture, hosts the front-end code of the web app. The Autodesk Forge Viewer API is well supported by React.
- Azure Functions is an event-driven serverless compute platform. It runs on demand and at scale in the cloud. With it, you can develop without having to worry about managing a server. In this architecture, it hosts the back-end code for the web app.
- Azure SQL Database stores and parses the extracted metadata from a model. Due to the complexity of a model's metadata, Autodesk recommends parsing the data by uploading it to Azure SQL Database in a specific structure, and then using a query that Autodesk provides.
- Azure Storage provides temporary storage for data that will be used to create a DT, but isn't yet ready for publishing by Azure Twins Service.

Scenario details

Creating a digital twins (DT) solution for a facility requires extensive data entry that's done by using either Azure native tools or external tools. The data often requires interpretations of drawings, documentation, and data models of other relevant systems of record. Such interpretations risk data fidelity, and cause inefficiencies in the DT implementation process, inefficiencies that hinder the adoption of DTs.

Architecture, engineering, and construction (AEC) professionals use building information modeling (BIM) as a foundation for efficient design and construction processes. BIM is the most effective tool for modeling a facility and its supporting systems, and produces a flexible, parametric, and relational dataset that can comprehensively document a built asset. It's becoming the industry standard tool for planning and managing the built environment.

BIM data is exceptionally valuable, and it can be improved with a digital representation, based on sensor telemetry, of the physical environment. Such a representation is known as a digital twin (DT). DTs help real estate owners and operators manage the operations of their buildings. This is the digital building lifecycle approach to managing.

This solution idea uses Azure Digital Twins and other Azure services to enable this process more effectively, from BIM to DT. Creating a DT for a facility requires extensive data entry that's done by using either Azure native tools or external tools. The data often requires interpretations of drawings, documentation, and data models of other relevant systems of record. Such interpretations risk data fidelity, and cause

inefficiencies in the DT implementation process, inefficiencies that hinder the adoption of DTs.

The key to the solution is having a web app that uses BIM data from Autodesk Forge to automate the creation of an Azure Digital Twins foundational dataset. The app provides both visual and relational context to support the instantiation of a DT in the Azure Digital Twins build process.

By providing the basis for holistic, responsive, and automated building management systems, this app helps address the vast range of challenges that arise when managing buildings digitally at large scale.

The solution in this article uses Autodesk Forge data specifically, but using BIM data from other providers can work also. The idea is broadly applicable to the process of creating a DT from BIM data.

> **Important**
>
> This document is created strictly for informative purposes to demonstrate how Autodesk Forge API can supply BIM data to a web app that creates a DT. Your use of third-party applications is subject to terms between you and the third party. Microsoft Corporation is not affiliated with, is not a partner to, and does not endorse or sponsor Autodesk or any of Autodesk's products. There are other sources of BIM data that you can use to create DTs.

Potential use cases

This solution is ideal for the facilities, real-estate, manufacturing, energy, and government industries.

BIM models can describe many structures besides offices, including:

- Datacenters.
- Factories.
- Power plants.
- Bridges.

These structures become more intelligent and advanced as sensors and connected devices become smaller and more affordable. Azure Digital Twins can bring greater accuracy, control, and predictability to the building owner’s building management data. You can, for example, manage frequency sensors within Azure Digital Twins for predictive maintenance of building roof chillers.

Real estate portfolio managers can use BIM and DTs to improve their understanding of the elements within defined spaces, for better building space management.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal authors:

- Marcus Farquhar | Technology Innovation and Experimentation Lead
- Kian Lutu | Program Manager for the Center of Innovation

To see non-public LinkedIn profiles, sign in to LinkedIn.

Next steps

- Azure Digital Twins Documentation
- Autodesk Forge
- Digital Twins Definition Language (GitHub)
- RealEstateCore, a smart building ontology for digital twins (video)
- RealEstateCore ontology

Related resources

- Azure IoT reference architecture
- Create smart places by using Azure Digital Twins
- Facilities management powered by mixed reality and IoT
- Cognizant Safe Buildings with IoT and Azure
- COVID-19 safe environments with IoT Edge monitoring and alerting
- Facilities architectures

Cognizant Safe Buildings with IoT and Azure

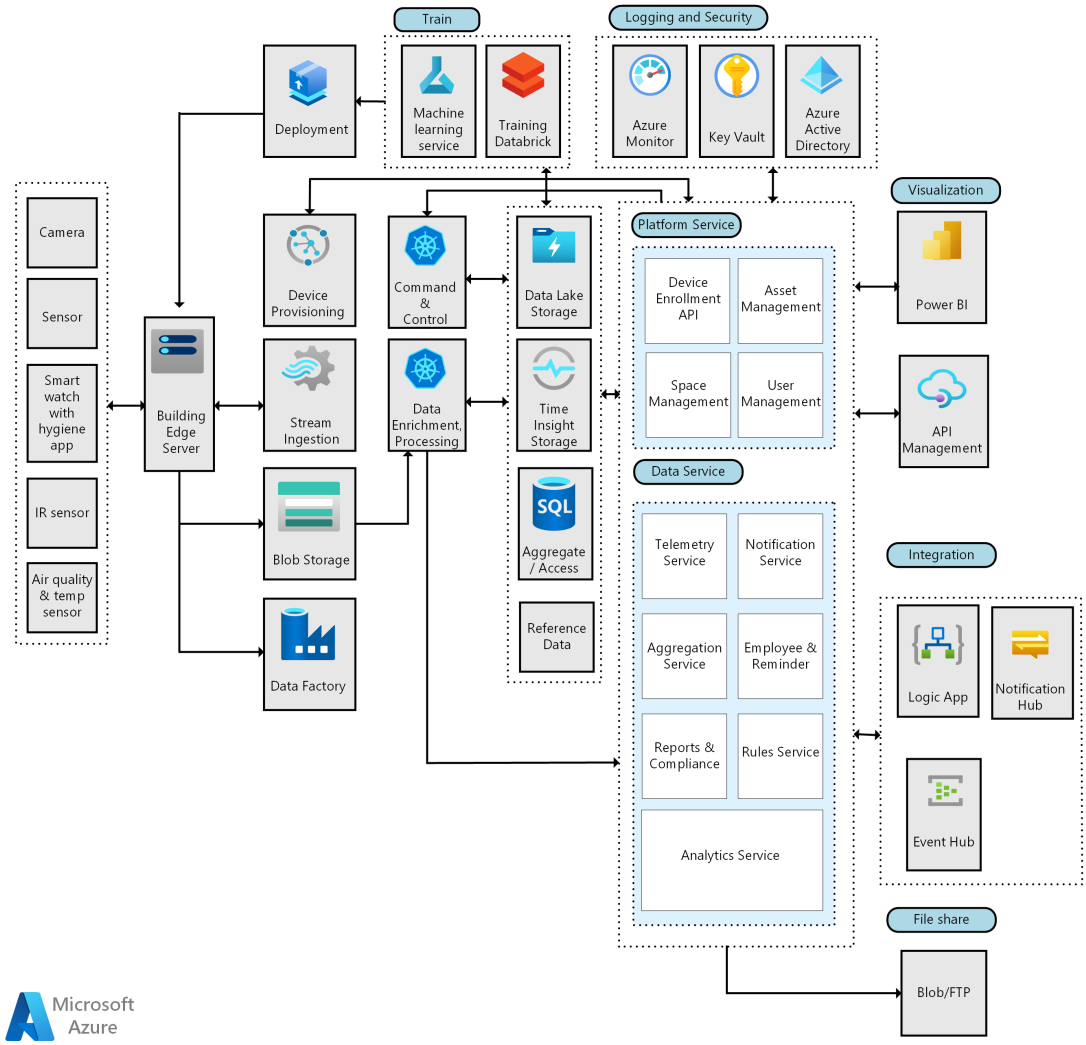
Azure IoT Edge | Azure IoT Hub | Azure Kubernetes Service (AKS) | Azure Stream Analytics | Azure Databricks

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**.

By using Azure IoT Hub and other Azure services, the **Cognizant Safe Buildings** solution brings together people, regulations, analytics, and technology to transform smart buildings into safe buildings. The solution collects biometric and environmental data from different devices. When the system detects any deviation to health and safety protocol in a building, the solution activates.

Architecture



Download a Visio file of this architecture.

Dataflow

1. Different devices collect data: thermal imaging cameras, LIDAR sensors, occupancy sensors, smart watches with hygiene apps, IR sensors, and air quality and temperature sensors. That data flows into the building's IoT Edge server.

2. The server feeds the data to the Azure IoT Hub, Azure Data Factory, and Azure Blob storage.

3. The Azure IoT Hub sends data through an Azure Kubernetes Service (AKS).

4. AKS routes the data so Azure Stream Analytics and Azure Databricks can analyze and enrich it.

5. **Safe Buildings** sends the processed data to various data stores:

    - Data Lake - Azure Data Lake Storage

- Time Insight - Azure Cosmos DB and Azure Database for PostgreSQL
- Aggregate/Access data - Azure SQL Database
- Reference Data - Azure Cache for Redis

6. From there, different services and APIs consume the data:
    - Data Service - These services consume the data, process it, and pass it on to downstream services that notify users of any abnormalities that **Safe Buildings** detected.
    - Platform Service - These services analyze device, asset, spatial, and user metrics and pass the information to a web app instance of Power BI. Power BI displays the statistics in a meaningful and useful way.
    - Training - Azure Databricks and Azure Machine Learning take the data and update deep learning models. **Safe Buildings** uses Azure Container Registry to distribute the models to the IoT Edge server and the Data Service.
    - Command and Control - AKS passes the data from the Platform Service to Azure Functions.

Components

- Azure IoT Edge allows you to deploy your cloud workloads to run on Internet of Things (IoT) edge devices via standard containers. Workloads like artificial intelligence, Azure and third-party services, or your own business logic.
- Azure IoT Hub enables highly secure and reliable communication between your IoT application and the devices it manages.
- Azure Kubernetes Service deploy and manage containerized applications more easily with a fully managed Kubernetes service.
- Azure Stream Analytics is an easy-to-use, real-time analytics service designed for mission-critical workloads.
- Azure Databricks unlocks insights from all your data and build artificial intelligence (AI) solutions. Set up your Apache Spark environment in minutes, autoscale, and collaborate on shared projects in an interactive workspace.
- Azure Data Lake Storage - A highly scalable and cost-effective data lake solution for big data analytics.

- Azure Cosmos DB - A fully managed NoSQL database service for modern app development. Azure Cosmos DB has guaranteed single-digit millisecond response times. It also guarantees 99.999-percent availability backed by SLAs, automatic and instant scalability, and open-source APIs for MongoDB and Cassandra.
- Azure Database for PostgreSQL - Focus on application innovation, not database management, with fully managed and intelligent Azure Database for PostgreSQL.
- Azure SQL Database - The intelligent, scalable, relational database service built for the cloud. It's evergreen and always up to date, with AI-powered and automated features that optimize performance and durability for you.
- Azure Cache for Redis - Fully managed, open source–compatible in-memory data store to power fast, scalable applications.
- Azure Machine Learning empowers developers and data scientists to build, train, and deploy machine learning models faster. Continuous retraining, updating, and improving data collection and analysis makes the models better over time.
- Azure Monitor collects, analyzes, and acts on telemetry data from your Azure and on-premises environments.
- Power BI enables team members to discover insights hidden in your data.
- Azure Functions lets you build and debug locally without another setup, deploy and operate at scale in the cloud, and integrate services using triggers and bindings.
- In addition, **Safe Buildings** has access to these other Azure services: Microsoft Entra ID, Azure API Management, Azure Blob storage, Azure Container Registry, Azure Data Factory, Azure Event Hubs, Azure Key Vault, Azure Logic Apps, Azure Notification Hubs, Azure VMs, and Azure Web Apps.

Scenario details

The world is cautiously reopening in a physical business environment where COVID-19 remains a factor. To help keep people healthy, Cognizant adapted their OneFacility solution to make workspaces safer for everyone who enters them. The **Cognizant Safe Buildings** solution provides a strategic approach to create safe buildings with sustainably healthy working environments.

The solution monitors critical insights so you can act and respond to risks. It also builds confidence that you're focused on making sure people are safe. The measurements the

solution reacts to are: human body temperature, effective physical distancing, hand sanitization compliance, and air quality tracking.

By using Azure IoT Hub and other Azure services, the **Cognizant Safe Buildings** solution brings together people, regulations, analytics, and technology to transform smart buildings into safe buildings. The solution collects biometric and environmental data from different devices. When the system detects any deviation to health and safety protocol in a building, the solution activates. **Safe Buildings** layers safety controls to protect, monitor, and respond with real-time alerts.

Potential use cases

Safe Buildings adapts to stores, offices, factories, warehouses, recreation spaces, hotels, and restaurants. It's adaptable to any interior space where people congregate. This solution is ideal for the facilities, real estate, retail, travel and hospitality, and healthcare industries.

Next steps

- For more information, please contact iotcovidsupport@microsoft.com.
- Visit the Microsoft commercial marketplace for info on Cognizant's OneFacility.

Related resources

- Azure IoT reference architecture
- COVID-19 safe environments with IoT Edge monitoring and alerting
- Create smart places by using Azure Digital Twins
- UVEN smart and secure disinfection and lighting

COVID-19 safe environments with IoT Edge monitoring and alerting

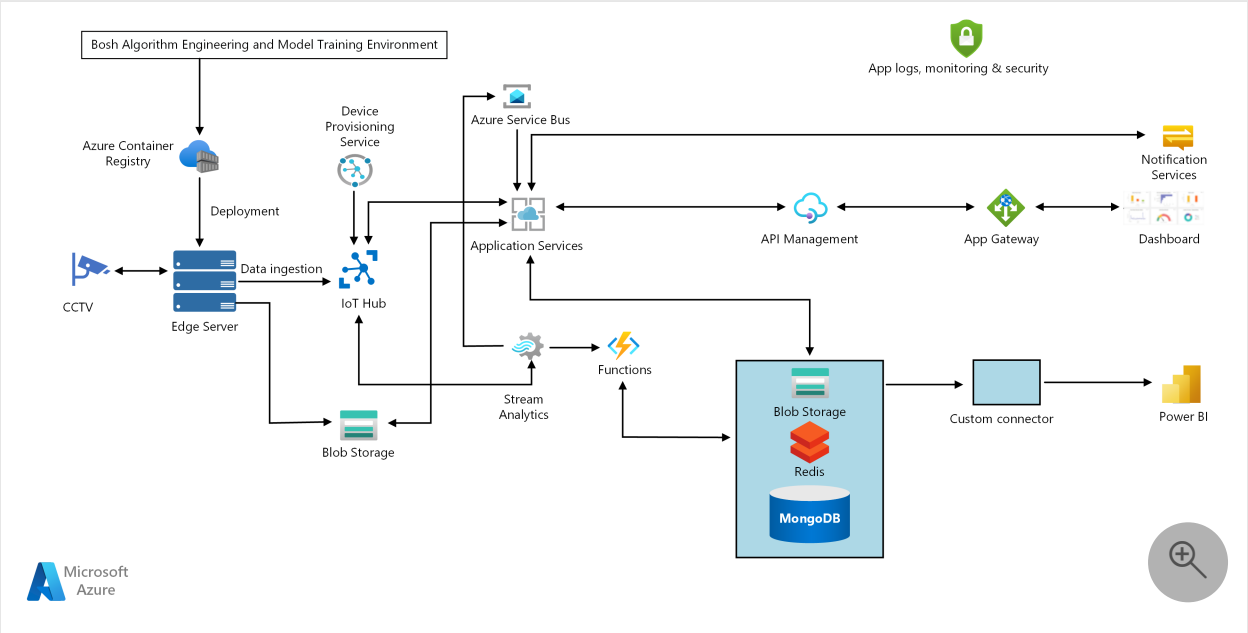
Azure Digital Twins | Azure IoT Edge | Azure Kubernetes Service (AKS) | Azure Service Bus | Azure Stream Analytics | Azure Monitor

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 article showcases a COVID-19 safe solution that is being implemented in workplace facilities across Europe, North America, and Latin America.

Architecture



Download a Visio file of this architecture.

Dataflow

1. CCTVs send video data to Internet of Things (IoT) edge servers. Edge computing handles device registration, provisioning, and data ingestion.

2. The Bosch Algorithm Engineering and Model Training Environment uses custom vision analytics to continually retrain machine learning (ML) models, and directly updates edge servers.
3. Edge servers send data to onboard and cloud stream analytics and blob storage. IoT Edge intelligent devices limit costs by preprocessing and sending only necessary data to the cloud.
4. Stream analytics perform data enrichment and validation on both edge and cloud data.
5. Service bus device-to-cloud and cloud-to-device messaging send data and telemetry to and from the cloud.
6. An interactive visual dashboard app provides near real-time compliance monitoring and alerts.
7. The Azure cloud provides application logs, monitoring, security, application gateway, and API management for the app.
8. Redis, MongoDB, and blob storage store cloud data for Power BI analytics and visualizations via a custom connector.
9. The app sends notifications and alerts to stakeholders via Microsoft Teams.

Components

- Azure IoT Edge servers with onboard storage, computing, artificial intelligence (AI), and machine learning (ML) capabilities can quickly recognize and respond to sensor input.
- Bosch video analytics use Custom Vision skills and Azure Machine Learning to continually improve monitoring, detection, and real-time alert triggering.
- Azure Digital Twins IoT service creates comprehensive models of physical environments in a spatial intelligence graph. Rather than simply tracking individual devices, Digital Twins can virtually replicate the physical world by modeling the relationships between people, places, and devices.
- Azure Stream Analytics (ASA) provides real-time serverless stream processing that can run the same queries in the cloud and on the edge. ASA on IoT Edge can filter or aggregate data that needs to be sent to the cloud for further processing or storage.
- Azure Storage provides flexible, scalable, secure Blob storage for unstructured data in the Azure cloud or on the IoT Edge. The current solution also uses Redis and MongoDB data storage.
- Azure Service Bus messaging through Azure IoT Hub connects devices to Azure cloud resources, and can use queries to filter data to be sent to the cloud.
- Azure Kubernetes Service (AKS) is a managed service for developing, deploying, and managing containerized applications. In this solution, AKS manages an

interactive visual dashboard app that tracks and analyzes safety violations.

- By integrating with the Azure cloud, the solution can use services like Azure Monitor, Microsoft Defender for Cloud, and Microsoft Entra ID.
- Integration with Microsoft Teams allows automated notifications of relevant stakeholders like HR and Security.
- Microsoft Power BI visualizations enable well-informed and data-driven reporting and decision making.

Scenario details

As workplaces and other group spaces reopen after the COVID-19 shutdown, people need to follow health and safety guidelines for safe social distancing, mask and PPE use, and occupancy limits. **Bosch COVID-19 Safe Solution** combines existing closed-circuit TV (CCTV) infrastructure with the Azure intelligent edge and other Azure and Microsoft services to help organizations monitor, follow, and improve these health and safety practices.

This solution applies across industries as diverse as steel manufacturing, building construction, and automotive assembly. The goals of the solution are to:

- Ensure a safe work environment when resuming manufacturing after the COVID-19 lockdown.
- Monitor and enforce compliance with face mask policy, social distancing, and occupancy limits on factory premises.
- Increase visibility and control through efficient reporting and interactive dashboards.
- Deliver alerts and notifications so health and safety stakeholders could address safety violations and concerns.
- Improve compliance, reduce violations, and enable well-informed safety decisions over time.

Potential use cases

- Spaces with existing CCTV infrastructure.
- Public areas including stores, restaurants, public transportation, hospitals, schools, and entertainment and recreation spaces.
- Work areas including offices, factories, and warehouses.
- Organizations with multiple locations, to enable widespread, systemic data analysis and actions.

Next steps

For more information about this solution:

- Contact iotcovidsupport@microsoft.com
- See https://www.bosch-india-software.com

Product documentation:

- What is Azure Digital Twins?
- Azure Stream Analytics on IoT Edge
- Store data at the edge with Azure Blob Storage on IoT Edge

Learning paths:

- Build the intelligent edge with Azure IoT Edge
- Implement a data streaming solution with Azure Streaming Analytics

Related resources

- Azure IoT reference architecture
- Cognizant Safe Buildings with IoT and Azure
- Create smart places by using Azure Digital Twins
- UVEN smart and secure disinfection and lighting

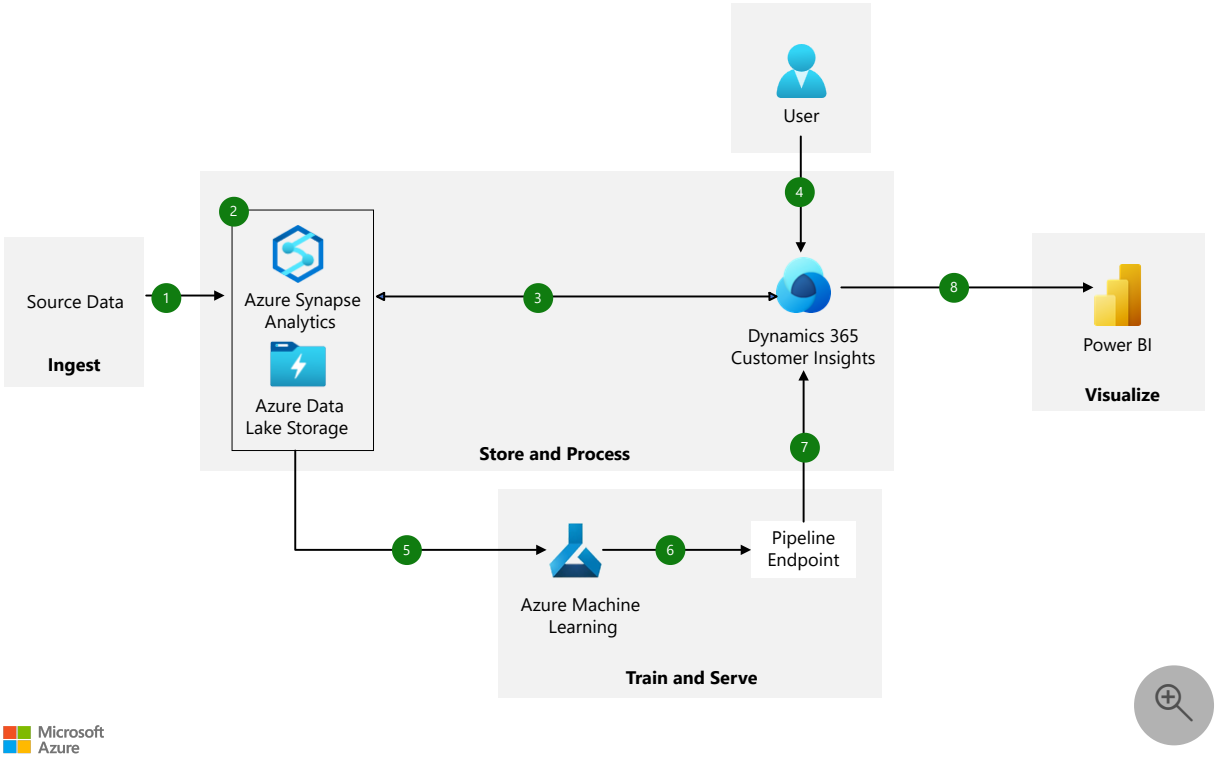
Customer 360 with Azure Synapse and Dynamics 365 Customer Insights

Customer Insights - Data | Azure Synapse Analytics | Azure Machine Learning | Power BI

This solution combines Azure Synapse Analytics with Dynamics 365 Customer Insights, to build a comprehensive view that presents your customer data and to provide the best customer experience.

Apache®, Apache Ignite, Ignite, 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.

Architecture



Download a Visio file of this architecture.

Dataflow

1. Azure Synapse Analytics ingests raw source data by using Azure Synapse pipelines.
2. Source data is stored in Azure Synapse Analytics and Azure Data Lake Storage Gen2.
3. Dynamics 365 Customer Insights connects to customer data from Azure Synapse.
4. Administrators configure unified customer profiles in Customer Insights, together with measures, segments, and enrichments. Unified customer profiles are ported from Customer Insights to Azure Synapse.
5. Administrators use the unified customer profile in Azure Synapse to create an Azure Machine Learning pipeline for retention prediction.
6. Administrators create a retention prediction model endpoint.
7. Administrators create a custom model workflow in Customer Insights to call the Azure Machine Learning pipeline to get the retention predictions.
8. Power BI ingests the Customer 360 data from Customer Insights to visualize the profiles and metrics.

Components

- Dynamics 365 Customer Insights can help you provide unmatched customer experiences by using world-class AI and analytics. Here, it's used to unify, segment, and enrich customer data.
- Azure Synapse Analytics is an analytics service that brings together data integration, enterprise data warehousing, and big data analytics. It's used here for data ingestion, storage, and processing.
- Data Lake Storage provides a massively scalable and secure data lake for your high-performance analytics workloads.
- Azure Machine Learning is an end-to-end machine learning service. It's used in this architecture to predict customer retention.
- Power BI can help you turn your data into coherent, visually immersive, and interactive insights. It's used here to visualize customer profiles and metrics.

Scenario details

Managing customer data from multiple sources and building a unified Customer 360 view isn't a new challenge. But it is becoming increasingly difficult with the increased number of interaction channels and touchpoints with customers. By combining Azure Synapse Analytics with Dynamics 365 Customer Insights, you can build a comprehensive view of your customers to provide the best customer experience.

Potential use cases

This solution was created for a property management organization. It can also be applied in industries like retail, financial services, manufacturing, and health care. It can be used by any organization that needs to bring data together across systems to build a Customer 360 profile and improve the customer experience.

You can use this solution to:

- Gain better insights from your customer data.
- Target sources of customer churn or dissatisfaction.
- Direct account and customer service activities.
- Run targeted promotions that are aimed at customer retention or upselling.

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.

This solution uses Microsoft Entra ID to authenticate users to the Azure solutions in the architecture. You can manage permissions via Microsoft Entra authentication or role-based access control.

Follow these security guidelines when you implement this solution:

- Security in Azure
- Access control for Azure Synapse
- User permissions for Customer Insights

Scalability

This solution uses Azure Synapse Spark clusters, which can be automatically scaled up and down based on the activity needs of your workload. For more information, see Azure Synapse Spark cluster autoscaling.

Azure Machine Learning training pipelines can be scaled up and down based on data size and other configuration parameters. The compute clusters support autoscaling and automatic shutdown to optimize for performance and cost.

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.

Dynamics 365 Customer Insights license pricing options are based on the number of customer profiles needed.

Azure Synapse Analytics has various pricing options to help you optimize costs. You can perform big data processing tasks like data engineering, data preparation, and machine learning directly in Azure Synapse by using memory-optimized or hardware-accelerated Apache Spark pools. Billing for usage of Spark pools is rounded up to the nearest minute.

Azure Machine Learning has no additional license charge. However, there are charges for compute and other Azure services that you consume, including but not limited to Azure Blob Storage, Azure Key Vault, Azure Container Registry, and Application Insights.

There are various Power BI product options to meet different requirements. Power BI Embedded provides an Azure-based option for embedding Power BI functionality in your applications.

You can deploy this solution with the following options.

- Dynamics 365 Customer Insights: 1,500 profiles
- Azure Synapse Analytics: Memory-optimized Spark cluster of medium size (8 vCores / 64 GB)
- Azure Machine Learning
  - Compute instance of type Standard_DS11_v2
  - Compute cluster of type Standard_D2_v2

Azure services like Azure Storage accounts, Key Vault, Container Registry, Application Insights, and so on, that are deployed with Azure Synapse Analytics and Azure Machine Learning incur other costs.

Deploy this scenario

To deploy this solution, follow the steps in the Getting Started guide and the step-by-step Deployment Guide. You can find them in the GitHub repository for the solution.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- Nalini Chandhi | Sr. Technical Specialist

Next steps

- Unlock customer intent with Dynamics 365 Customer Insights
- Product overview for Dynamics 365 Customer Insights
- What is Azure Machine Learning?
- What is Azure Synapse Analytics?

Related resources

- Enhanced customer dimension with Dynamics 365 Customer Insights
- Modern data warehouse for small and medium businesses
- Clinical insights with Microsoft Cloud for Healthcare
- Analytics architecture design

Facilities management powered by mixed reality and IoT

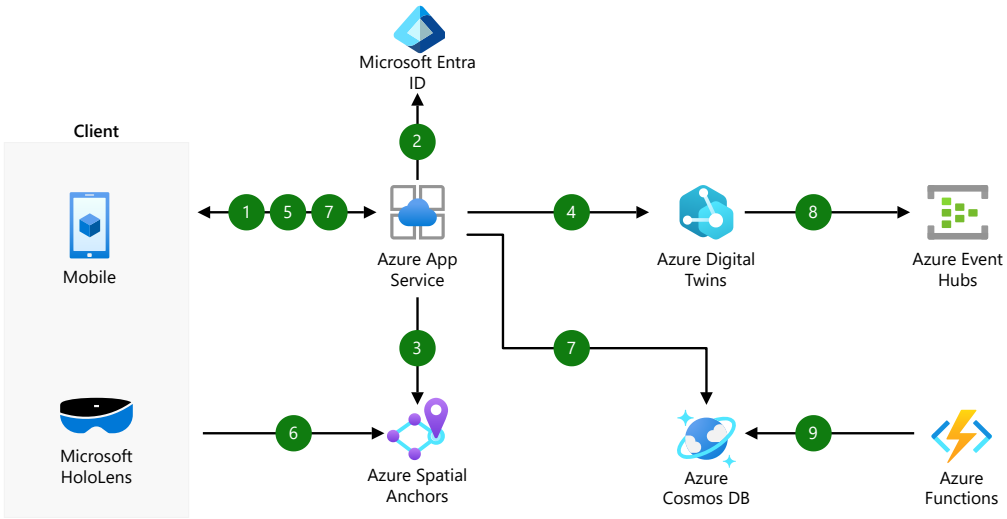
Azure Cosmos DB | Azure Digital Twins | Azure Event Hubs | Azure Functions | Azure Spatial Anchors

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**.

Improve uptime and operations in travel and hospitality, manufacturing, retail, and more with mixed reality and IoT.

Architecture



Download a Visio file of this architecture.

Dataflow

1. The client authenticates to the facilities management web service and specifies the space's name in the Azure Digital Twins object model.
2. The client's web service authenticates itself to Microsoft Entra ID.
3. The Microsoft Entra token is then sent to the Azure Spatial Anchors service to retrieve an access token for the client to use later.
4. Your app service retrieves information about the IoT sensors present in the area specified by the client. It returns IoT sensor IDs and their anchor IDs in Azure Spatial Anchors.
5. The Azure Spatial Anchors authorization token is returned to the client alongside the anchor IDs of the IoT sensors and other metadata required by the client application.
6. The client application completes a visual scan of the environment and retrieves its position in the area. It retrieves the position of all nearby anchors by using the nearby API of Azure Spatial Anchors.
7. The client application requests IoT sensor data and controls to be displayed as holograms in the space where the sensors exist, making it easy for the operator to detect and fix any issues. The data is fetched by the app's web service from Azure Cosmos DB, the service storing this data.
8. When IoT sensor data is updated, Azure Digital Twins pushes it to Event Hubs.
9. Azure Functions uses an Event Hubs trigger to process the change and update data in Azure Cosmos DB as needed.

Components

- Spatial Anchors: Create multi-user, spatially aware mixed-reality experiences.
- Microsoft Entra ID: Synchronize on-premises directories and enable single sign-on.
- Azure Cosmos DB: Globally distributed, multi-model database for any scale.
- App Service: Quickly create powerful cloud apps for web and mobile.
- Event Hubs: Receive telemetry from millions of devices.
- Azure Digital Twins: Build next-generation IoT spatial intelligence solutions.

Scenario details

Potential use cases

This scenario shows how you can visualize a virtual replica of your physical space with real-time data in the context of your environment. It's built on Azure Spatial Anchors

and Azure Digital Twins.

Next steps

- Share Spatial Anchors across devices
- Create a new tenant in Microsoft Entra ID
- Build a .NET web app with Azure Cosmos DB for NoSQL and the Azure portal
- Authenticate and authorize users end-to-end in Azure App Service
- Azure Event Hubs - A big data streaming platform and event ingestion service
- Deploy Azure Digital Twins and configure a spatial graph
- Introduction to Azure Functions

Related resources

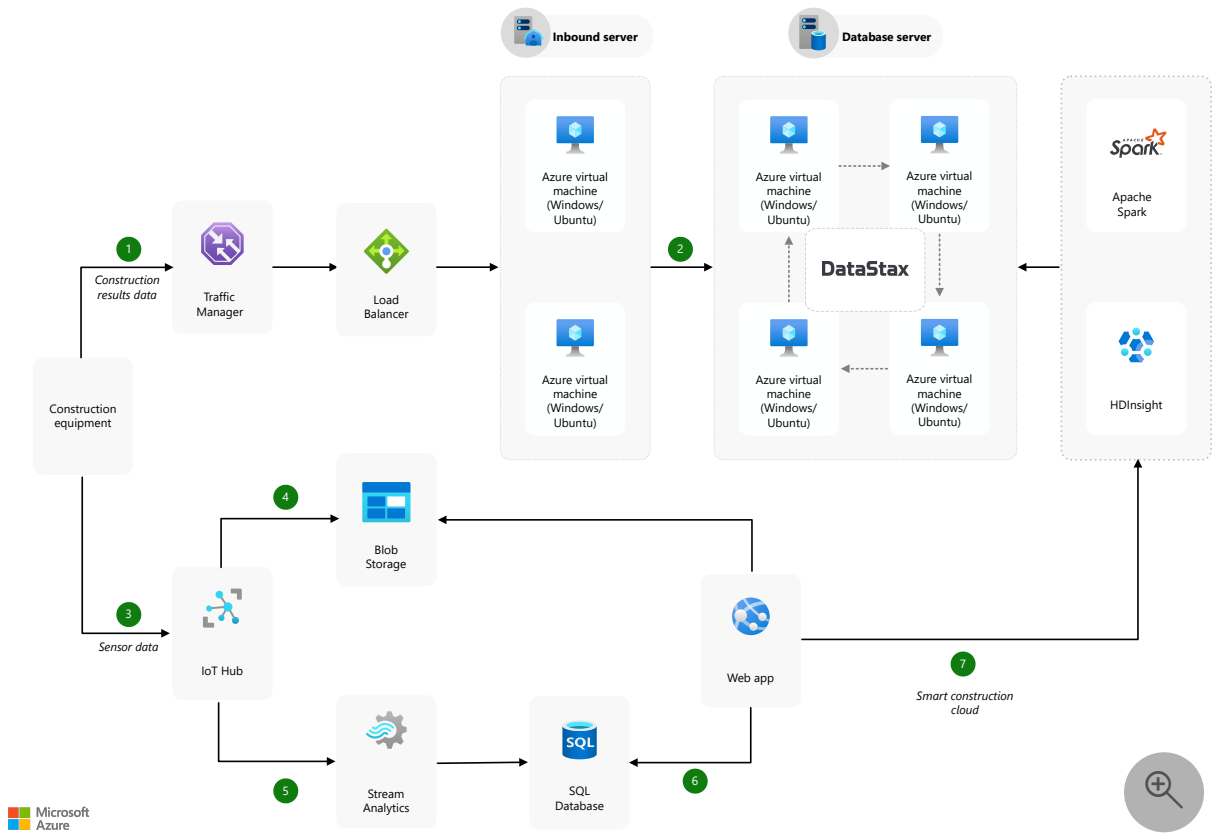
- Azure digital twins builder
- Training and procedural guidance powered by mixed reality
- Cognizant Safe Buildings with IoT and Azure
- Environment monitoring and supply chain optimization with IoT

IoT and data analytics

Azure Cosmos DB | Azure IoT Hub | Azure SQL Database | Azure Table Storage

This example scenario is relevant to organizations building solutions that integrate data from many IoT devices into a comprehensive data analysis architecture to improve and automate decision making. Potential applications include construction, mining, manufacturing, or other industry solutions involving large volumes of data from many IoT-based data inputs.

Architecture



Download a Visio file of this architecture.

Dataflow

The data flows through the solution as follows:

1. Construction equipment collects sensor data and sends the construction results data at regular intervals to load balanced web services hosted on a cluster of Azure virtual machines.

2. The custom web services ingest the construction results data and store it in an Apache Cassandra cluster also running on Azure virtual machines.
3. Another dataset is gathered by IoT sensors on various construction equipment and sent to IoT Hub.
4. Raw data collected is sent directly from IoT Hub to Azure blob storage and is immediately available for viewing and analysis.
5. Data collected via IoT Hub is processed in near real time by an Azure Stream Analytics job and stored in an Azure SQL database.
6. The Smart Construction Cloud web application is available to analysts and end users to view and analyze sensor data and imagery.
7. Batch jobs are initiated on demand by users of the web application. The batch job runs in Apache Spark on HDInsight and analyzes new data stored in the Cassandra cluster.

Components

- IoT Hub acts as a central message hub for secure bi-directional communication with per-device identity between the cloud platform and the construction equipment and other site elements. IoT Hub can rapidly collect data for each device for ingestion into the data analytics pipeline.
- Azure Stream Analytics is an event-processing engine that can analyze high volumes of data streaming from devices and other data sources. It also supports extracting information from data streams to identify patterns and relationships. In this scenario, Stream Analytics ingests and analyzes data from IoT devices and stores the results in Azure SQL Database.
- Azure SQL Database contains the results of analyzed data from IoT devices and meters, which can be viewed by analysts and users via an Azure-based Web application.
- Blob storage stores image data gathered from the IoT hub devices. The image data can be viewed via the web application.
- Traffic Manager controls the distribution of user traffic for service endpoints in different Azure regions.
- Load Balancer distributes data submissions from construction equipment devices across the VM-based web services to provide high availability.
- Azure Virtual Machines host the web services that receive and ingest the construction results data into the Apache Cassandra database.
- Apache Cassandra is a distributed NoSQL database used to store construction data for later processing by Apache Spark.
- Web Apps hosts the end-user web application, which can be used to query and view source data and images. Users can also initiate batch jobs in Apache Spark via

the application.
- Apache Spark on HDInsight supports in-memory processing to boost the performance of big-data analytic applications. In this scenario, Spark is used to run complex algorithms over the data stored in Apache Cassandra.

Alternatives

- Azure Cosmos DB is an alternative NoSQL database technology. Azure Cosmos DB provides multi-master support at global scale with multiple well-defined consistency levels to meet various customer requirements. It also supports the Azure Cosmos DB for Apache Cassandra.
- Azure Databricks is an Apache Spark-based analytics platform optimized for Azure. It is integrated with Azure to provide one-click setup, streamlined workflows, and an interactive collaborative workspace.
- Data Lake Storage is an alternative to Blob storage. For this scenario, Data Lake Storage was not available in the targeted region.
- Web Apps could also be used to host the web services for ingesting construction results data.
- Many technology options are available for real-time message ingestion, data storage, stream processing, storage of analytical data, and analytics and reporting.

Scenario details

In this scenario, a construction equipment manufacturer builds vehicles, meters, and drones that use IoT and GPS technologies to emit telemetry data. The company wants to modernize their data architecture to better monitor operating conditions and equipment health. Replacing the company's legacy solution using on-premises infrastructure would be both time intensive and labor intensive, and would not be able to scale sufficiently to handle the anticipated data volume.

The company wants to build a cloud-based "smart construction" solution. It should gather a comprehensive set of data for a construction site and automate the operation and maintenance of the various elements of the site. The company's goals include:

- Integrating and analyzing all construction site equipment and data to minimize equipment downtime and reduce theft.
- Remotely and automatically controlling construction equipment to mitigate the effects of a labor shortage, ultimately requiring fewer workers and enabling lower-skilled workers to succeed.
- Minimizing the operating costs and labor requirements for the supporting infrastructure, while increasing productivity and safety.

- Easily scaling the infrastructure to support increases in telemetry data.
- Complying with all relevant legal requirements by provisioning resources in-country/region without compromising system availability.
- Using open-source software to maximize the investment in workers' current skills.

Using managed Azure services such as IoT Hub and HDInsight will allow the customer to rapidly build and deploy a comprehensive solution with a lower operating cost. If you have additional data analytics needs, you should review the list of available fully managed data analytics services in Azure.

Potential use cases

Other relevant use cases include:

- Construction (facilities and real-estate), mining (energy), or equipment manufacturing scenarios
- Large-scale collection of device data for storage and analysis
- Ingestion and analysis of large datasets

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 broad availability of Azure regions is an important factor for this scenario. Having more than one Azure region in a single country/region can provide disaster recovery while also enabling compliance with contractual obligations and law enforcement requirements. Azure's high-speed communication between regions is also an important factor in this scenario.

Azure support for open-source technologies allowed the customer to take advantage of their existing workforce skills. The customer can also accelerate the adoption of new technologies with lower costs and operating workloads compared to an on-premises 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.

The following considerations will drive a substantial portion of the costs for this solution.

- Azure virtual machine costs will increase linearly as additional instances are provisioned. Virtual machines that are deallocated will only incur storage costs, and not compute costs. These deallocated machines can then be reallocated when demand is high.
- IoT Hub costs are driven by the number of IoT units provisioned as well as the service tier chosen, which determines the number of messages per day per unit allowed.
- Stream Analytics is priced by the number of streaming units required to process the data into the service.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- Alex Buck | Senior Content Developer

Related resources

Additional IoT architectures:

- Azure IoT reference architecture
- IoT using Azure Cosmos DB
- Retail - Buy online, pickup in store (BOPIS)
- Predictive maintenance with the intelligent IoT Edge
- Secure your IoT SaaS app with the Microsoft identity platform

IoT architecture guides:

- IoT concepts
- Azure IoT Edge Vision
- Azure industrial IoT analytics guidance
- IoT patterns

IoT connected light, power, and internet

Azure Databricks | Azure IoT Hub | Azure Machine Learning | Azure Synapse Analytics | Power BI

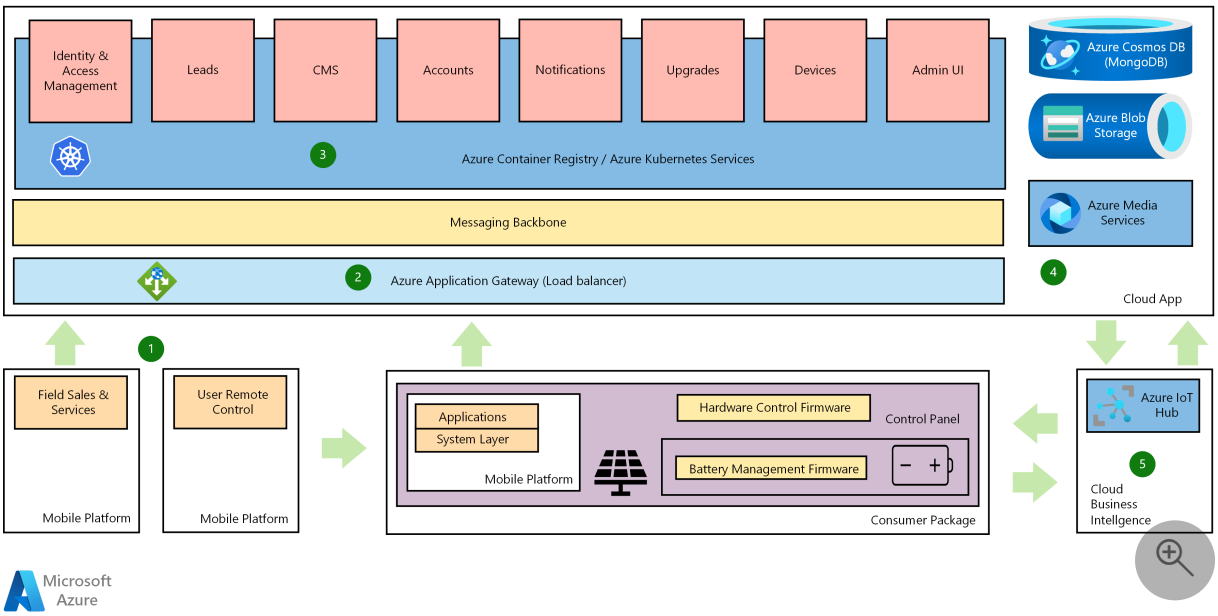
Solution ideas

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 an internet of things (IoT) solution that provides power, light, and internet services to remote locations.

Architecture

Application solution



Download a Visio file of this architecture.

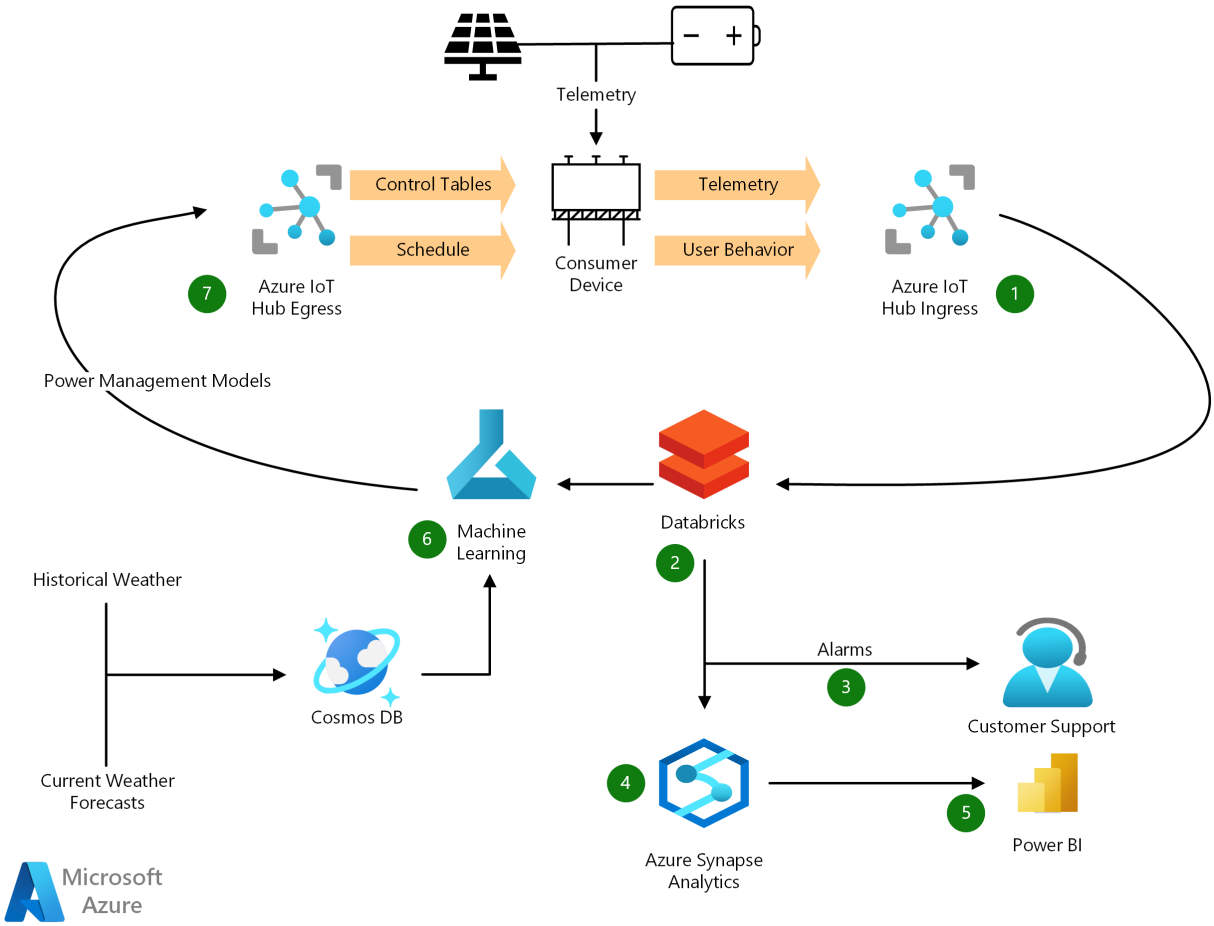
Dataflow

This is a containerized microservices app with end-user interfaces:

1. Field sales and service agents use a mobile platform to interact with the cloud application via Azure Application Gateway. End users use a built-in interface or mobile app to access and control their devices.
2. Application Gateway uses messaging protocols to interact with users and operators.
3. The cloud app consists of containerized microservices that provide features and interfaces like identity and access management, device upgrades, notifications, and commerce services.
4. Depending on the features used, the app accesses Azure services and resources like Azure Blob Storage for unstructured data storage, Azure Cosmos DB for large structured databases, and Azure Media Services for entertainment content.
5. The IoT gateway also sends streaming telemetry and user data to the cloud via Azure IoT Hub, to use for analytics and machine learning (ML).

Analytics and machine learning solution

The business intelligence part of the process includes the following data analysis and control loop:



Download a Visio file of this architecture.

Dataflow

1. IoT Hub receives the streaming telemetry and user data from the IoT devices, and routes events to Azure Databricks via Azure Functions.
2. Azure Databricks extracts, transforms, and loads (ETLs) the event data.
3. Azure Databricks uses Azure Functions to send some events, like alarms, directly to a customer support app for action.
4. Azure Databricks sends the ETL data to Azure Synapse, which performs analytics and stores the data.
5. Power BI reports use the analyzed data and insights. The service provider can use the data for system evaluation and future planning.
6. Azure Machine Learning uses a Databricks cluster to train and retrain ML models for power management. Model retraining combines current data with stored external data, like historical weather and forecasts, on Azure Cosmos DB.
7. The retrained model triggers a pipeline that packages the model and sends the retrained models to IoT Hub. IoT Hub sends the updated models to the IoT devices to use for device management.

Components

- Azure Application Gateway manages and load balances traffic to and from cloud web apps.
- Azure Kubernetes Service (AKS) hosts and simplifies Kubernetes orchestration of Docker containerized apps.
- Azure Container Registry is a managed, private registry service that supports AKS applications at scale.
- Azure IoT Hub is a central cloud message hub for bi-directional communications between IoT applications and devices.
- Azure Databricks is a fast, easy, and collaborative Apache Spark-based analytics service for big data pipelines.
- Azure Synapse Analytics, formerly SQL Data Warehouse, is an analytics service that brings together enterprise data warehousing and big data analytics.
- Power BI is a collection of software services, apps, and connectors that turn data into coherent, immersive, interactive visualizations and reports.
- Azure Machine Learning is a cloud-based ML environment that uses existing data to forecast future behaviors, outcomes, and trends.

Scenario details

A large-scale telecommunications company based the solution on IoT devices that can act as energy and internet hubs for homes and small businesses.

The IoT devices have an IoT gateway that acts as a hub for data transfer and customized service delivery. The devices use a rooftop solar panel to charge a battery, which delivers LED light, USB power, and cellular connectivity. The gateway collects and transmits telemetry data from the solar panel, battery, and output devices. The solution also includes an IoT device with an integrated SIM card and tablet to provide a user interface.

The basic IoT device provides LED light, USB device charging, internet connectivity, and user support through alerts and chatbots. Users can get more services and content on demand or by subscription.

The overall solution combines IoT connected devices with Azure platform-based mobile apps. The solution delivers clean, low-cost power and internet services with high reliability and minimal downtime.

Azure supports two major workstreams in this IoT solution:

- Real-time IoT device telemetry detects transient or long-running anomalies. The system can respond via real-time chatbots and take device actions. For example, in low-power conditions, a device can reduce power usage for background or inactive features. The user continues to get a good experience with the services they're actively using.

- Post-processing data analytics and machine learning evaluate usage and incidents to determine predictive maintenance and future needs. Alerts can notify customers about parts that are predicted to fail soon.

Potential use cases

The following scenarios and industries could use this solution:

- Locations with limited centralized power and internet connectivity.
- News, entertainment, and educational organizations, to deliver content and programming.
- Financial institutions, to provide online commerce and banking services.
- Government and public health agencies, for emergency and support communications.

Next steps

- IoT concepts and Azure IoT Hub

- Introduction to private Docker container registries in Azure
- What is dedicated SQL pool (formerly SQL DW) in Azure Synapse Analytics?
- MLOps: Model management, deployment, lineage, and monitoring with Azure Machine Learning
- Set up a development environment with Azure Databricks and AutoML in Azure Machine Learning
- A solar-powered device will bring online entertainment, education to villages
- Veriown case study

Related resources

- Choose an Internet of Things (IoT) solution in Azure
- Extract actionable insights from IoT data
- Task-based consumer mobile app
- Ingestion, ETL, and stream processing pipelines with Azure Databricks

IoT device connectivity for healthcare facilities

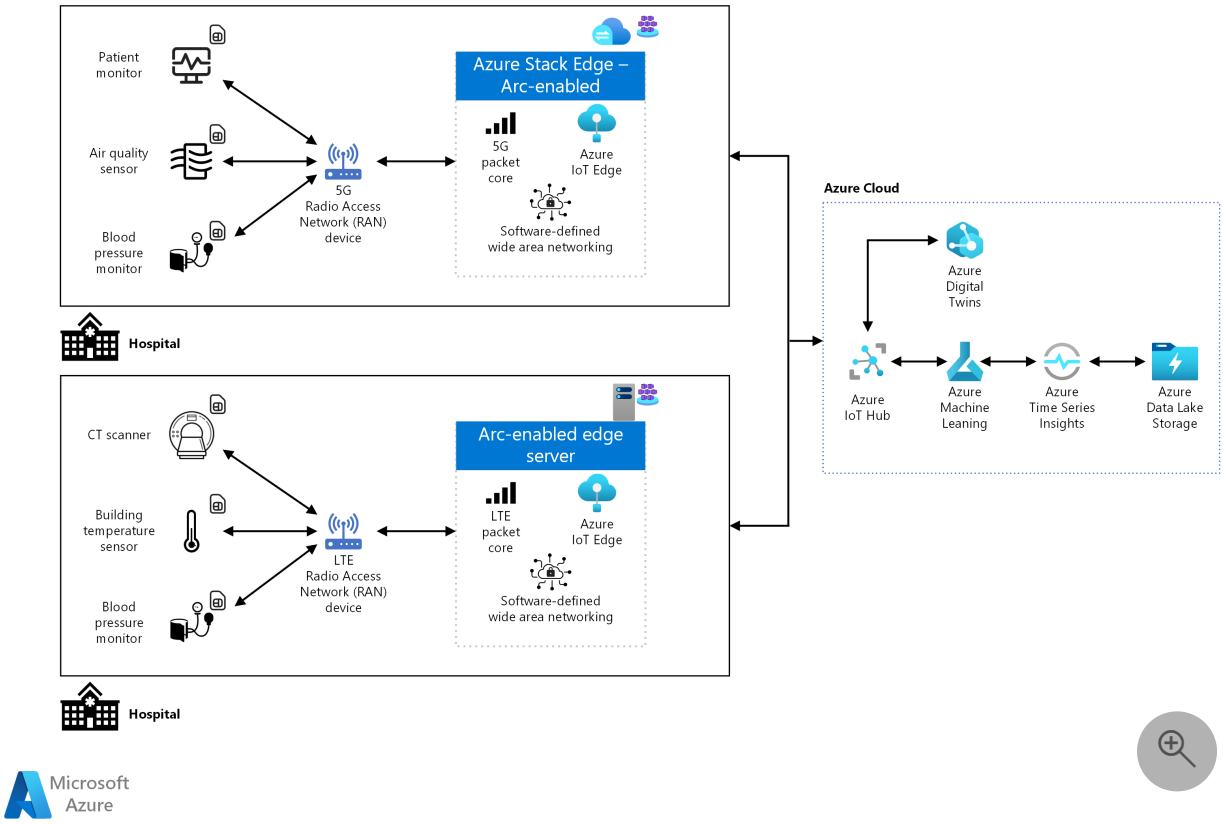
Azure Arc | Azure IoT Edge | Azure IoT Hub | Azure Sphere | Azure Stack Edge

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 describes how buildings and campuses can securely and reliably connect, and scale their on-premises Internet of Things (IoT) devices to the cloud.

Architecture



Download a Visio file of this architecture.

Workflow

1. Hospital buildings use various connected devices to monitor both patient health and facility performance.
    - Health-tracking devices include patient monitors, CT scanners, and blood pressure monitors.
    - Building safety and quality devices include air quality and building temperature sensors.
2. The patient health and building monitoring devices send data to LTE or 5G Radio Access Network (RAN) devices.
3. The 5G or LTE radios in the hospitals forward the data to the 5G or LTE packet cores running on the edge servers. The edge servers can be Azure Stack Edge or any Azure Arc-enabled servers.
4. On the edge servers, the IoT Edge runtime can preprocess data before sending it to Azure for further analysis.
5. In the cloud, Azure IoT Hub ingests data quickly and securely, and sends it to Azure Machine Learning.
6. Azure Machine Learning incorporates the new data to further optimize the model that controls the smart building settings.
7. Data from Azure IoT Hub also feeds into Azure Digital Twins, which provides a map of the hospitals' networked IoT devices as a virtual simulation.
8. Data also feeds into Azure Time Series Insights, which can analyze patient health over a period of time, or treatment efficacy over several hospitals. Time Series Insights also offers a visualization layer to aid in decision-making.
9. All the data is stored in Azure Data Lake Storage, which can store data of any format and size.

Components

This solution uses the following Azure components:

- Azure Stack Edge is a portfolio of devices that bring compute, storage, and intelligence to the IoT Edge. Azure Stack Edge acts as a cloud storage gateway that enables data transfers to Azure, while retaining local access to files.
- Azure Arc-enabled Kubernetes connects Kubernetes clusters running inside or outside of Azure.

- Azure Sphere is a comprehensive IoT security solution that includes hardware, OS, and cloud components for IoT device security.
- Azure IoT Edge deploys cloud intelligence locally on IoT devices.
- Azure IoT Hub is a cloud-based managed service for bidirectional communication between IoT devices and Azure.
- Azure Machine Learning is an integrated data science solution for data scientists and developers to build, train, and deploy machine learning models.
- Azure Digital Twins is an IoT platform that creates digital representations of real-world things, places, processes, and people in the cloud.
- Azure Time Series Insights is an end-to-end IoT analytics platform to monitor, analyze, and visualize industrial IoT analytics data at scale.
- Azure Data Lake Storage is a scalable and secure data lake for high-performance analytics workloads.

Scenario details

Cloud services can store and analyze the IoT data to diagnose anomalies and take corrective or preventive actions. Azure services can further analyze and store the data, and use machine learning to optimize building settings.

Potential use cases

In this solution, a healthcare facility uses LTE or 5G-enabled IoT devices to track both patient health and building performance. The devices use built-in Azure Sphere certified chips to stream data to on-premises edge servers, which communicate with the Azure cloud. On-premises network administrators can view network health through the packet cores on the edge servers.

Other examples of this approach include:

- Predictive maintenance for machines in a coffeehouse.
- Safety and compliance monitoring for perishable food and drink temperatures in a food manufacturing plant.
- Detecting the optimal point for resource extraction in the energy sector, based on data collected by autonomous exploration vehicles.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- Nikhil Ravi | Production Management Leader

Next steps

- Azure Arc on Kubernetes on Azure Stack Edge
- Azure Private MEC

Related resources

- Manage configurations for Azure Arc enabled servers
- Deploy AI and ML computing on-premises and to the edge

UVEN smart and secure disinfection and lighting

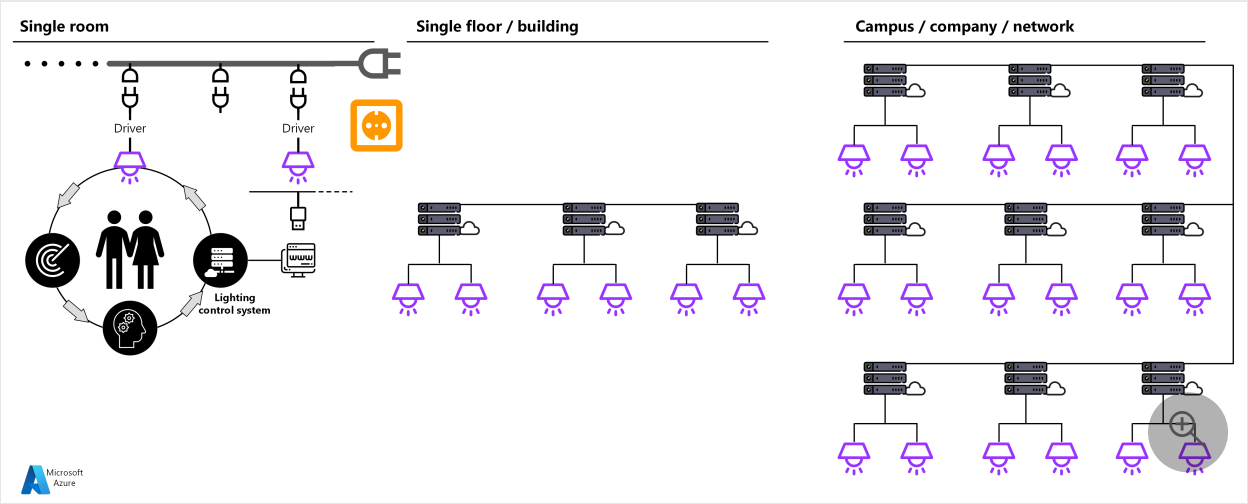
Azure Sphere

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**.

Smart, connected Internet of Things (IoT) devices can make life healthier and safer. For example, research increasingly confirms the importance of high-quality light and natural light cycles for promoting human alertness, health, well-being, and productivity. BrainLit's patented BioCentric Lighting™ (BCL) system is a dynamic, self-learning, IoT-based system that creates high-quality natural lighting for indoor environments.

Architecture



Download a Visio file of this architecture.

Dataflow

- The system uses radar proximity sensors to detect when people are present. Whenever a space is unoccupied, the panels emit UV radiation to heavily reduce or eliminate viruses and bacteria.

- One UVEN lighting control system (LCS) can control both BCL light and UVC radiation emissions for up to 20 luminaires. The LCS connects to radar sensors, electrical power, and to the internet or other LCSs for updates and real-time calibration.
- Several cells can connect in a cluster. Only one of the clustered LCSs needs to connect to the internet. An LCS or cluster can operate standalone, or in an overall UVEN BCL network system with other clusters.
- A UVEN BCL Net is completely decentralized and offers unlimited scalability, control, and flexibility.

Components

The UVEN system consists of light-emitting diode (LED) luminaires that are controlled by Azure Sphere-powered lighting control systems.

UVEN components

The basic UVEN ceiling panel consists of dynamic, adaptable LEDs that integrate BCL with disinfection in the same luminaire. As well as offering a large range of visible and UV emissions, the LEDs provide exceptional color and detail rendering, and the ability to dim smoothly down to 0.1%. LEDs are low-cost, long-lasting, and environmentally sustainable compared to traditional incandescent and fluorescent indoor light sources.

When people are present, the ceiling panels emit BCL light only. In unoccupied spaces, the panels emit UV radiation to heavily reduce or eliminate viruses and bacteria. Separate drivers optimized to operate in their respective regimes maintain full BCL and UV functionality simultaneously.

Radar motion sensors, alarms, and manual overrides provide double fail-safe insurance so users receive beneficial light only. Flexible, scalable, updatable LCSs based on Azure Sphere technology control luminaires, sensors, and safety and alarm mechanisms.

UV radiation can damage surfaces and equipment. UVEN advanced sensors and controllers ensure that the luminaires generate the minimum doses of UVC radiation to ensure disinfection. The system can continually integrate new research developments and public health guidelines via its cloud connection, providing the best scientifically based lighting and disinfection.

UVEN administrative support tools include an admin panel to maintain information about users and devices, and a dashboard to provide an overall view of user activity and exposure.

Azure Sphere

Azure Sphere is a secure application platform for IoT connected devices with built-in cloud communication and security. The platform consists of an Azure Sphere-certified secured MCU chip that runs a custom high-level Linux-based microcontroller operating system (OS). The Azure Sphere OS provides a platform for IoT application development, including both high-level and real-time capable applications. The OS connects to the cloud-based Azure Sphere Security Service (AS3) for continuous, renewable security.

AS3 establishes a secure connection between a device and the internet or cloud, ensures secure boot, and enables maintenance, updates, and control for Azure Sphere-certified chips. AS3 authenticates device identity, ensures the integrity and trust of the system software, and certifies that the device is running a trusted code base. The service also provides a secure channel to automatically download and install OS and customer application updates to deployed devices.

The MediaTek 3620 is the first Azure Sphere-certified chip. The chip includes an ARM Cortex-A7 500-MHz processor, two ARM Cortex-M4F 200-MHz I/O subsystems, Wi-Fi capability, and the Microsoft Pluton security subsystem with a dedicated ARM Cortex-M4F core. The Linux-based Azure Sphere OS lets developers write applications that can run on either the A7 core with access to external communications, or as real-time capable apps on one of the M4 processors. Developers can distribute applications to Azure Sphere devices through the same secure mechanism as the Azure Sphere OS updates.

Azure Sphere *greenfield* implementations involve designing and building new IoT devices with Azure Sphere-certified chips. *Brownfield* implementation allows existing IoT devices to securely connect to the internet through an Azure Sphere Guardian module. The guardian module contains an Azure Sphere certified chip, the Azure Sphere OS, and AS3. The module connects to the IoT device through an existing peripheral, and then securely connects to the internet. The IoT device itself isn't exposed to the internet.

Scenario details

The BCL system can also deliver radiant energy in the non-visible ultraviolet (UV) spectrum to kill viruses. All UV radiation has antimicrobial capabilities. The shortest wavelength, 200-300 nm UVC radiation, causes DNA inactivation and prevents viral replication. UVC radiation disinfection is an important weapon in the fight against COVID. BrainLit's new UVEN concept combines BCL light control with UVC disinfection to promote health and well-being and help kill viruses.

Other UVC virus-killing technology that uses plug-in devices, robots, drones, and wands can't provide overall space coverage or performance and effectiveness data. To operate these devices safely and efficiently disrupts business and personnel, resulting in loss of productivity and lower space utilization. UVEN disinfection operates in unoccupied spaces during non-use times, providing safe, comprehensive, autonomous microbe deactivation without business disruption.

Because IoT devices work directly on the physical environment and may use and collect sensitive data, device safety and security are paramount. UVEN double fail-safe features ensure occupants receive beneficial light only. The Azure Sphere standalone microprocessor unit (MCU)-based platform securely runs IoT apps and connects directly to the cloud for complete Azure-based security and the latest OS and app updates.

The system can continually integrate new research developments and public health updates via its cloud connection, ensuring an up-to-date and scientifically based lighting and disinfection system.

UVEN combined BCL and UVC technology offers:

- A smart, safe, and secure integrated system solution for lighting and disinfection, with optimal usage of floor areas.
- Long-term solutions that can adapt lighting and disinfection doses and recipes to changing circumstances.
- Overall better health and improved immune systems for occupants through optimized lighting and a virus-free environment.

Potential use cases

This solution is ideal for the healthcare and facilities industries. We recommend using the solution in the following scenarios:

- UVEN cells can fit any layout of a specific area, like a reception counter, or split between several areas, like a number of washrooms.
- The modular system can scale to fit any space, from individual rooms and homes to large campuses and public areas.
- Azure Sphere-based infrastructure can easily integrate with existing IoT devices or building management systems.

Next steps

- For more information, please contact iotcovidsupport@microsoft.com.
- For more information about the BCL system, see the BrainLit website.

- For more information about Azure Sphere, see the Azure Sphere documentation.
- For Azure Sphere code samples, see Azure Sphere Samples.

Related resources

- Azure IoT reference architecture
- Cognizant Safe Buildings with IoT and Azure
- COVID-19 safe environments with IoT Edge monitoring and alerting
- Create smart places by using Azure Digital Twins

Create smart places by using Azure Digital Twins

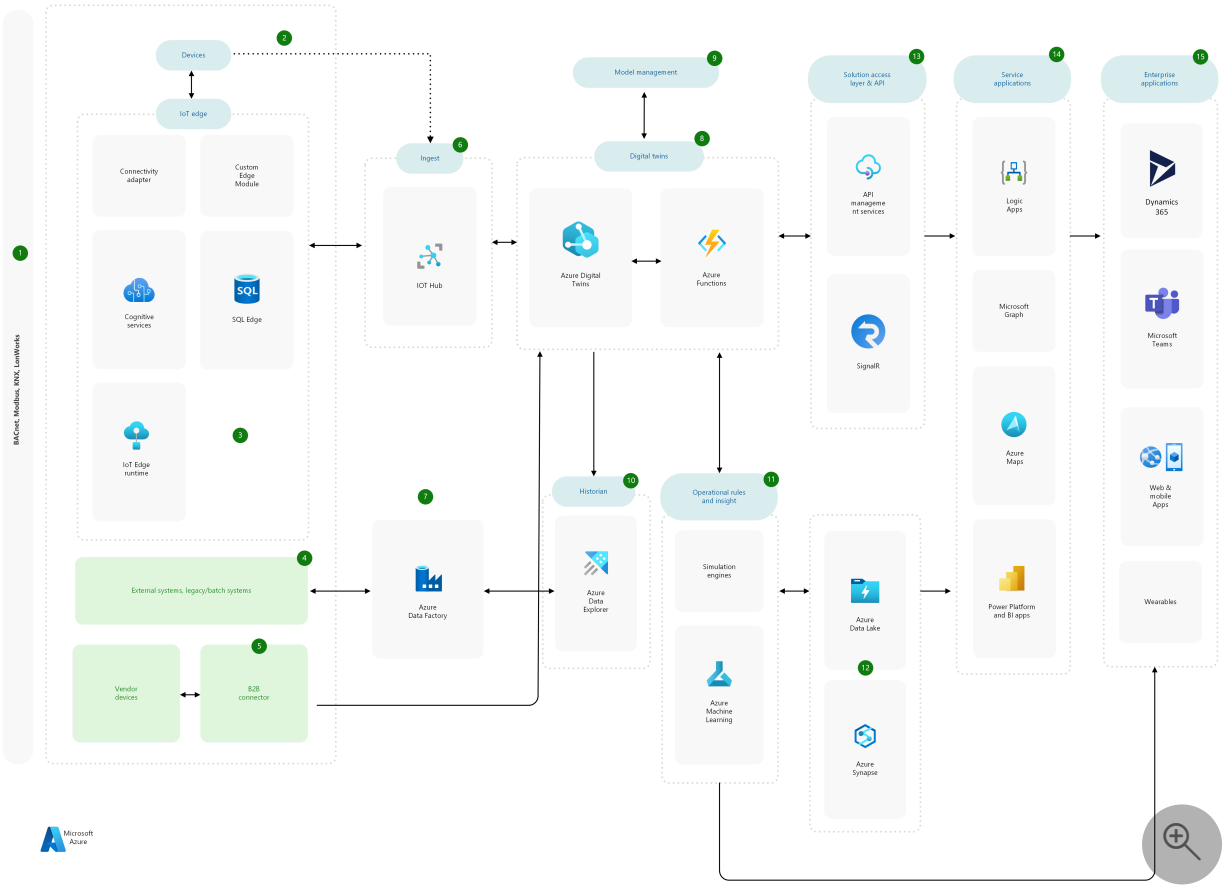
Azure Data Explorer | Azure Data Factory | Azure Digital Twins | Azure Functions | Azure IoT Hub

This article outlines a solution for smart spaces. Azure Digital Twins forms the core of the architecture by modeling the environment. Azure IoT Hub, which is a managed IoT service, also plays a significant role, as does the analytics service Azure Data Explorer.

Architecture

The following diagram shows the flow of data in this solution:

- The boxes that contain multiple icons represent categories of services. Within each category, services work independently or together to provide functionality.
- Arrows between boxes represent communication between the corresponding areas.



Download a Visio file of this architecture.

Dataflow

1. The environment can use these and other communication protocols:

    - Building Automation Controls network (BACnet)
    - Modbus
    - KNX
    - LonWorks

2. On-premises devices and systems send telemetry and other data to the cloud. Data sources include:

    - Brownfield devices
    - Direct-connect sensors
    - Sensors that independent software vendors (ISVs) provide
    - Existing business systems

3. Devices, sensors, and actuators generate telemetry. Some devices interact directly with IoT Hub. Other devices send data to IoT Hub through Azure IoT Edge.

4. External, batch, or legacy systems send data to Azure Data Factory. This static data typically originates in files and databases.

5. Business-to-business connectors translate vendor data and stream it to Azure Digital Twins.

6. IoT Hub ingests device telemetry. IoT Hub also provides these services:

    - Device-level security
    - Device provisioning services
    - Device twins
    - Command and control services
    - Scale-out capabilities

7. Data Factory transforms semi-static data and transfers it to Azure Data Explorer or to long-term storage.

8. Azure Functions receives the IoT Hub data and uses Azure Digital Twins APIs to update Azure Digital Twins. Azure Digital Twins holds the spatial graph of the buildings and environment. Azure Digital Twins models the environment with Digital Twins Definition Language (DTDL). Azure Functions processes the data, performing fault detection and graph updates.

9. Various components create, store, and load DTDL models.

10. Azure Digital Twins sends the data through Azure Event Grid to Azure Data Explorer. This analytics service functions as a historian by storing the solution's time series data.

11. Simulation engines and AI tools process the data. Examples include Azure Cognitive Services, AI models, and partner simulation services.

12. Azure Data Lake provides long-term storage for the data. Azure Synapse Analytics analyzes and reports on the data.

13. For visualization tools and enterprise apps, the solution access layer provides secure access to core system services:

    - Azure API Management offers functionality for normalizing, securing, and customizing APIs. This platform also enforces usage quotas and rate limits.

    - Azure SignalR Service sends notifications to UIs when telemetry and data changes.

    - For applications that exchange data asynchronously or at volume, various components provide publishing and subscribing mechanisms:
      - IoT Hub
      - Azure Service Bus queues
      - Azure Event Hubs
      - Web hooks

14. Service applications collect data from the access control API layer. These applications then analyze and prepare the data for end-user applications. Microsoft tools like Power Apps, Power BI, and Azure Maps create reports and insights on data in the Azure data stores.

15. Enterprise applications use the prepared data. Examples include:

    - Dynamics 365 modules.

    - ISV solutions.

    - Microsoft Teams apps.

    - Field-optimized solutions such as mobile apps and wearables:
      - HoloLens
      - RealWear HMT

Components

The solution uses these components:

Core components

- IoT Hub connects devices to Azure cloud resources. This managed service provides:
  - Device-level security.
  - Device provisioning services.
  - Device twins.
  - Command and control services.
  - Scale-out capabilities.

- Azure IoT SDKs provide the recommended way for devices to connect to IoT Hub. Devices that can use these kits include:
  - Azure Sphere devices.
  - Devices that run Azure RTOS.

- IoT Edge runs cloud workloads on IoT Edge devices. Specifically, this central message hub can run real-time analytics through Azure Machine Learning and Azure Stream Analytics. IoT Edge also functions as a gateway to IoT Hub for:
  - Devices with low-power requirements.
  - Legacy devices.
  - Constrained devices.

- Data Factory is an integration service that works with potentially large blocks of data from disparate data stores. You can use this platform to orchestrate and automate data transformation workflows. For instance, Data Factory can bridge the gap between semi-static stores and historian components like Azure Data Explorer.

- Business-to-business connectors translate and stream data bidirectionally between vendor components and Azure Digital Twins. A growing number of vendors use DTDL to create industry-standard models. RealEstateCore provides an example. As a result, these integrations are expected to become simpler over time.

- Azure Digital Twins stores digital representations of IoT devices and environments. You can use this data for data propagation or real-time analysis. Internally, Azure Digital Twins:
  - Models environments with DTDL.
  - Offers a REST API for entering data.
  - Provides SDKs that support control and data plane operations for various languages.

You can build ontologies, or pre-existing model sets, by using DTDL. You can also start with an industry-supported model:
  - RealEstateCore ontology
  - Smart Cities ontology
  - Energy Grid ontology

- Azure Digital Twins Explorer is a developer tool that you can use to visualize and interact with Azure Digital Twins data, models, and graphs. This tool is currently in public preview.

- Model management components maintain the DTDL model:

  - For model creation, these options are available:
    - Azure Digital Twins Explorer
    - ISV solutions
    - Custom-built tools
    - Text or code editors

  - Repositories store ontologies:
    - GitHub stores RealEstateCore, the Smart Cities ontology, and the Energy Grid ontology.
    - For custom ontologies, customized repositories and solution-specific repos in GitHub are available.

  - For loading models into Azure Digital Twins, these options exist:
    - UploadModels, a tool for uploading DTDL ontologies
    - Samples in the Azure Digital Twins tools repository

- Azure Functions is an event-driven serverless compute platform. With Functions, you can use triggers and bindings to integrate services at scale.

- Azure Data Explorer is a fast, fully managed data analytics service. You can use this service for real-time analysis on large volumes of data. Azure Data Explorer can handle diverse data streams from applications, websites, IoT devices, and other sources.

- Azure Cognitive Services provides AI functionality. These services offer a set of pre-trained, neural network models for the cloud. The REST APIs and client library SDKs can help you build cognitive intelligence into apps. You can use Cognitive Services functionality:
  - In near real time.
  - At certain data thresholds.
  - On demand.

    - For complex jobs with long processing times.

- Azure Machine Learning is a cloud-based environment that helps you build, deploy, and manage predictive analytics solutions. With these models, you can forecast behavior, outcomes, and trends.

- Azure Data Lake stores a large amount of data in its native, raw format. The data typically comes from multiple, heterogeneous sources and may be structured, semi-structured, or unstructured.

- Azure Synapse Analytics is an analytics service for data warehouses and big data systems. This service integrates with Power BI, Machine Learning, and other Azure services.

- 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 logs call metadata and enforces usage quotas and rate limits.

- Azure Service Bus is a fully managed enterprise message broker. Service Bus supports message queues and publish-subscribe topics.

- Azure Event Hubs is a fully managed streaming platform for big data.

- Azure SignalR Service is an open-source software library that provides a way to send notifications to web apps in real time.

Service applications

- Azure Logic Apps automates workflows by connecting apps and data across clouds.

- Azure Maps offers geospatial APIs for adding maps, spatial analytics, and mobility solutions to apps.

- Microsoft Graph provides tools for accessing data in Microsoft 365, Windows 10, and Enterprise Mobility + Security.

- Power Platform is a collection of products and services that provide low-code tools for creating efficient and flexible solutions:
    - Power Apps is a suite of apps, services, connectors, and a data platform. You can use Power Apps to transform manual business operations into digital, automated processes.

- Power BI is a collection of software services and apps that display analytics information.
- Power Automate streamlines repetitive tasks and paperless processes.
- Power Virtual Agents provides no-code chatbots to meet customer and employee needs at scale.

Enterprise applications

- Dynamics 365 is a portfolio of applications for managing business operations.
- Microsoft Teams provides services for meeting, messaging, calling, and collaborating.
- Azure App Service and its Web Apps feature provide a framework for building, deploying, and scaling web apps.

Shared support components

These services provide support for components in all areas of the solution:

- Azure Monitor collects and analyzes app telemetry, such as performance metrics and activity logs. This service notifies apps and personnel about irregular conditions.
- Microsoft Defender for IoT is a unified security service that protects IoT systems by identifying vulnerabilities and threats.
- Azure DevOps Services provides services, tools, and environments for managing coding projects and deployments.
- Microsoft Entra ID is a cloud-based identity service that controls access to Azure and other cloud apps, including ISV solutions and on-premises solutions.
- Azure Key Vault securely stores and controls access to a system's secrets, such as API keys, passwords, certificates, and cryptographic keys.

Alternatives

- Azure Cosmos DB is another option for data storage. This fully managed NoSQL database service scales easily. Azure Cosmos DB offers various ways to access data, including:
  - Document databases.
  - Graph databases.

    - SQL-style queries.
    - An Azure Cosmos DB for Apache Cassandra.

    Azure Synapse Link for Azure Cosmos DB provides a way to run analytics on Azure Cosmos DB data by using Azure Synapse Analytics. As a result, you can combine various data services in solutions that use Azure Cosmos DB.

- Event Hubs can also provide an ingestion service that's scalable and secure. Unlike IoT Hub, which supports bidirectional communication with devices, Event Hubs supports one-way traffic. As a result, you can't use Event Hubs to send commands and policies back to devices. Event Hubs also doesn't offer device-level security. But Event Hubs is appropriate for environments with a high volume of messages from a low number of input devices.

Solution details

Smart places are physical environments that bring together connected devices and data sources. By using these environments, you can see and control:

- Products and systems.
- Interior and exterior spaces.
- Personal experiences with surroundings.

Smart places can include buildings, college campuses, corporate campuses, stadiums, and cities. These environments provide value by helping property owners, facility managers, and occupants operate and maintain sites. Smart places also make spaces more efficient, cost effective, comfortable, and productive.

Smart spaces digitally model spaces and compile relevant data. From that data, you can derive insights on how people, places, and devices are connected.

Potential use cases

This solution applies to many areas:

- Smart campuses (education industry)
- Facilities management (real estate)
- Smart stadiums (sports industry)
- Smart offices
- Energy optimization

Business outcomes

In this example solution, a large commercial real estate owner is digitally transforming an office property. This improvement combines legacy facilities-management data with new features and technologies including:

- Occupancy sensing.
- Cafe queue optimization.
- Parking.
- Shuttle services.

This effort requires integrating brownfield devices and modern Internet of Things (IoT) devices that monitor the physical space. The brownfield devices communicate through common building transports such as BACnet and Modbus.

The company's goals include:

- Optimizing energy usage by diagnosing faults and streamlining field service management. This optimization integrates the existing building management system with devices.
- Deriving new spatial insights and offering innovative occupant experiences by connecting modern devices.
- Developing a cohesive digital model of the environment by bringing together multiple sources of data. The model should expand data analysis opportunities.
- Creating a scalable solution that can collect and archive millions of data points.
- Building a solution that can easily add partner solutions. The solution should also incorporate partner data into the environment's digital twin.

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 solution.

Reliability

Reliability ensures your application can meet the commitments you make to your customers. For more information, see Overview of the reliability pillar.

Scalability

Solutions for smart places solutions can be relatively simple, low-volume implementations. They can also be sophisticated implementations that handle a high volume of data. A solution that aggregates heating, ventilation, and air conditioning (HVAC) telemetry across a large campus is an example of a high-volume implementation.

The core Azure services in this solution are scalable by design and well suited for complex solutions. But when you combine these services, ensure that they don't create choke points. Early in the development cycle, run performance tests at scheduled intervals to identify potential problems.

Flexibility

Design your smart space to be well integrated but also flexible. Smart places use cases are rapidly evolving. At some point after you deploy your solution, you'll need to add new sensors, data types, AI functionality, and visualization techniques. To increase flexibility:

- Choose a loosely coupled solution like the proposed architecture.
- Use industry standards for data ontology. This approach helps reduce the time needed to add new functionality and integrate new software.
- Use API Management. This platform provides a way to create multiple API styles and signatures for a single underlying API.

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.

Legacy building solutions often rely on a lack of external connectivity as their primary source of security. But even data that doesn't identify people can provide information about a business or the people in a building. For instance, organizations use cameras to count people, track assets, and provide security data.

Be careful where you process and save images. Ensure that you address all customer requirements, including privacy issues. Make security a priority throughout the data life cycle of your smart space solution. Specifically, be aware of what data you collect, where you process and store it, and what conclusions you draw from it.

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 an IoT solution.

- For other cost considerations, see Principles of cost optimization in the Microsoft Azure Well-Architected Framework documentation.
- For a discussion about optimizing the cost of services that IoT solutions commonly use, see Azure IoT Reference Architecture.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- Mark Kottke | Senior Architect

Other contributor:

- Matthew Cosner | Principal Software Engineering Manager

Next steps

- Powering Microsoft smart buildings with Microsoft Azure Digital Twins
- Develop with Azure Digital Twins
- Brookfield sets a new standard for innovation in real estate with WillowTwin and Azure Digital Twins
- Global sustainability leader targets new heights of carbon neutrality with Azure Digital Twins

Related resources

- Get started with Azure IoT solutions
- IoT solutions conceptual overview
- Vision with Azure IoT Edge
- Azure Industrial IoT analytics guidance
- Choose an Internet of Things (IoT) solution in Azure
- End-to-end manufacturing using computer vision on the edge
- COVID-19 safe environments with IoT Edge monitoring and alerting

- IoT analytics with Azure Data Explorer
- Cognizant Safe Buildings with IoT and Azure

Solutions for the aerospace industry

Article • 06/30/2023

Intelligent cloud analytics, AI and machine learning, speech capabilities, and improved security are just some of the benefits that aerospace enterprises gain by using Azure.

Airbus is a leader in designing, manufacturing, and servicing commercial and military aircraft, helicopters, satellites, and launch vehicles. Watch this video to learn how they use AI to improve pilot training and satellite operations.

Airbus also uses Azure to fine-tune processes for designing, building, and operating complex products. Watch How Airbus is transforming aerospace with Microsoft Azure Stack to learn more.

In the following video, you can learn about the Microsoft partnership with SpaceX that brings worldwide satellite connectivity to cloud services:

https://www.youtube-nocookie.com/embed/OWYYo2VGkHQ

Architectures for aerospace

The following articles provide detailed analysis of architectures created and recommended for the aerospace industry.

Architecture	Summary	Technology focus
Advanced Azure Kubernetes Service (AKS) microservices architecture	Learn about a scalable AKS microservices architecture. An example scenario describes a company that uses a drone-based delivery service.	Containers
Deploy microservices with Azure Container Apps	Deploy existing microservice applications by using Azure Container Apps. View a drone-based delivery service example.	Containers
Design a microservices architecture	Design and build a microservices architecture on Azure by using a reference implementation that illustrates best practices. View a drone-based delivery service example.	Microservices
End-to-end geospatial storage, analysis, and visualization	This article provides a high-level approach to use cloud-native capabilities, along with some open-source software options and commercial options.	Databases

Architecture	Summary	Technology focus
Ingest FAA SWIM content to analyze flight data	Integrate Chef Infra, Chef InSpec, Test Kitchen, Terraform, Terraform Cloud, and GitHub Actions to automate and create data analytics environments.	Analytics
Organize spaceborne geospatial data with SpatioTemporal Asset Catalog (STAC)	This reference architecture shows an end-to-end implementation of SpatioTemporal Asset Catalog (STAC) to structure geospatial data.	Databases
Process Synthetic Aperture Radar (SAR) data in Azure	View a reference architecture that enables processing SAR/Remote Sensing data on Azure by using Apache Spark on Azure Synapse.	Analytics
Serverless web application	Review a reference architecture that shows a serverless web application that serves static content from Azure Blob Storage. An example scenario describes a company that uses a drone-based delivery service.	Serverless
Spaceborne data analysis with Azure Synapse Analytics	View a reference architecture that enables geospatial workloads on Azure by using Azure Synapse.	Analytics

Solution ideas for aerospace

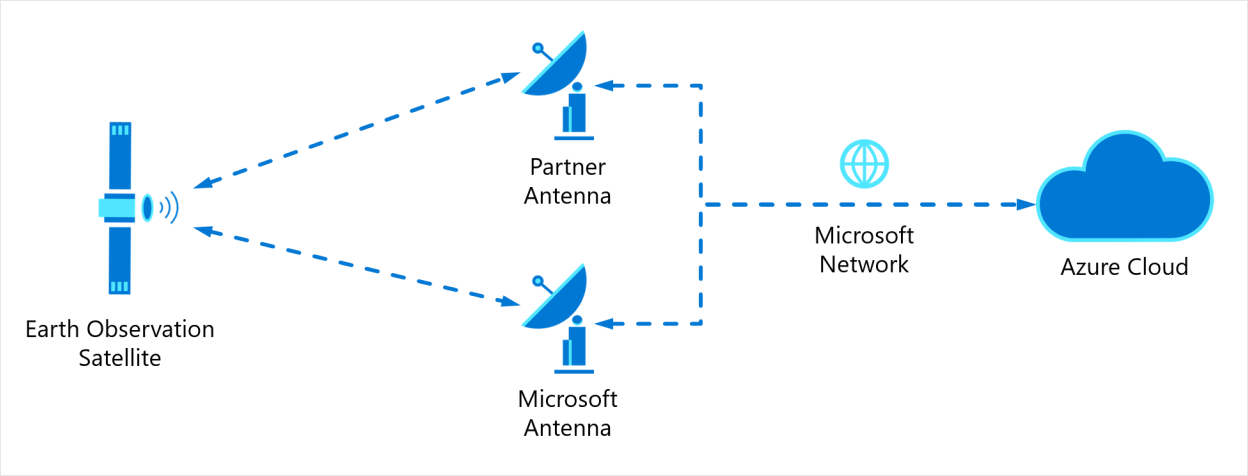
Following are some other ideas that you can use as a starting point for your aerospace solution.

- Data cache
- Predictive aircraft engine monitoring
- Predictive maintenance
- Vision classifier model with Azure Custom Vision Cognitive Service

Azure Orbital Ground Station overview

Article • 11/02/2023

With Azure Orbital Ground Station, your space data is delivered with near-zero latency to your Azure region over the secure and highly available Microsoft network. Azure Orbital Ground Station supports both Microsoft and industry leading Partner ground station networks, ensuring access to the best sites and networks to support your space missions. Deploying and operating a large, globally distributed ground station solution for your space mission can now be done with the reliability and flexibility of the cloud—at any classification level.



Product highlights

- Self-service scheduling of spacecraft contacts to ingest data, monitor satellite health and status, or transmit commands to satellites.
- The managed data path provides one-click access to a global network of ground stations and direct data ingestion into your private Azure virtual network.
- Take advantage of integrated software modems from Kratos for X and S bands, or leverage virtual RF and GNU radio for unrestricted modem implementations.
- Avoid building and managing ground station infrastructure and instead pay-as-you-go with any antenna in the global Azure Orbital network.
- Space data are transmitted and managed securely, according to stringent compliance requirements.
- Integrate command and control software with the Azure Orbital Ground Station API to manage fleet operations.

Links to learn more

- Overview, features, security, and FAQ

- Pricing and SLA
- Microsoft Learn training session
- Azure Space tech blog
- General Availability press announcement

Next steps

- About Microsoft and Partner ground stations
- Get started with Azure Orbital Ground Station
- Support all mission phases

Ground station contact resource

Article • 12/04/2023

To establish connectivity with your spacecraft, schedule and execute a contact on a ground station. A contact, sometimes called a ground station 'pass,' can only occur when the spacecraft passes over a specified ground station while orbiting. You can find available contact opportunities and schedule contacts for your spacecraft through the Azure Orbital Ground Station API or Azure portal.

Contacts are scheduled for a particular combination of a spacecraft and contact profile. When you schedule a contact for a spacecraft, a contact resource is created under your spacecraft resource in your Azure resource group. The contact is only associated with that particular spacecraft and can't be transferred to another spacecraft, resource group, or region.

Contact parameters

The contact resource contains the start time and end time of the pass and other parameters related to pass operations. The full list is below.

Expand table

Parameter	Description
Reservation Start Time	Start time of pass in UTC.
Reservation End Time	End time of pass in UTC.
Maximum Elevation Degrees	The maximum elevation the spacecraft will be in the sky relative to the horizon in degrees. This is used to gauge the quality of the pass.
TX Start Time	Start time of permissible transmission window in UTC. This start time will be equal to or come after Reservation Start Time.
TX End Time	End time of permissible transmission window in UTC. This end time will be equal to or come before Reservation End Time.
RX Start Time	Start time of permissible reception window in UTC. This start time will be equal to or come after Reservation Start Time.
RX End Time	End time of permissible reception window in UTC. This end time will be equal to or come before Reservation End Time.

Parameter	Description
Start Azimuth	Starting azimuth position of the spacecraft measured clockwise from North in degrees.
End Azimuth	End azimuth position of the spacecraft measured clockwise from North in degrees.
Start Elevation	Starting elevation position of the spacecraft measured from the horizon up in degrees.
End Elevation	Starting elevation position of the spacecraft measured from the horizon up in degrees.

The RX and TX start/end times might differ depending on the individual station masks. Billing meters are engaged between the Reservation Start Time and Reservation End Time.

Schedule a contact

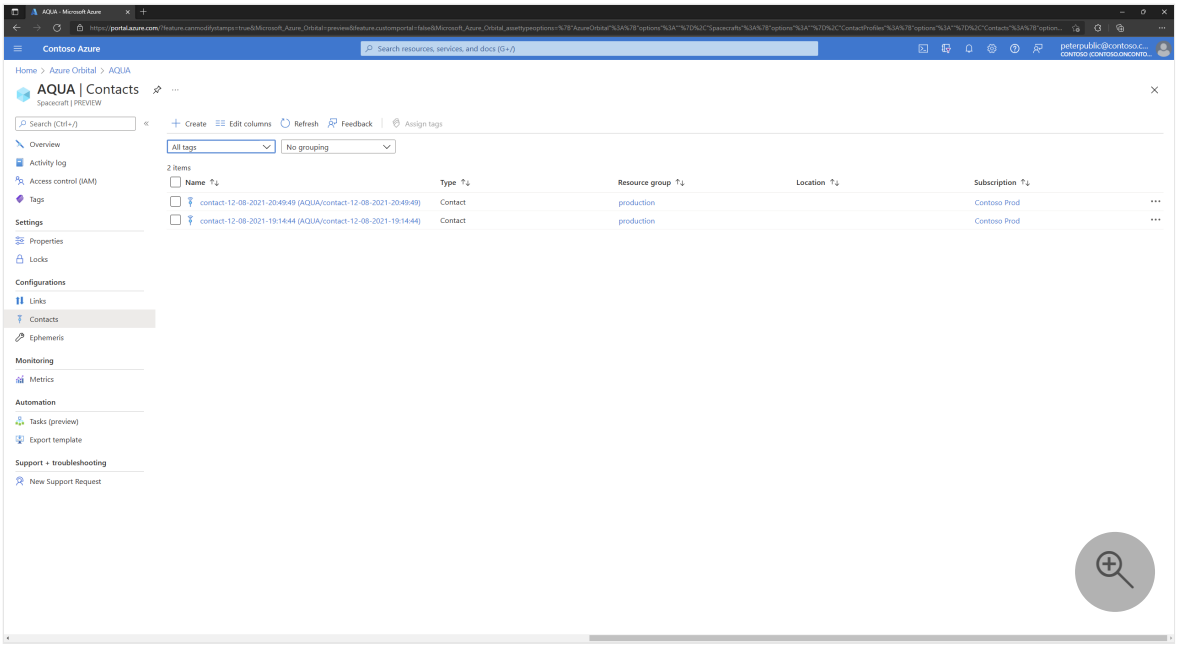
Use the Azure portal or Azure Orbital Ground Station API to create a contact resource for your spacecraft resource.

Cancel a scheduled contact

In order to cancel a scheduled contact, you must delete the contact resource.

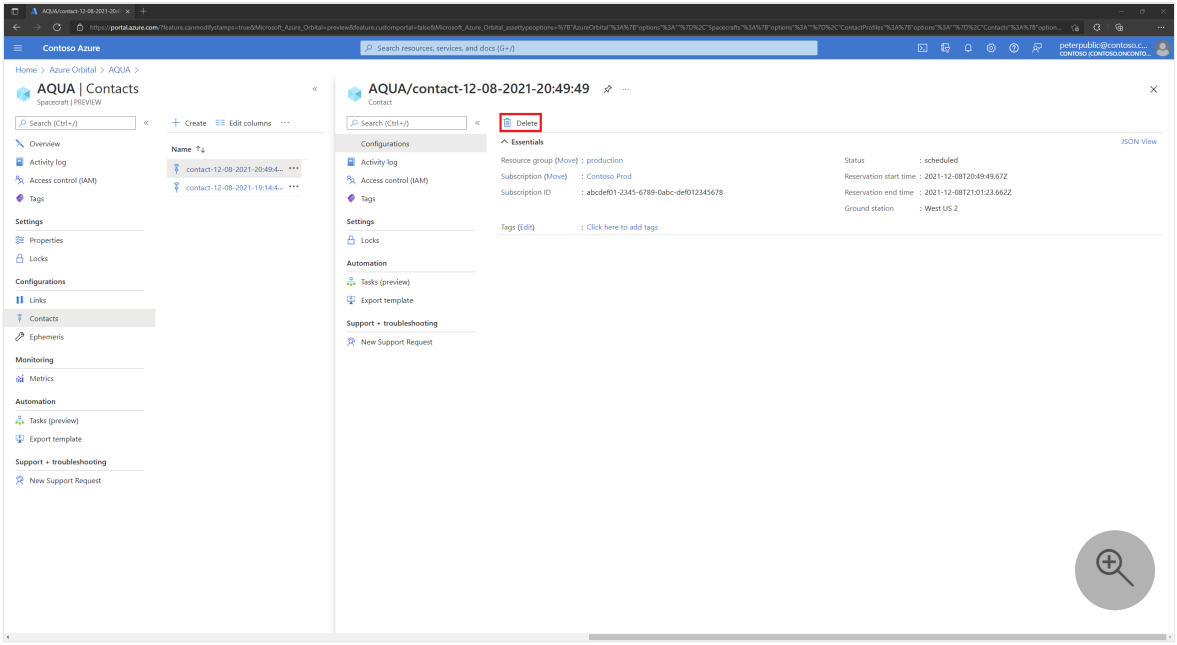
To delete a contact resource via the Azure portal:

1. In the Azure portal search box, enter **Spacecraft**. Select **Spacecraft** in the search results.
2. In the **Spacecraft** page, click the spacecraft associated with the scheduled contact.
3. Click **Contacts** from the left menu bar in the spacecraft's overview page.



4. Click the contact to be deleted.

5. Click **Delete** from the top bar of the contact's configuration view.



6. The scheduled contact will be canceled once the contact entry is deleted.

Alternatively, use the Contacts REST Operation Group to delete a contact with the Azure Orbital Ground Station API.

Next steps

- Schedule a contact
- Update the spacecraft TLE

Ground station contact profile resource

Article • 12/04/2023

The contact profile resource stores pass requirements such as links and endpoint details. Use this resource along with the spacecraft resource during contact scheduling to view and schedule available passes.

You can create many contact profiles to represent different types of passes depending on your mission operations. For example, you can create a contact profile for a command and control pass or a contact profile for a downlink-only pass. These resources are mutable and don't undergo an authorization process like spacecraft resources do. One contact profile can be used with many spacecraft resources.

Understand links and channels

A whole band, unique in direction and polarity, is called a link. Channels, which are children under links, specify the center frequency, bandwidth, and endpoints. Typically there's only one channel per link, but some applications require multiple channels per link.

You can specify EIRP and G/T requirements for each link. EIRP applies to uplinks and G/T applies to downlinks. You can provide a name for each link and channel to keep track of these properties. Each channel has a modem associated with it. Follow the steps in how to set up a software modem to understand the options.

Contact profile parameters

Expand table

Parameter	Description
Pass parameters	
Minimum viable contact duration	The minimum duration of a contact in ISO 8601 format. Acts as a prerequisite to show available time slots to communicate with your spacecraft. If an available time window is less than this time, it won't be in the list of available options. Avoid changing on a pass-by-pass basis and instead create multiple contact profiles if you require flexibility.
Minimum elevation	The minimum elevation of a contact, after acquisition of signal (AOS), in decimal degrees. Acts as a prerequisite to show available time slots to communicate with your spacecraft. Using a higher value might reduce the

Parameter	Description
	duration of the contact. Avoid changing on a pass-by-pass basis and instead create multiple contact profiles if you require flexibility.
Auto track configuration	The frequency band to be used for autotracking during the contact (X band, S band, or Disabled).
Event Hubs Namespace and Instance	The Event Hubs namespace/instance to send telemetry data of your contacts.
Network Configuration	
Virtual Network	The virtual network used for a contact. This VNET must be in the same region as the contact profile.
Subnet	The subnet used for a contact. This subnet must be within the above VNET, be delegated to the Microsoft.Orbital service, and have a minimum address prefix of size /24.
Third-party configuration	Mission configuration and provider name associated with a partner ground network.
Links	
Direction	Direction of the link (uplink or downlink).
Gain/Temperature	Required gain to noise temperature in dB/K.
EIRP in dBW	Required effective isotropic radiated power in dBW.
Polarization	Link polarization (RHCP, LHCP, Dual, or Linear Vertical).
Channels	
Center Frequency	The channel center frequency in MHz.
Bandwidth	The channel bandwidth in MHz.
Endpoint	The name, IP address, port, and protocol of the data delivery endpoint.
Demodulation Configuration	Copy of the modem configuration file such as Kratos QRadio or Kratos QuantumRx. Only valid for downlink directions. If provided, the modem connects to the customer endpoint and sends demodulated data instead of a VITA.49 stream.
Modulation Configuration	Copy of the modem configuration file such as Kratos QRadio. Only valid for uplink directions. If provided, the modem connects to the customer endpoint and accepts commands from the customer instead of a VITA.49 stream.

Example of dual-polarization downlink contact profile

Refer to the example below to understand how to specify an RHCP channel and an LHCP channel if your mission requires dual-polarization on downlink. To find this information about your contact profile, navigate to the contact profile resource overview in Azure portal and click 'JSON view.'

JSON

```
{
  "location": "eastus2",
  "tags": null,
  "id": "/subscriptions/c1be1141-a7c9-4aac-9608-
3c2e2f1152c3/resourceGroups/contoso-
Rgp/providers/Microsoft.Orbital/contactProfiles/CONTOSO-CP",
  "name": "CONTOSO-CP",
  "type": "Microsoft.Orbital/contactProfiles",
  "properties": {
    "provisioningState": "Succeeded",
    "minimumViableContactDuration": "PT1M",
    "minimumElevationDegrees": 5,
    "autoTrackingConfiguration": "disabled",
    "eventHubUri": "/subscriptions/c1be1141-a7c9-4aac-9608-
3c2e2f1152c3/resourceGroups/contoso-
Rgp/providers/Microsoft.EventHub/namespaces/contosoHub/eventhubs/contosoHub"
,
    "networkConfiguration": {
      "subnetId": "/subscriptions/c1be1141-a7c9-4aac-9608-
3c2e2f1152c3/resourceGroups/contoso-
Rgp/providers/Microsoft.Network/virtualNetworks/contoso-
vnet/subnets/orbital-delegated-subnet"
    },
    "links": [
      {
        "name": "contoso-downlink-rhcp",
        "polarization": "RHCP",
        "direction": "downlink",
        "gainOverTemperature": 25,
        "eirpdBW": 0,
        "channels": [
          {
            "name": "contoso-downlink-channel-rhcp",
            "centerFrequencyMHz": 8160,
            "bandwidthMHz": 15,
            "endPoint": {
              "ipAddress": "10.1.0.5",
              "endPointName": "ContosoTest_Downlink_RHCP",
              "port": "51103",
              "protocol": "UDP"
            },
```

```
            "modulationConfiguration": null,
            "demodulationConfiguration": null,
            "encodingConfiguration": null,
            "decodingConfiguration": null
          }
        ]
      }
      {
        "name": "contoso-downlink-lhcp",
        "polarization": "LHCP",
        "direction": "downlink",
        "gainOverTemperature": 25,
        "eirpdBW": 0,
        "channels": [
          {
            "name": "contoso-downlink-channel-lhcp",
            "centerFrequencyMHz": 8160,
            "bandwidthMHz": 15,
            "endPoint": {
              "ipAddress": "10.1.0.5",
              "endPointName": "ContosoTest_Downlink_LHCP",
              "port": "51104",
              "protocol": "UDP"
            },
            "modulationConfiguration": null,
            "demodulationConfiguration": null,
            "encodingConfiguration": null,
            "decodingConfiguration": null
          }
        ]
      }
    ]
  }
}
```

Create a contact profile

Follow these instructions to create a contact profile via the Azure portal or use the Azure Orbital Ground Station API.

Modify or delete a contact profile

To modify or delete a contact profile via the Azure portal, navigate to the contact profile resource.

- To modify minimum viable contact duration, minimum elevation, auto tracking, or events hubs telemetry, click 'Overview' on the left panel then click 'Edit properties.'

- To edit links and channels, click 'Links' under 'Configurations' on the left panel then click 'Edit link' on the desired link.
- To edit third-party configurations, click 'Third-Party Configurations' under 'Configurations' on the left panel then click 'Edit' on the desired configuration.
- To delete a contact profile, click 'Overview' on the left panel then click 'Delete.'

You can also use the Azure Orbital Ground Station API to modify or delete a contact profile.

Configure a contact profile for applicable partner ground stations

After onboarding with a partner ground station network, you receive a name that identifies your configuration file. When creating your contact profile, add this configuration name to your link in the 'Third-Party Configuration" parameter. This links your contact profile to the partner network.

Next steps

- Schedule a contact
- Configure the RF chain
- Update the Spacecraft TLE

Spacecraft resource

Article • 12/03/2023

Learn about how you can represent your spacecraft details in Azure Orbital Ground Station.

Spacecraft parameters

The spacecraft resource captures three types of information:

- **Ephemeris** - The latest spacecraft TLE to predict the position and velocity of the satellite.
- **Links** - RF details on center frequency, bandwidth, direction, and polarization for each link.
- **Authorizations** - Regulatory authorizations are held on a per-link, per-site basis.

Ephemeris

The spacecraft ephemeris is captured in Azure Orbital Ground Station using a Two-Line Element, or TLE.

A TLE is associated with the spacecraft to determine contact opportunities at the time of scheduling. The TLE is also used to determine the path the antenna must follow during the contact as the spacecraft passes over the ground station during contact execution.

As TLEs are prone to expiration, the user must keep the TLE up-to-date using the TLE update procedure. A TLE that is more than two weeks old might result in an unsuccessful contact.

Links

Make sure to capture each link that you wish to use with Azure Orbital Ground Station when you create the spacecraft resource. The following details are required:

Expand table

Field	Values
Direction	Uplink or Downlink
Center Frequency	Center frequency in MHz

Field	Values
Bandwidth	Bandwidth in MHz
Polarization	RHCP, LHCP, or Linear Vertical

Dual polarization schemes are represented by two individual links with their respective LHCP and RHCP polarizations.

Authorizations

In order to uphold regulatory requirements across the world, the spacecraft resource contains authorizations for specific links and sites that permit usage of the Azure Orbital Ground Station sites.

The platform will deny scheduling or execution of contacts if the spacecraft resource links aren't authorized. The platform will also deny contact if a profile contains links that aren't included in the spacecraft resource authorized links.

Learn how to initiate ground station licensing and authorize a spacecraft resource.

Create a spacecraft resource

Learn how to create and authorize a spacecraft resource in the Azure portal or Azure Orbital Ground Station API.

Modify or delete spacecraft resources

Spacecraft resources can be modified and deleted via the Azure portal or the Azure Orbital Ground Station API. Once the spacecraft resource is created, modification to the resource is dependent on the authorization status:

- When the spacecraft is **unauthorized**, the spacecraft resource can be modified. The API is the recommended way to update the spacecraft resource as the Azure portal only allows for TLE updates.
- After the spacecraft is **authorized**, TLE updates are the only modifications possible. Other fields, such as links, become immutable.

To delete the spacecraft resource, you must first delete all scheduled contacts associated with that spacecraft resource. See contact resource for more information.

To delete a spacecraft via the Azure portal, navigate to the spacecraft resource. Click 'Overview' on the left panel, then click 'Delete.' Alternatively, use the Spacecrafts REST Operation Group to delete a spacecraft with the Azure Orbital Ground Station API.

Next steps

- Create and authorize a spacecraft resource

Modular Datacenter overview

Article • 02/28/2022

Azure Modular Datacenter (MDC) is based on Azure Stack Hub. MDC is a portable, rapidly deployable datacenter appropriate for supporting large-scale combat operations in temporary and fixed command posts.

Azure Stack Hub is a horizontally and vertically scalable solution that provides multitenant, native hybrid cloud capabilities for infrastructure as a service (IaaS) and platform as a service (PaaS) services to environments at the edge. Azure Stack Hub supports a variety of modular scenarios for temporary and fixed command posts and expeditionary forces.

Azure Stack Hub is an integrated hardware and software appliance that's commercially available in a variety of capacities based on scaling through node scale-units. Azure Stack Hub is available with extensions, which include general-purpose graphics processing unit (GPU)-enabled configurations and expandable external storage.

Use MDC

Azure Stack delivers on four core principles that are consistent with and extend Azure's capabilities for modular edge scenarios.

Develop and deliver apps with a common DevOps model including API symmetry with Azure

The consistency between Azure and Azure Stack means solutions are developed once, deployed to support a variety of use cases, and secured and sustained by using a common set of tools. Examples of tools include Azure Key Vault for key management and Azure Monitor for resource monitoring and management. Azure Stack is interoperable with on-premises data, apps, and tools for DevOps and secure operations, for example, key management.

Deliver Azure services on-premises

Azure Stack runs in disadvantaged (contested, congested, or denied) communications environments and robust communications environments. Azure Stack isn't dependent on connectivity to Azure to execute mission apps and enable local operations.

Use integrated hardware and software delivery experience

The modular edge requires a range of capabilities that provide baseline compute and storage services and also advanced capability for machine learning, AI, and analytics. The ability to connect to your secure cloud from the modular edge when able or, conversely, to operate independently from it in austere environments is critical to provide access to data needed for decisions.

Keep your datacenter secure and available with hybrid cloud security operations

The cloud native design of Azure Stack removes the operational complexities of traditional virtualization environments. Administrators can choose when to patch and orchestrate the entire operation through the installation process and the Azure Stack management fabric built natively into the system.

Remote assisted administration by Microsoft or a Microsoft-managed offering in the field for Azure Stack devices can be enabled by providing users with the appropriate access through role-based access control (RBAC) to perform actions through the administrative portal, privileged management endpoint, or command-line interface (CLI). This capability enables all patching and other administrative and monitoring activities to be performed by Microsoft. Azure Stack is field upgradable and updatable through secure administration portal or CLI commands that both apply security updates and IaaS and PaaS feature updates to maintain commercial parity with Azure, where appropriate, from local or remote networks.

Benefits of using MDC

MDC supports a consistent environment with Azure in disadvantaged communications environments:

- Static, modular, rapidly deployable datacenters with Azure cloud services to power large analytics apps in modular operations centers.
- Azure's modular edge offerings maintain consistency between the cloud and the edge through a singular approach for supporting IaaS primitives such as virtual machines, storage, and virtual networking.
- Azure Active Directory and RBAC support.
- Common administrative interfaces.

- API symmetry and support for Microsoft, third party, and open-source DevOps tools.
- Management and monitoring through Azure Log Analytics and Azure Security Center.
- Bring the agility of cloud computing to your on-premises environment and the edge by enabling a hybrid cloud.
  You can:
  - Reuse code and run cloud-native apps consistently across Azure and your on-premises environments.
  - Run traditional virtualized workloads with optional connections to Azure services.
  - Transfer data to the cloud, or keep it in your sovereign datacenter to maintain compliance.
  - Run hardware-accelerated machine learning, containerized, or virtualized workloads all at the intelligent edge.

Next steps

Azure Stack Hub capacity planning

Using domain analysis to model microservices

Article • 12/16/2022

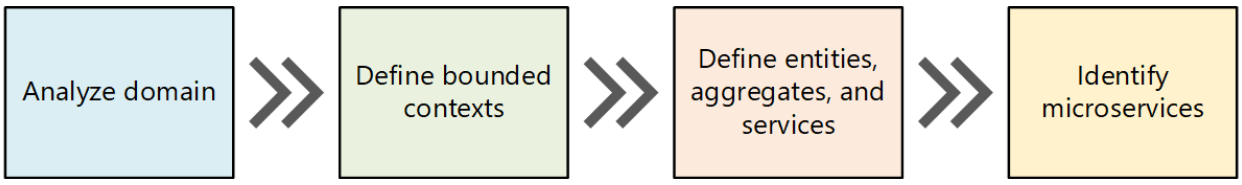
One of the biggest challenges of microservices is to define the boundaries of individual services. The general rule is that a service should do "one thing" — but putting that rule into practice requires careful thought. There is no mechanical process that will produce the "right" design. You have to 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.

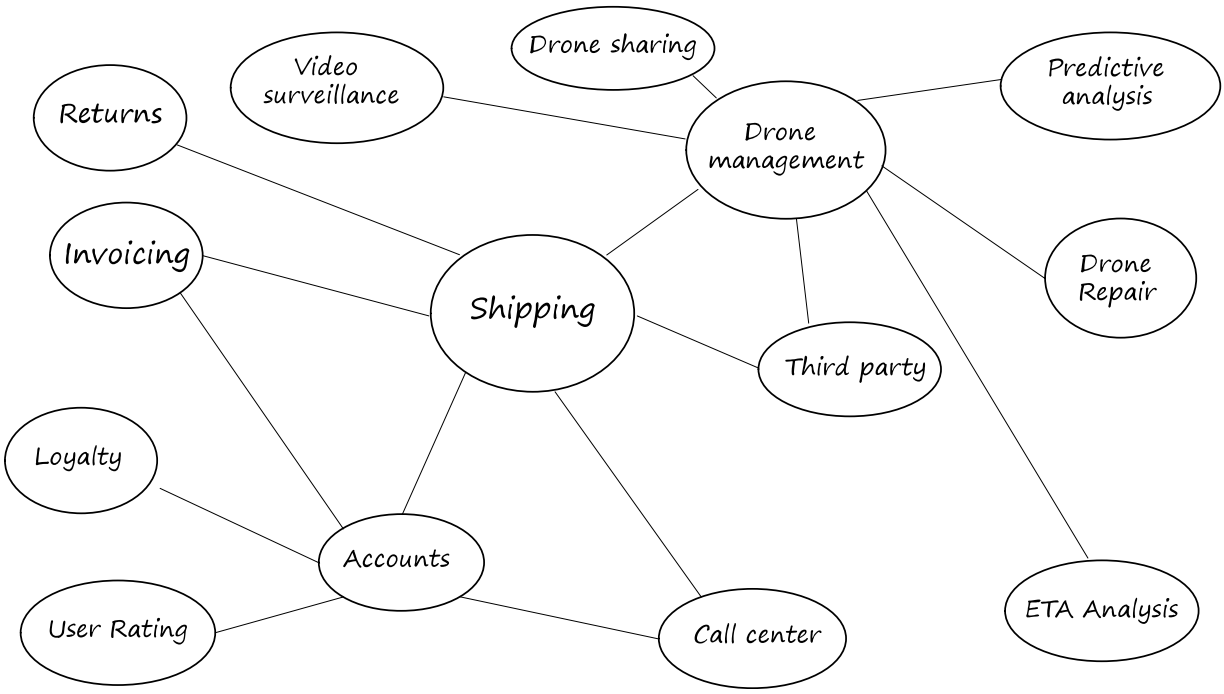
Before writing any code, you need a bird's eye view of the system that you are creating. DDD starts by modeling the business domain and creating a *domain model*. The domain model is an abstract model of the business domain. It distills and organizes domain knowledge, and provides a common language for 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.

> ⓘ **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.

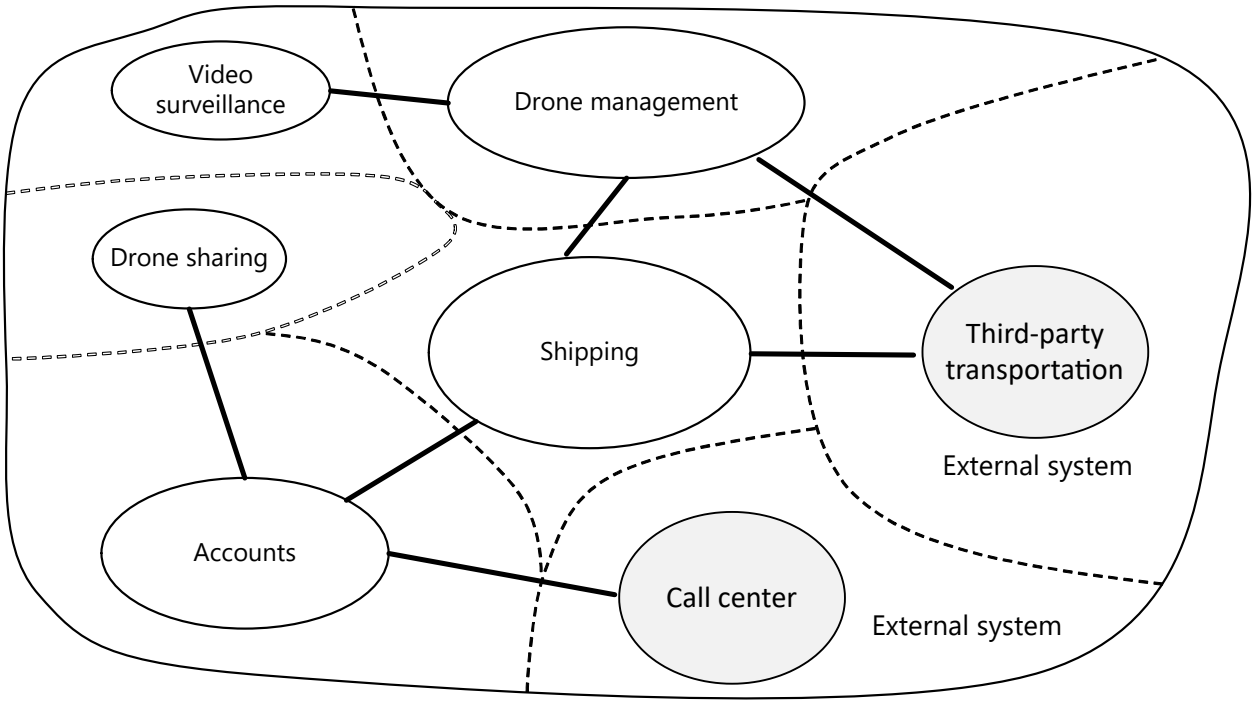
Define bounded contexts

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

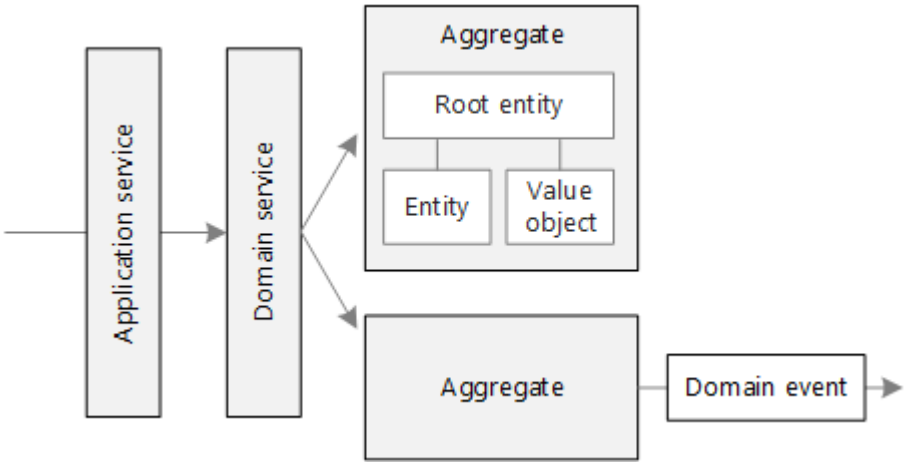
Using tactical DDD to design microservices

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.

> ⓘ **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**.

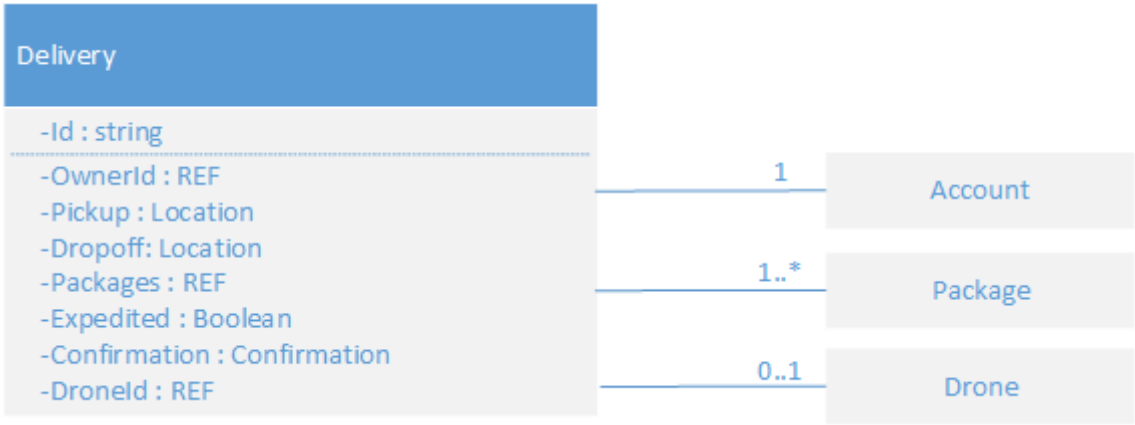
- Delivery
- Package
- Drone

- 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.

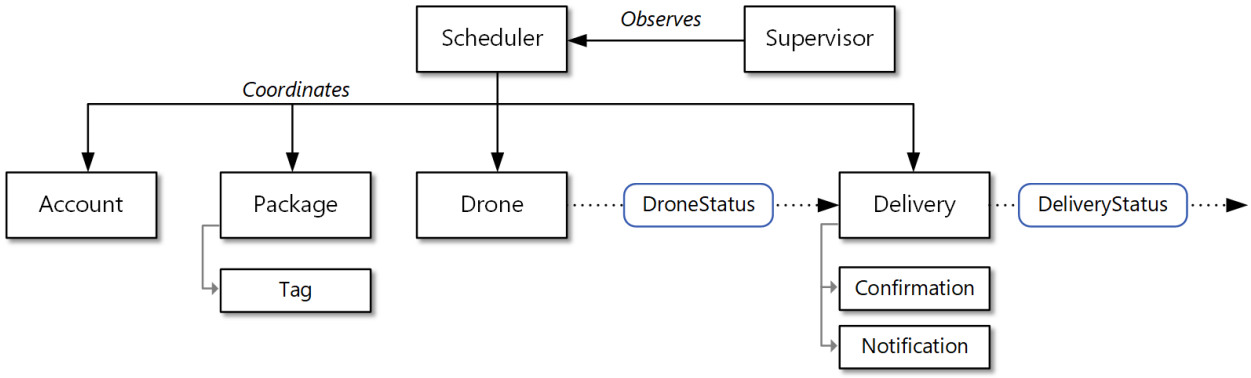


There are two domain events:

- 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

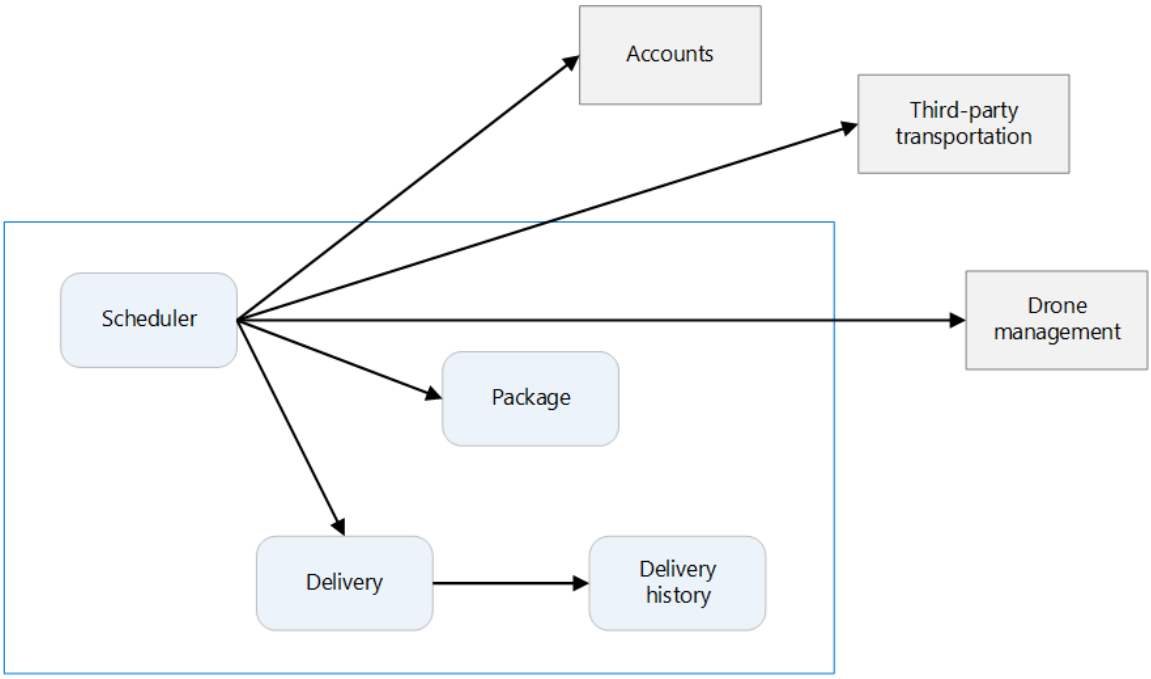
Related resources

- 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.



Shipping bounded context

Microsoft Azure

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.

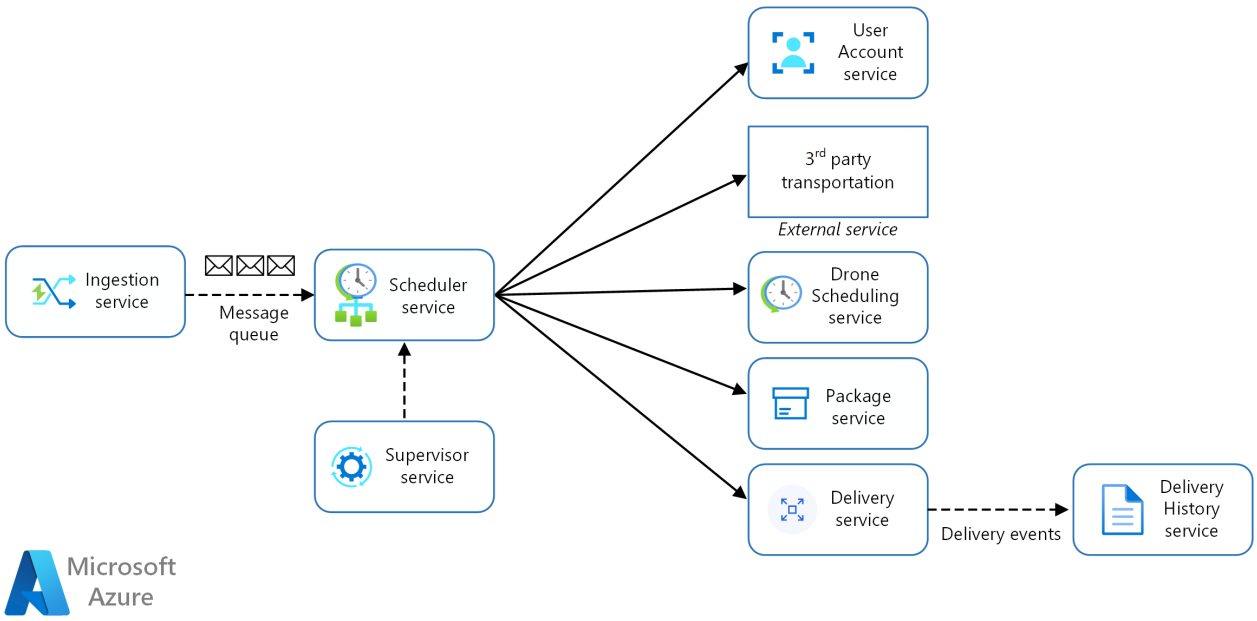
The details of the Drone and Account bounded contexts are beyond the scope of this guidance, so we created mock services for them in our reference implementation. But here are some factors to consider in this situation:

- 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

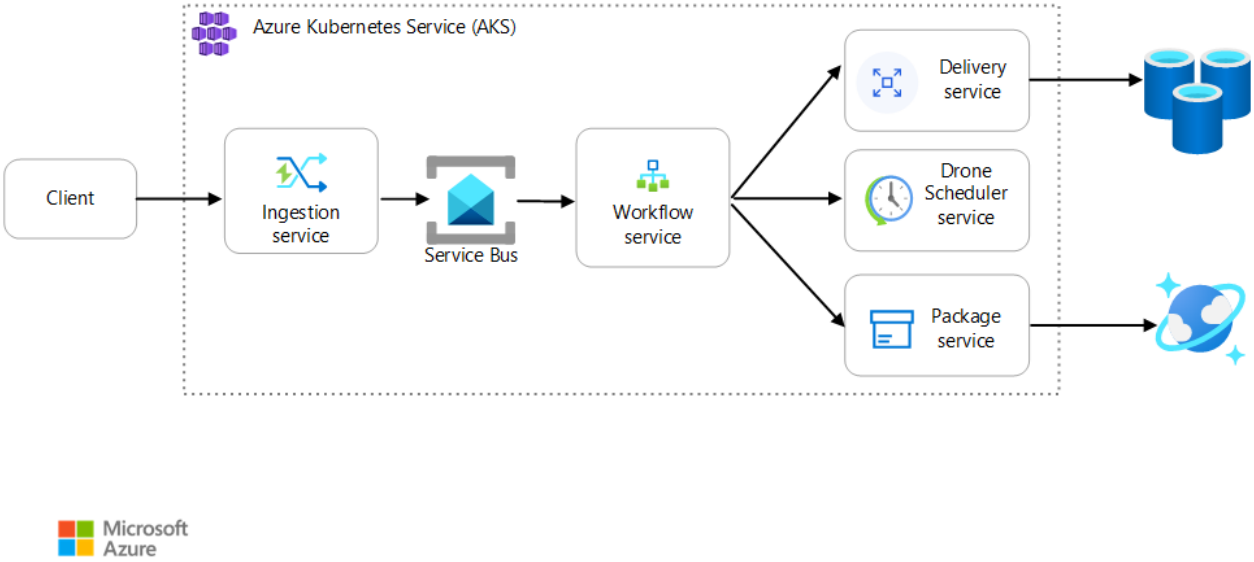
Prerequisites

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

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.

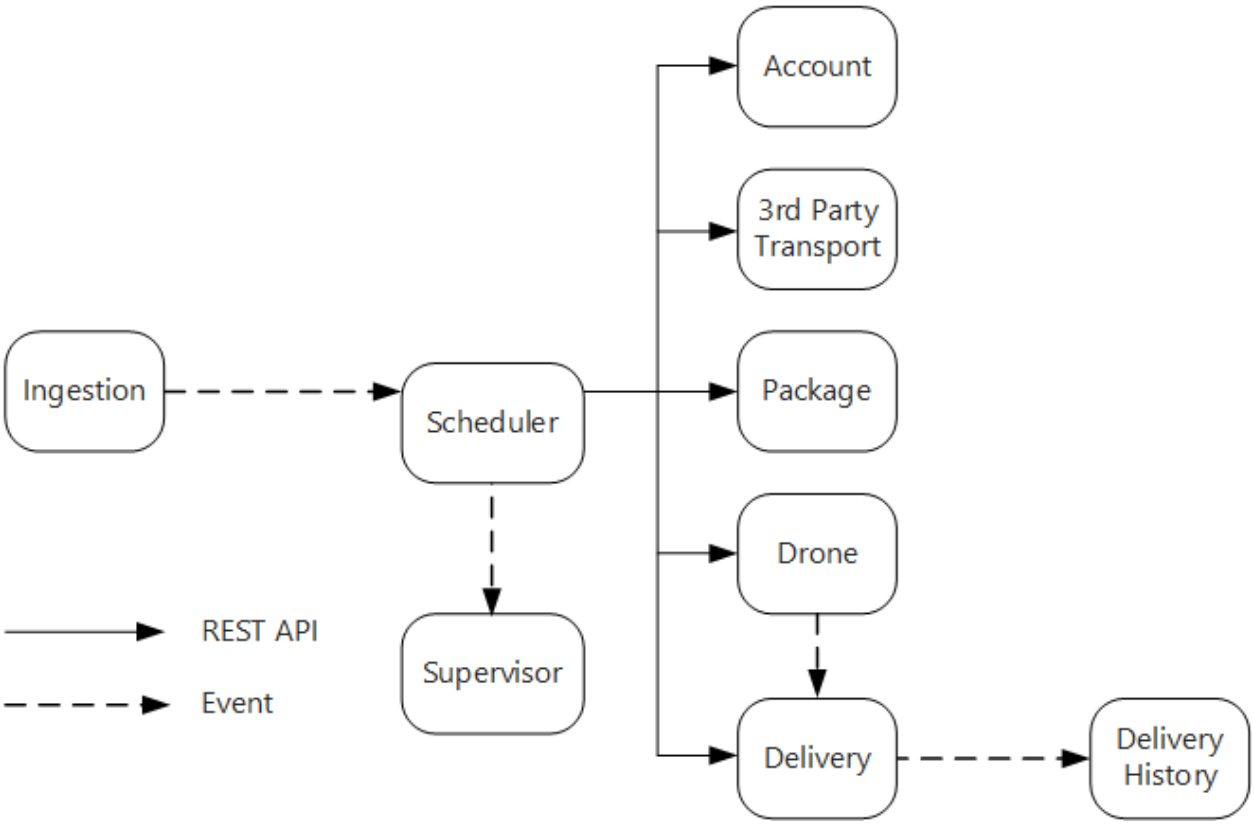
With these considerations in mind, the development team made the following design choices for the Drone Delivery application:

- 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

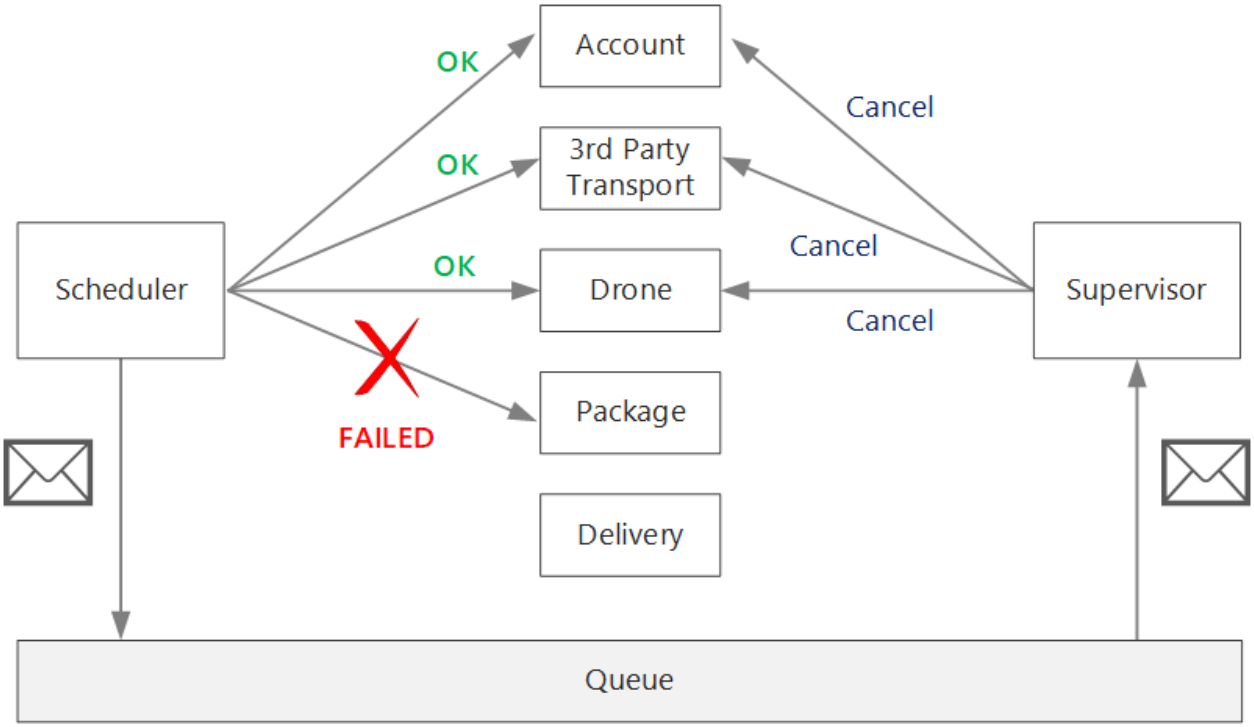
A common challenge in microservices is correctly handling transactions that span multiple services. Often in this scenario, the success of a transaction is all or nothing — if one of the participating services fails, the entire transaction must fail.

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

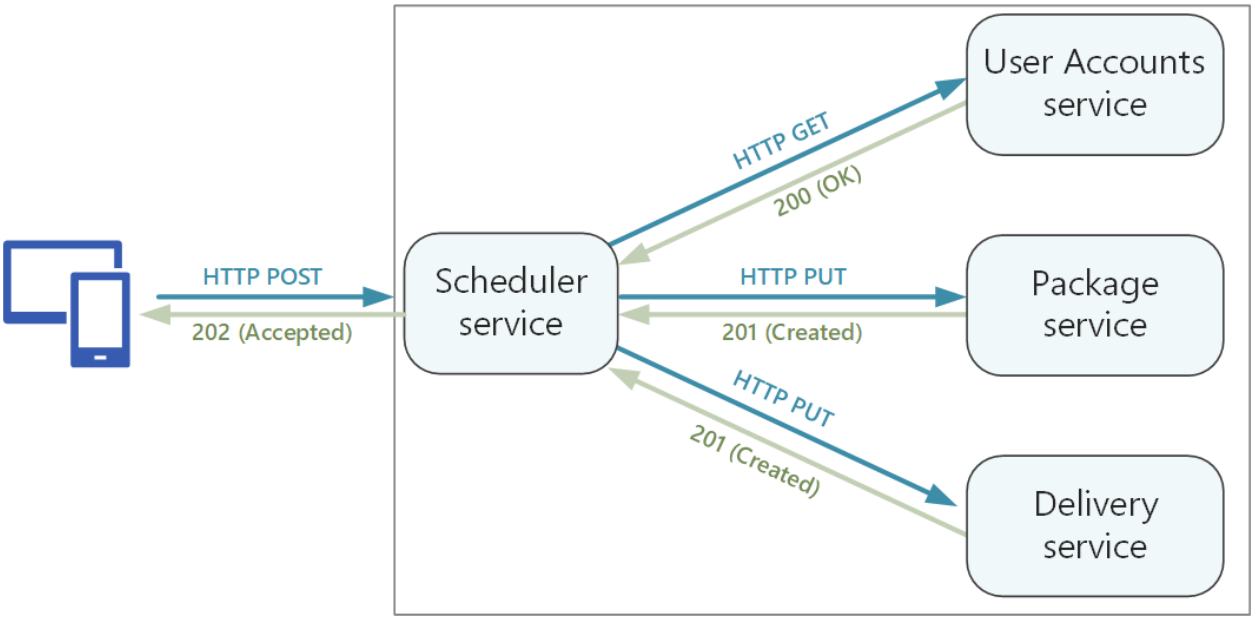
- Design a microservices architecture
- Using domain analysis to model microservices

- 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

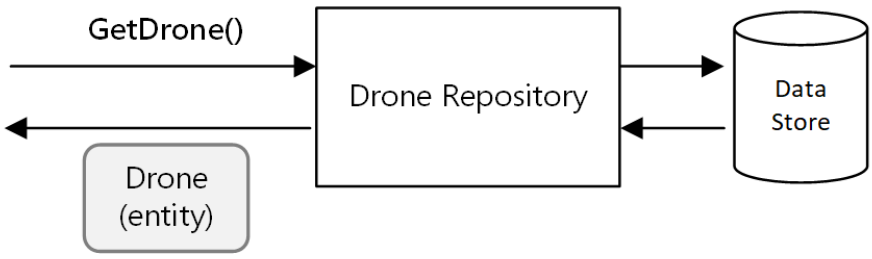
Patterns such as entity, aggregate, and value object are designed to place certain constraints on the objects in your domain model. In many discussions of DDD, the patterns are modeled using object-oriented (OO) language concepts like constructors or property getters and setters. For example, *value 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

```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;
    }
}
```

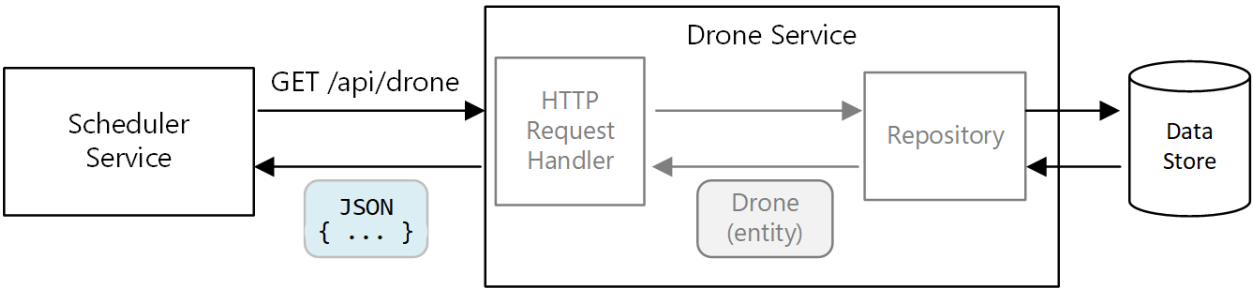
These sorts of coding practices are particularly important when building a traditional monolithic application. With a large code base, many subsystems might use the `Location` object, so it's important for the object to enforce correct behavior.

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

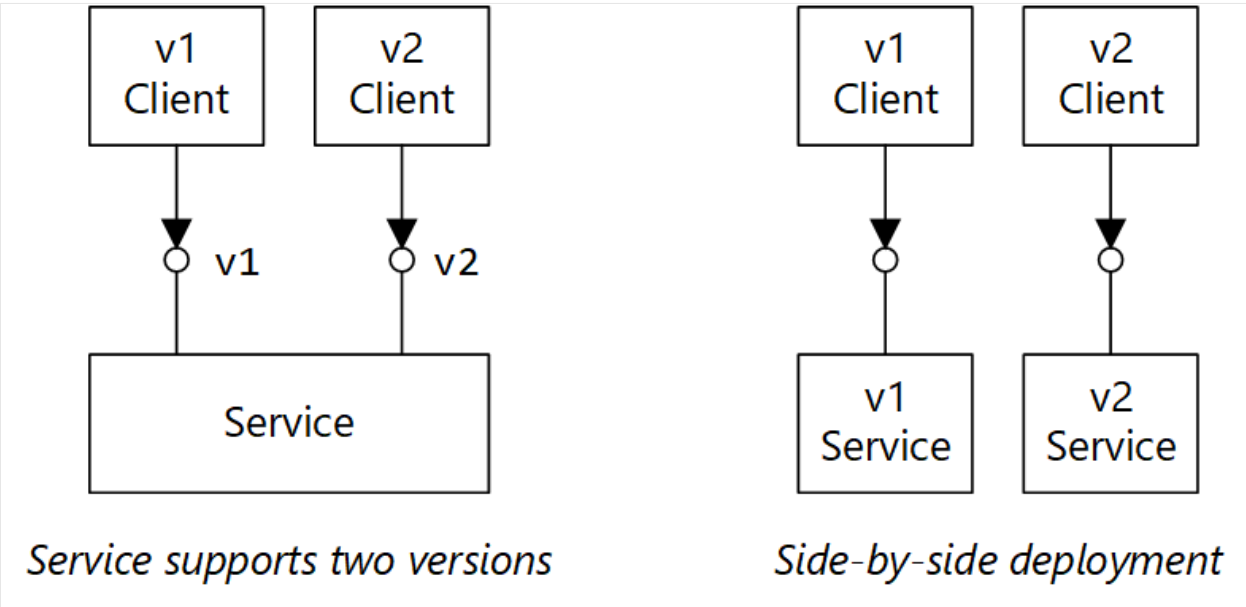
DDD concept	REST equivalent	Example
Aggregate	Resource	`{ "1":1234, "status":"pending"... }`
Identity	URL	`https://delivery-service/deliveries/1`
Child entities	Links	`{ "href": "/deliveries/1/confirmation" }`
Update value objects	PUT or PATCH	`PUT https://delivery-service/deliveries/1/dropoff`
Repository	Collection	`https://delivery-service/deliveries?status=pending`

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

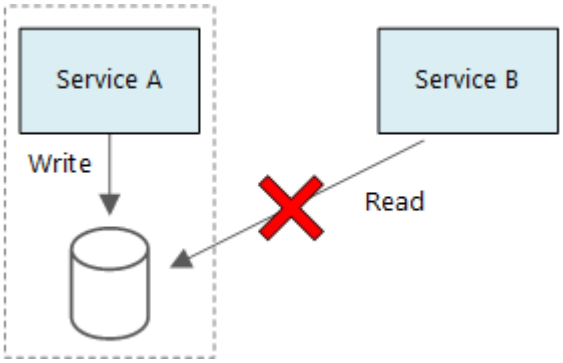
Data considerations for microservices

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.

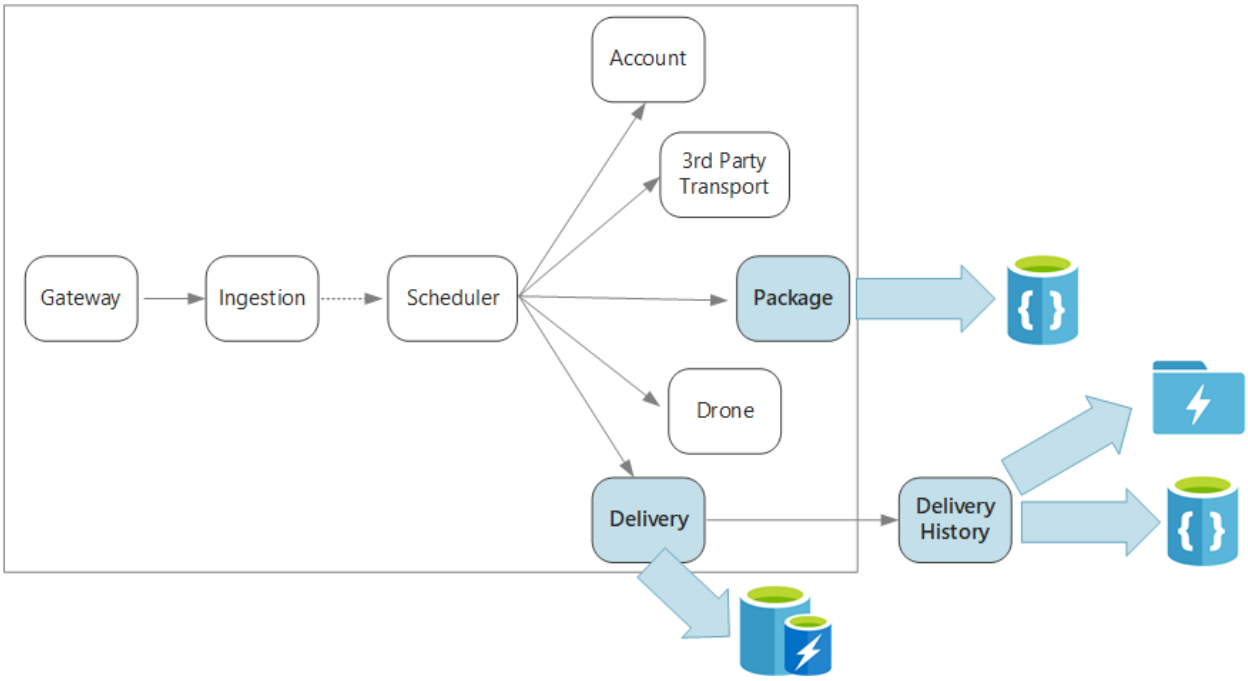
- At high scale, events can become a bottleneck on the system, so consider using aggregation or batching to reduce the total load.

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

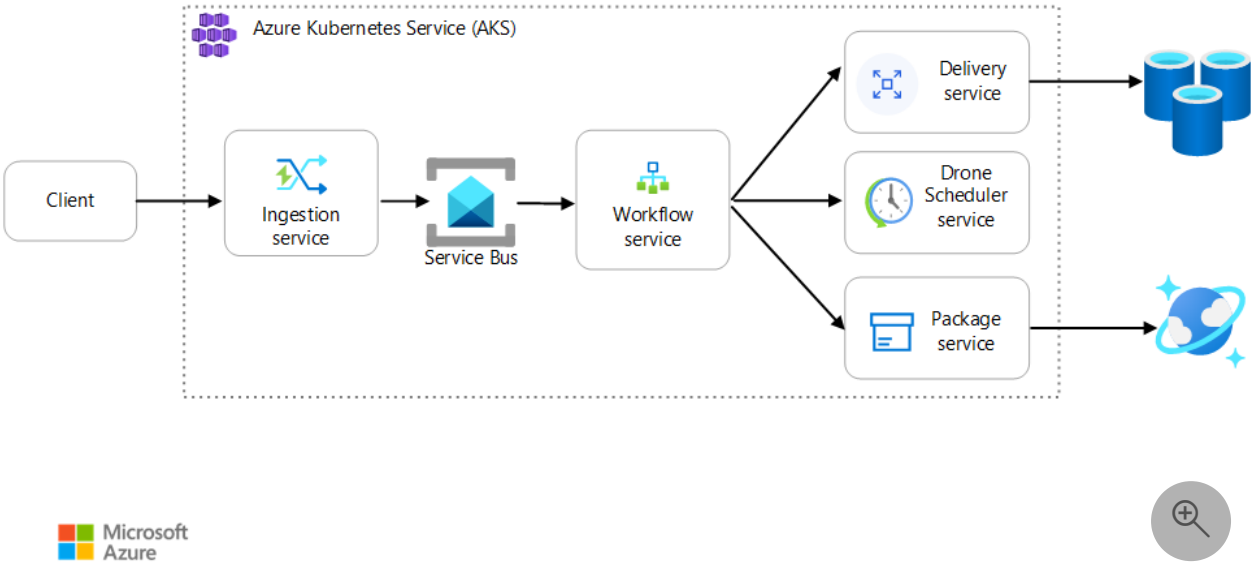
Related resources

- Using domain analysis to model microservices
- Design a microservices architecture
- Design APIs for microservices
- Microservices architecture design

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:



Download a Visio file of this architecture.

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.

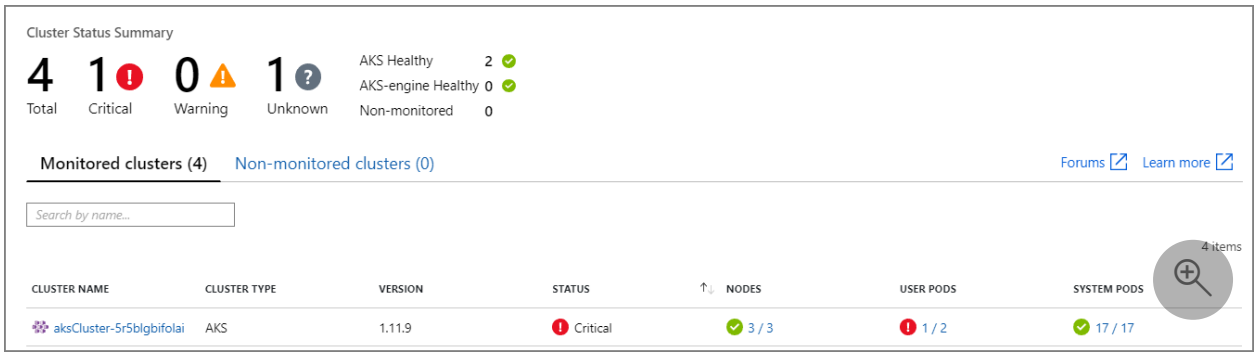
- **Container** metrics. For containerized applications, you need to collect metrics at the container level, not just at the VM level.

- **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

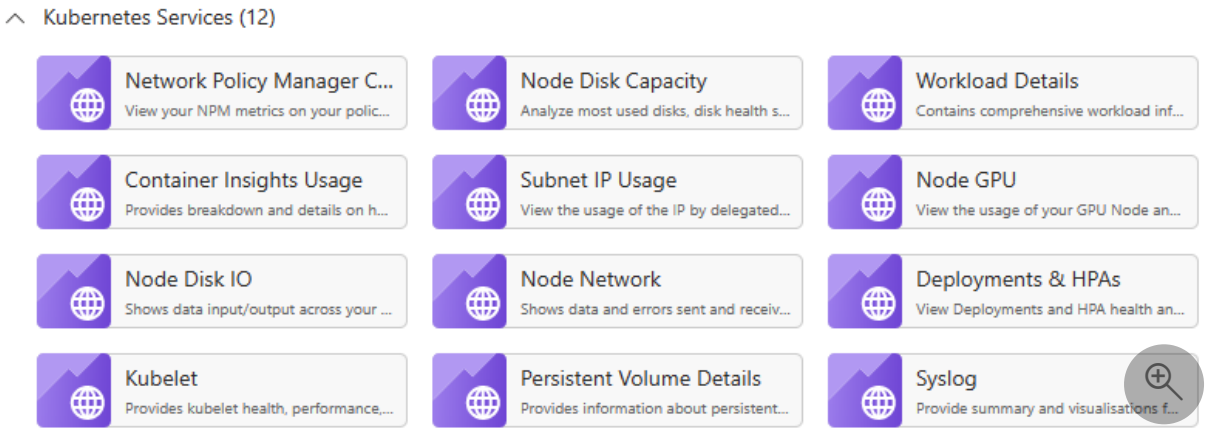
Use Azure Monitor to monitor the health of your clusters. The following screenshot shows a cluster that has critical errors in user-deployed pods:



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.



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 traceIds into responses and payloads, and/or reading traceIds 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

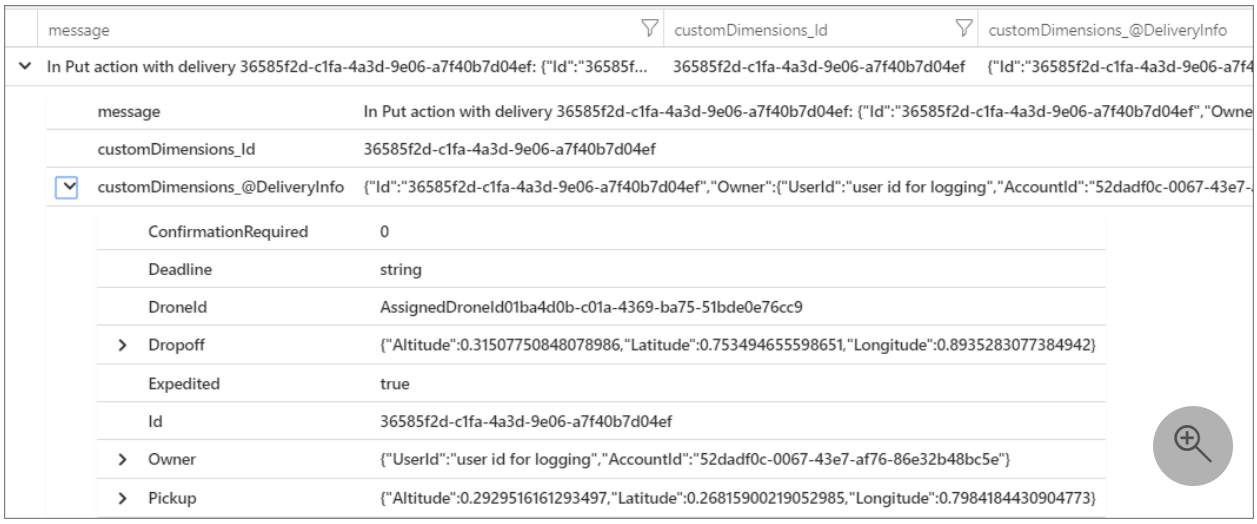
```
{"@t":"2019-06-13T00:57:09.9932697Z","@mt":"In Put action with delivery
{Id}: {@DeliveryInfo}","Id":"36585f2d-c1fa-4a3d-9e06-
a7f40b7d04ef","DeliveryInfo":{...
```

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:



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
```

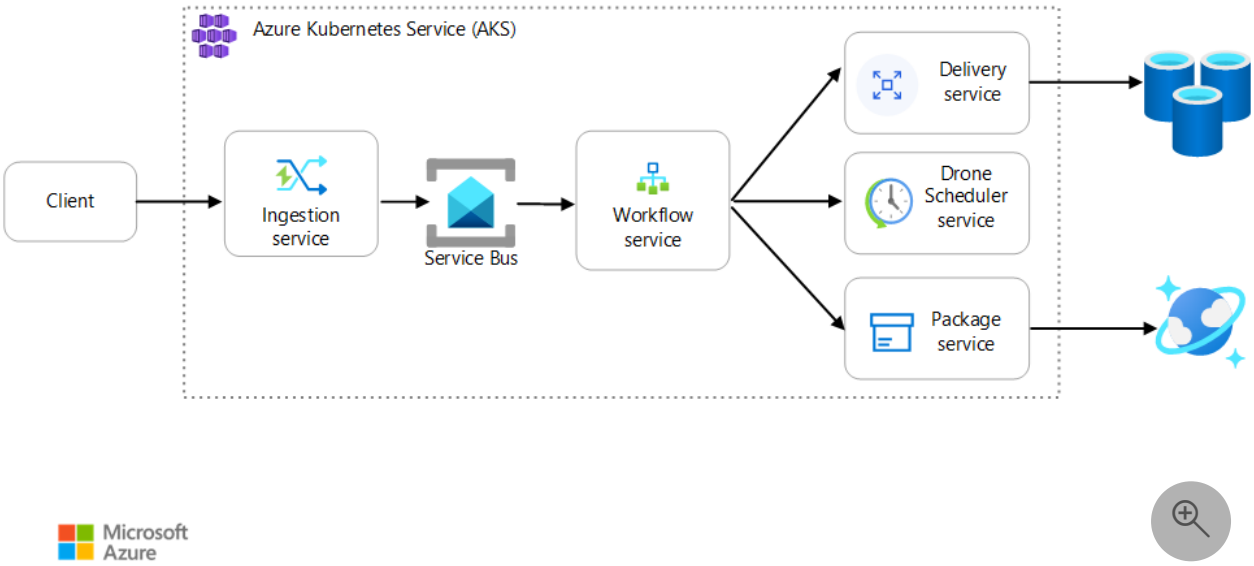
If you use a library or framework that's already instrumented with OpenTelemetry, it handles creating spans and requests, but the application code might also create units of work. For example, a method that loops through an array of entities that performs work on each one might create a span for each iteration of the processing loop. For information about adding instrumentation to application and library code, see the OpenTelemery 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

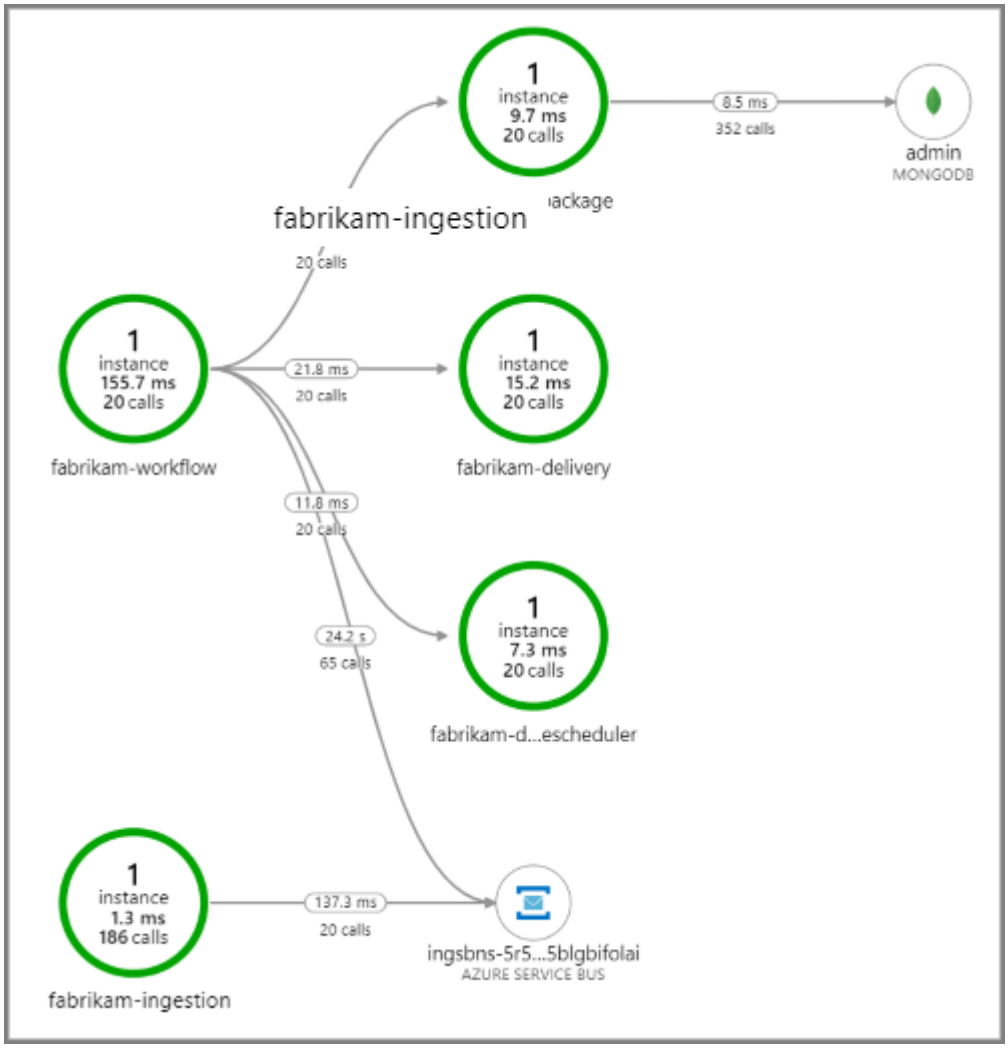
This example describes the path of a distributed transaction through a set of microservices. The example is based on a drone delivery application.



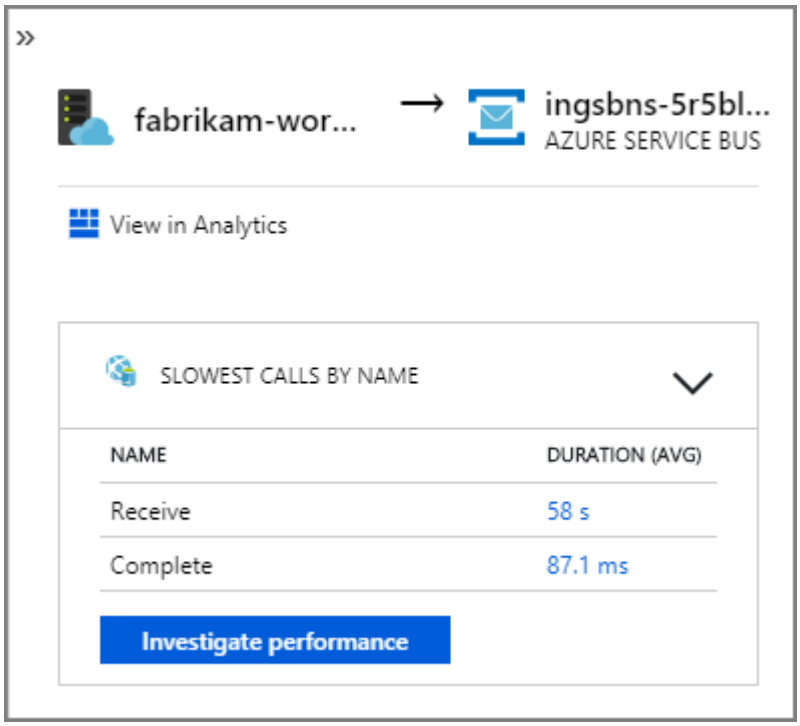
In this scenario, the distributed transaction includes these steps:

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).

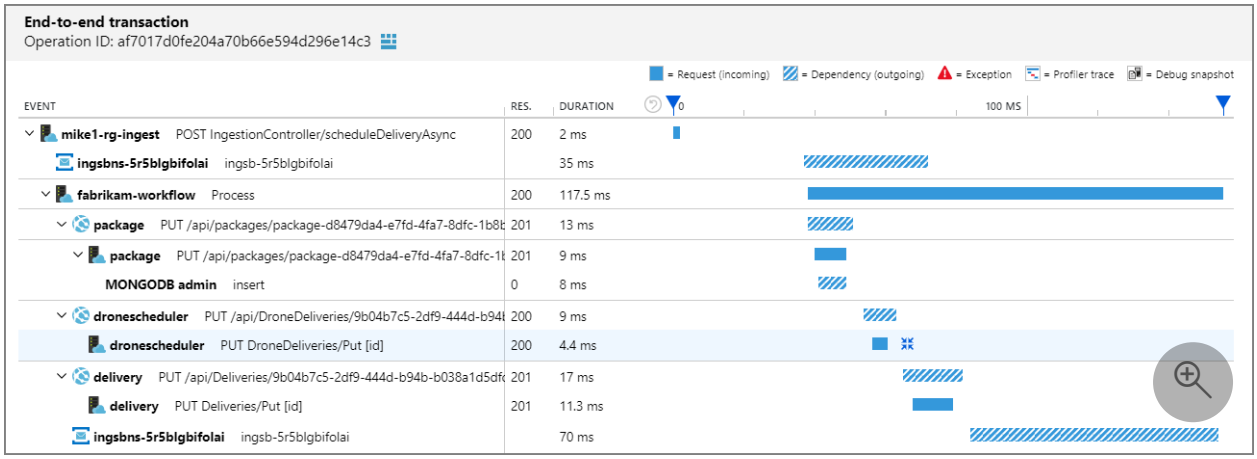
The following screenshot shows the application map for the drone delivery application. This map shows calls to the public API endpoint that result in a workflow that involves five microservices.



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:

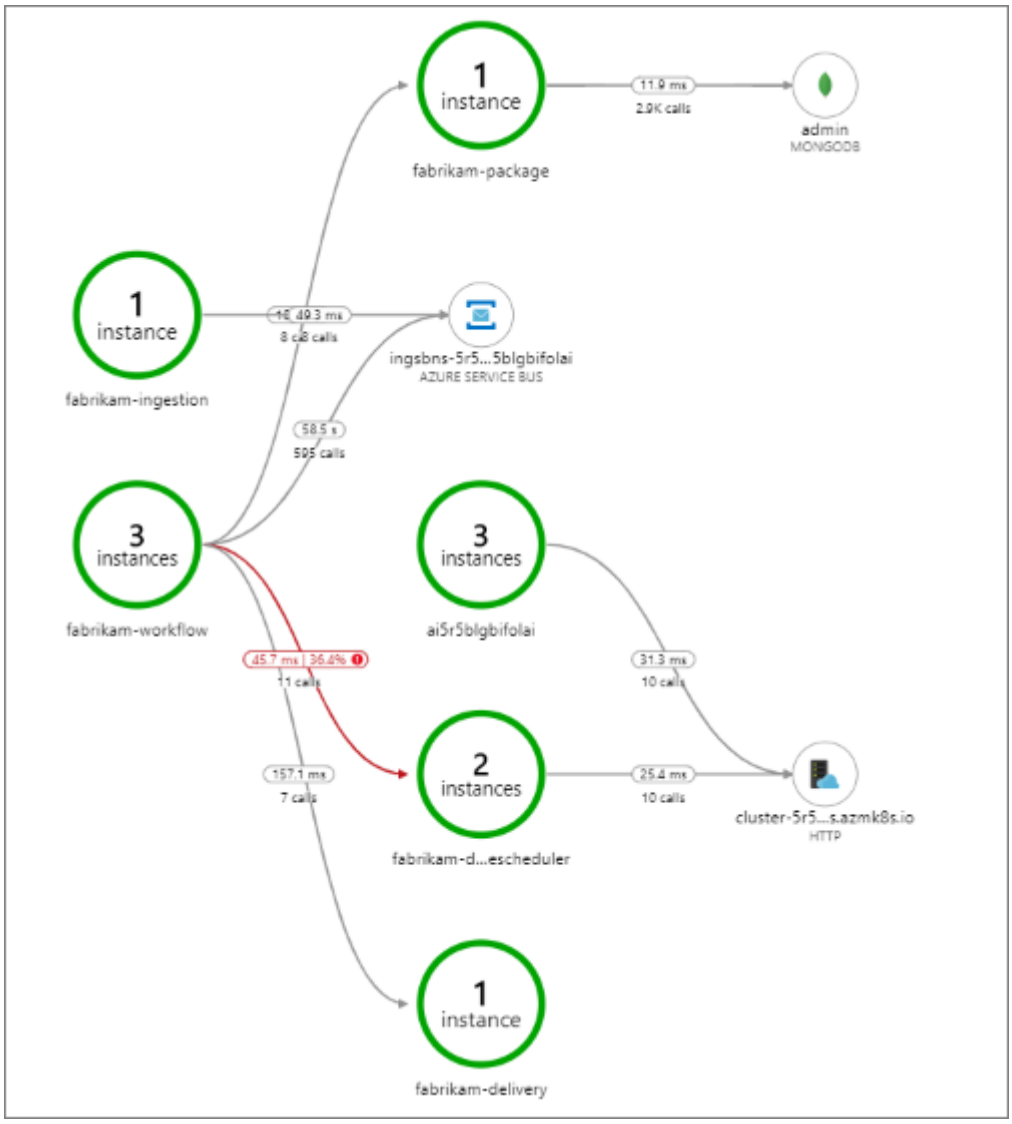


Because every call includes an operation ID, you can also view the end-to-end steps of a single transaction, including timing information and the HTTP calls at each step. Here's the visualization of one such transaction:

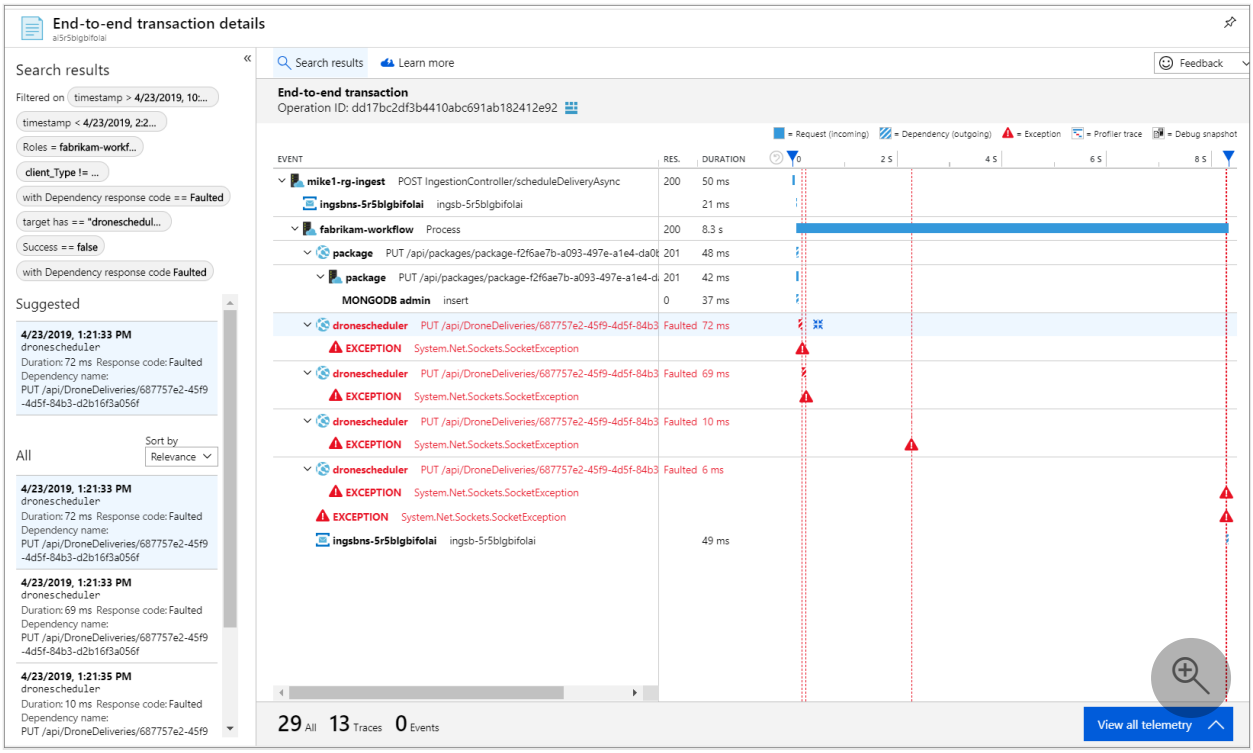


This visualization shows the steps from the ingestion service to the queue, from the queue to the Workflow service, and from the Workflow service to the other back-end services. The last step is the Workflow service marking the Service Bus message as completed.

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 5*xx* 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

Distributed business transaction performance tuning

Azure Kubernetes Service (AKS) | Azure Cache for Redis

This article describes how a development team used metrics to find bottlenecks and improve the performance of a distributed system. The article is based on actual load testing that we did for a sample application. The application is from the Azure Kubernetes Service (AKS) Baseline for microservices.

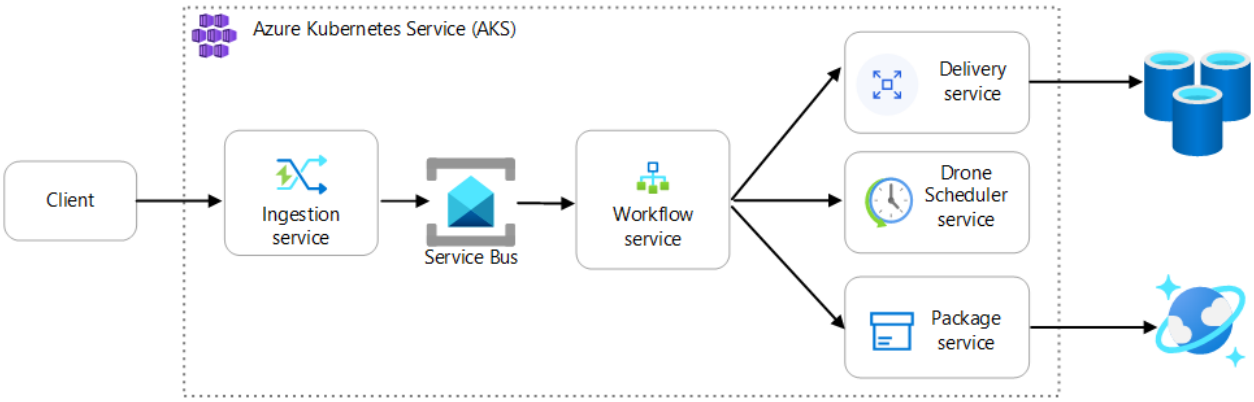
This article is part of a series. Read the first part here.

Scenario: A client application initiates a business transaction that involves multiple steps.

This scenario involves a drone delivery application that runs on AKS. Customers use a web app to schedule deliveries by drone. Each transaction requires multiple steps that are performed by separate microservices on the back end:

- The Delivery service manages deliveries.
- The Drone Scheduler service schedules drones for pickup.
- The Package service manages packages.

There are two other services: An Ingestion service that accepts client requests and puts them on a queue for processing, and a Workflow service that coordinates the steps in the workflow.



For more information about this scenario, see Designing a microservices architecture.

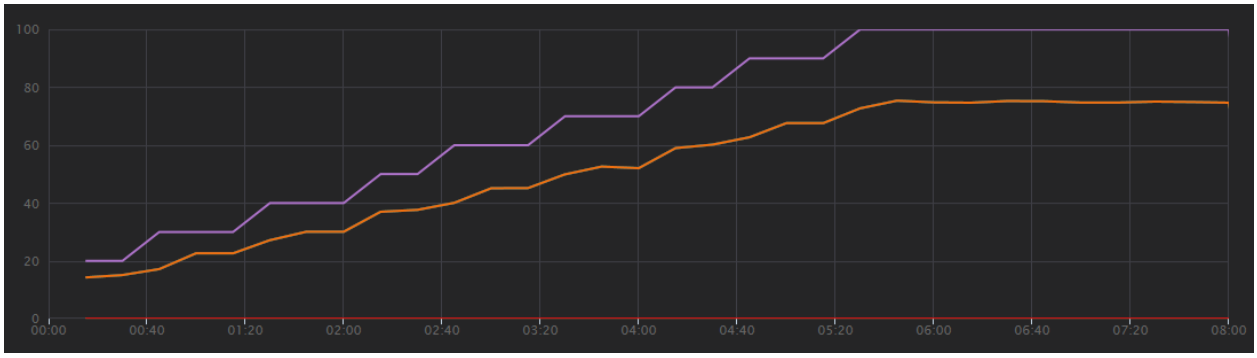
Test 1: Baseline

For the first load test, the team created a six-node AKS cluster and deployed three replicas of each microservice. The load test was a step-load test, starting at two simulated users and ramping up to 40 simulated users.

Expand table

Setting	Value
Cluster nodes	6
Pods	3 per service

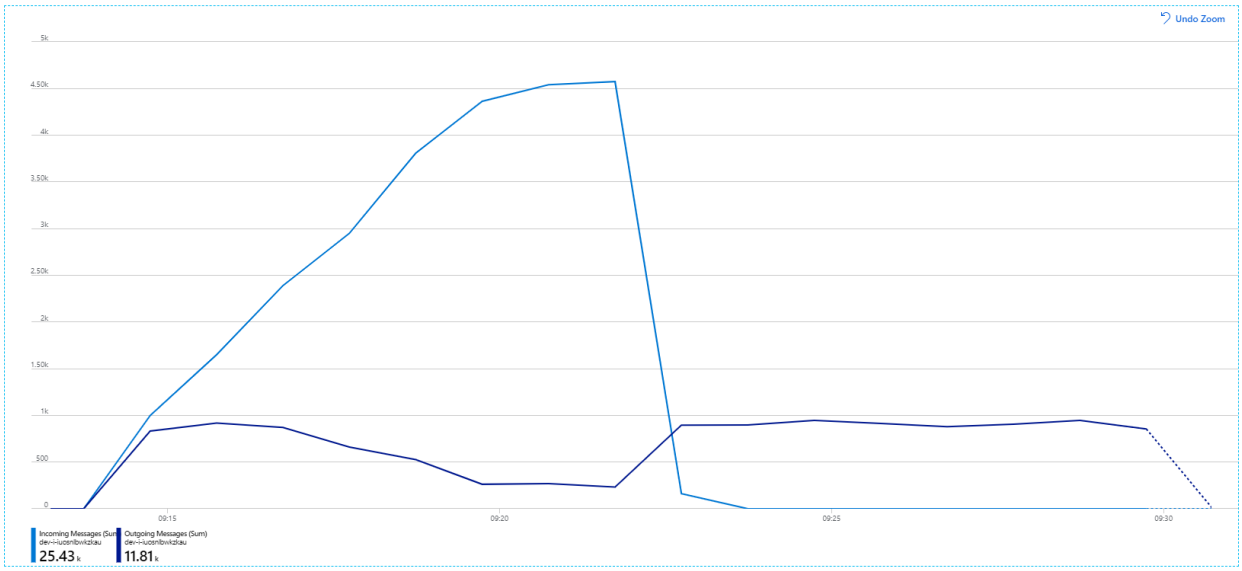
The following graph shows the results of the load test, as shown in Visual Studio. The purple line plots user load, and the orange line plots total requests.



The first thing to realize about this scenario is that client requests per second is not a useful metric of performance. That's because the application processes requests asynchronously, so the client gets a response right away. The response code is always HTTP 202 (Accepted), meaning the request was accepted but processing is not complete.

What we really want to know is whether the backend is keeping up with the request rate. The Service Bus queue can absorb spikes, but if the backend cannot handle a sustained load, processing will fall further and further behind.

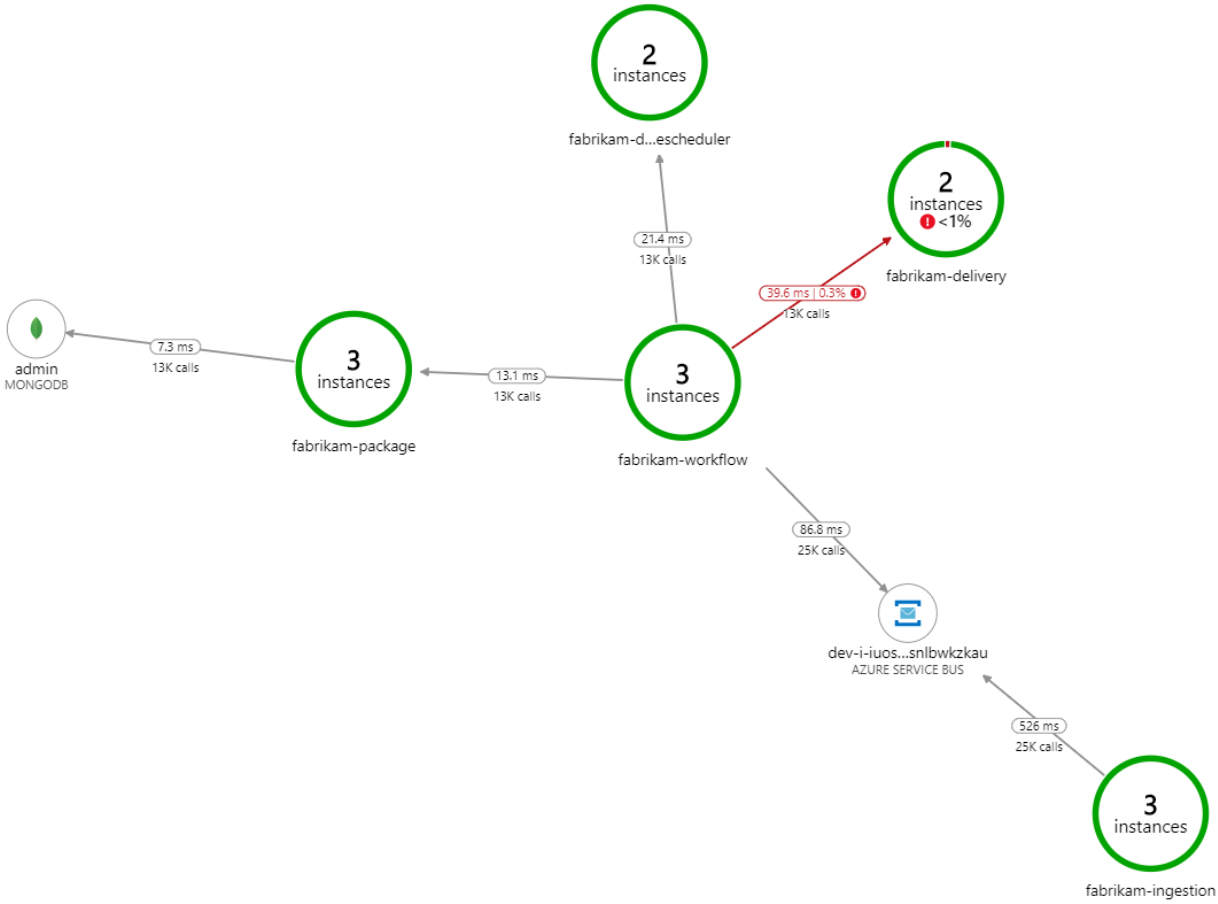
Here's a more informative graph. It plots the number incoming and outgoing messages on the Service Bus queue. Incoming messages are shown in light blue, and outgoing messages are shown in dark blue:



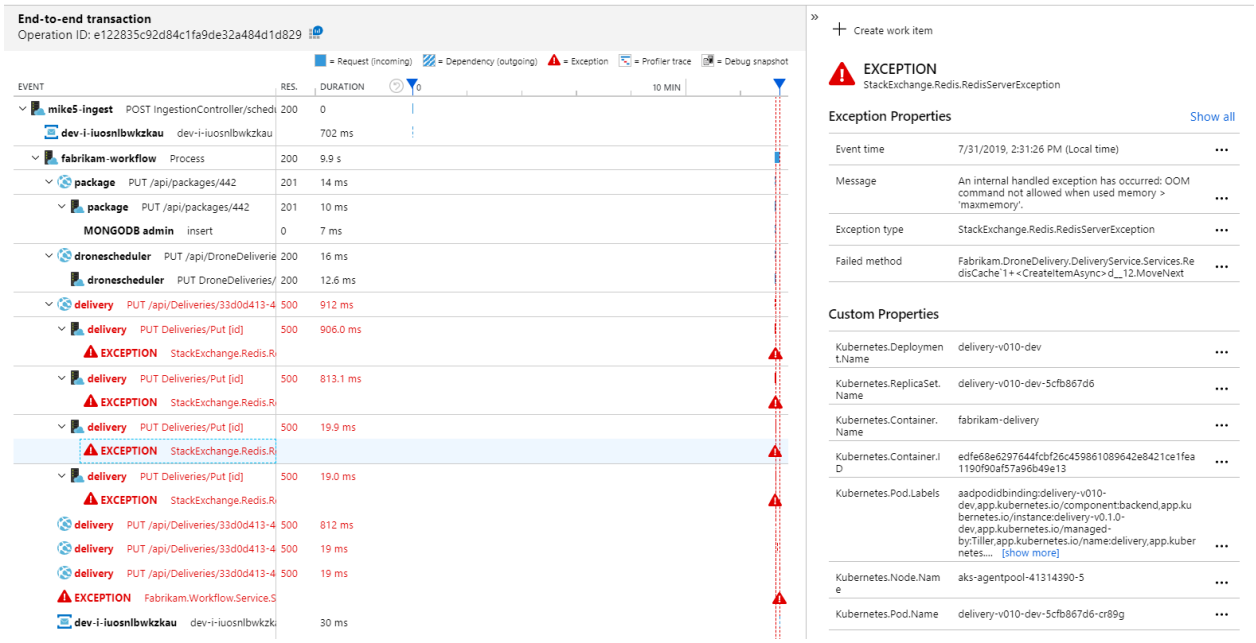
This chart is showing that the rate of incoming messages increases, reaching a peak and then dropping back to zero at the end of the load test. But the number of outgoing messages peaks early in the test and then actually drops. That means the Workflow service, which handles the requests, isn't keeping up. Even after the load test ends (around 9:22 on the graph), messages are still being processed as the Workflow service continues to drain the queue.

What's slowing down the processing? The first thing to look for is errors or exceptions that might indicate a systematic issue. The Application Map in Azure Monitor shows the graph of calls between components, and is a quick way to spot issues and then click through to get more details.

Sure enough, the Application Map shows that the Workflow service is getting errors from the Delivery service:



To see more details, you can select a node in the graph and click into an end-to-end transaction view. In this case, it shows that the Delivery service is returning HTTP 500 errors. The error messages indicate that an exception is being thrown due to memory limits in Azure Cache for Redis.

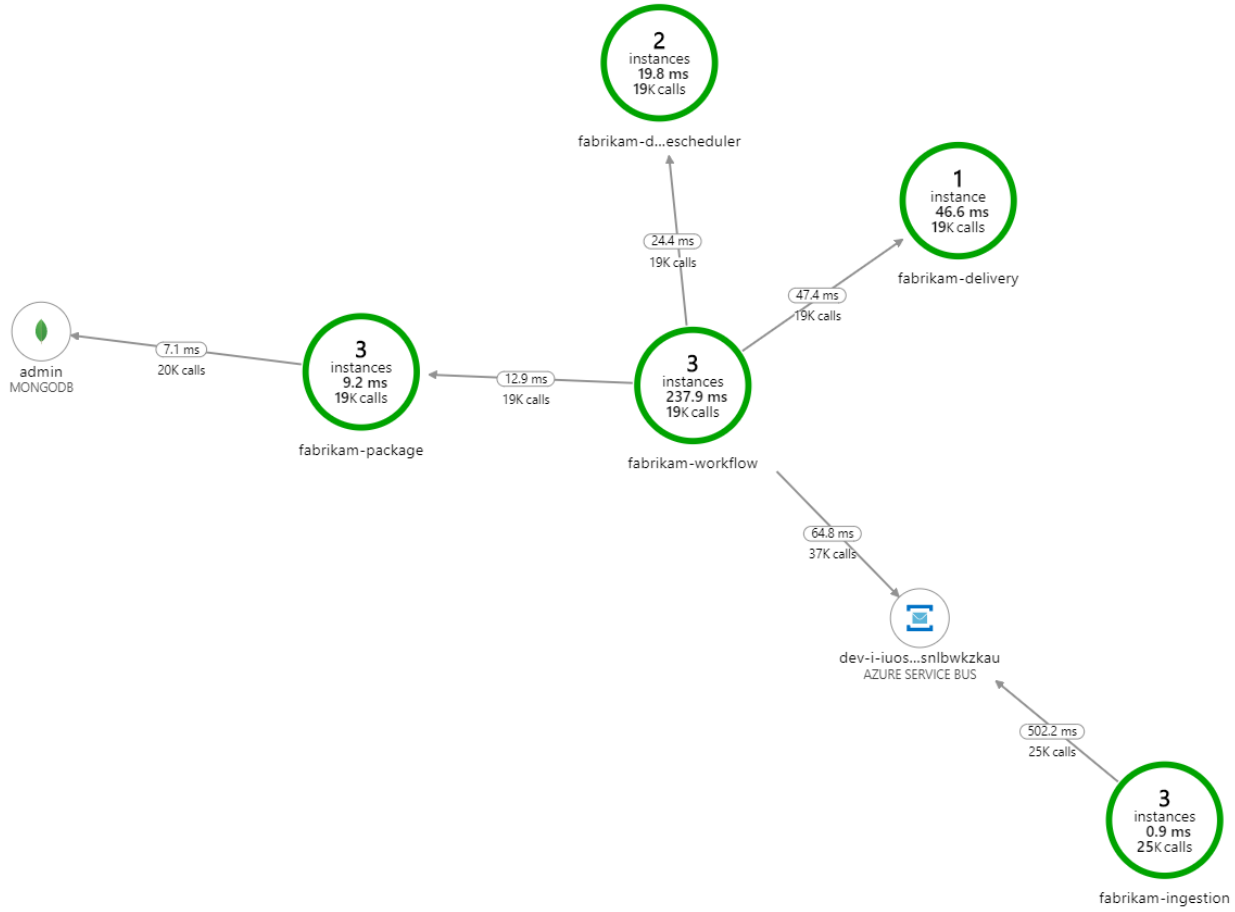


You may notice that these calls to Redis don't appear in the Application Map. That's because the .NET library for Application Insights doesn't have built-in support for tracking Redis as a dependency. (For a list of what's supported out of the box, see Dependency auto-collection.) As a fallback, you can use the TrackDependency API to

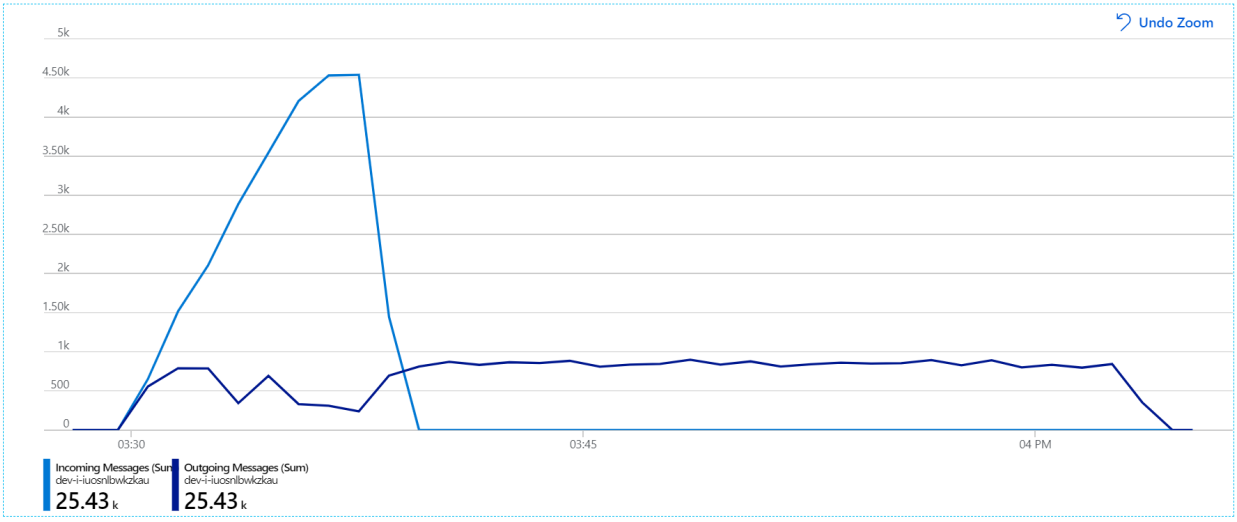
track any dependency. Load testing often reveals these kinds of gaps in the telemetry, which can be remediated.

Test 2: Increased cache size

For the second load test, the development team increased the cache size in Azure Cache for Redis. (See How to Scale Azure Cache for Redis.) This change resolved the out-of-memory exceptions, and now the Application Map shows zero errors:



However, there is still a dramatic lag in processing messages. At the peak of the load test, the incoming message rate is more than 5× the outgoing rate:



The following graph measures throughput in terms of message completion — that is, the rate at which the Workflow service marks the Service Bus messages as completed. Each point on the graph represents 5 seconds of data, showing ~16/sec maximum throughput.

This graph was generated by running a query in the Log Analytics workspace, using the Kusto query language:

Kusto

```
let start=datetime("2020-07-31T22:30:00.000Z");
let end=datetime("2020-07-31T22:45:00.000Z");
dependencies
| where cloud_RoleName == 'fabrikam-workflow'
| where timestamp > start and timestamp < end
| where type == 'Azure Service Bus'
| where target has 'https://dev-i-iuosnlbwkzkau.servicebus.windows.net'
| where client_Type == "PC"
| where name == "Complete"
| summarize succeeded=sumif(itemCount, success == true),
```

```
failed=sumif(itemCount, success == false) by bin(timestamp, 5s)
| render timechart
```

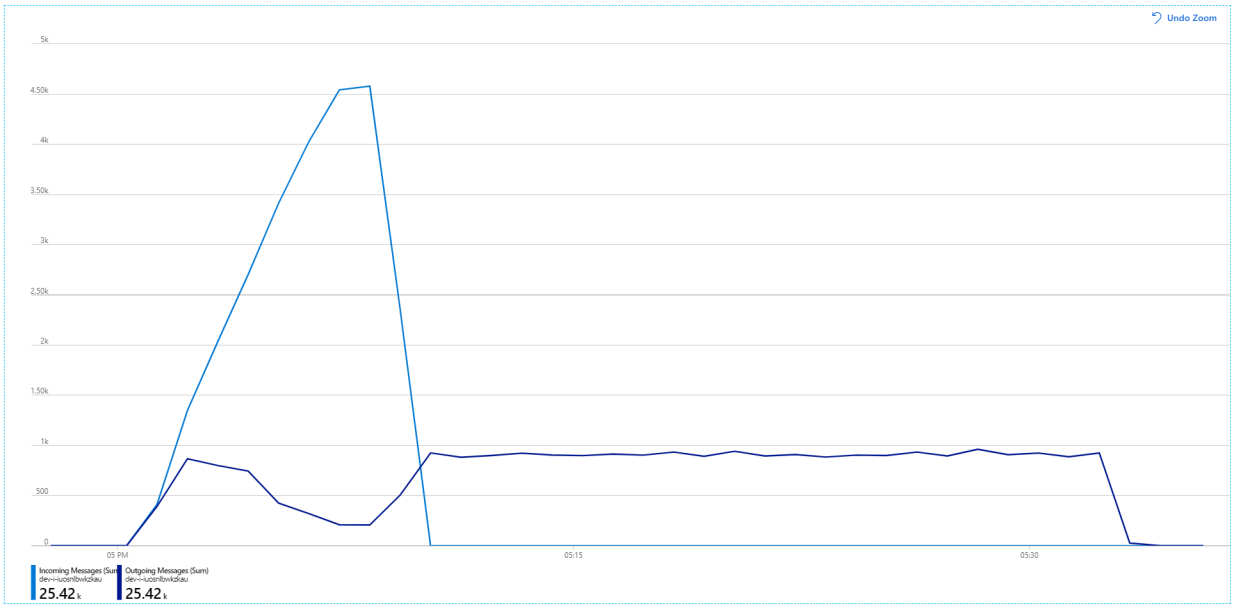
Test 3: Scale out the backend services

It appears the back end is the bottleneck. An easy next step is to scale out the business services (Package, Delivery, and Drone Scheduler), and see if throughput improves. For the next load test, the team scaled these services out from three replicas to six replicas.

Expand table

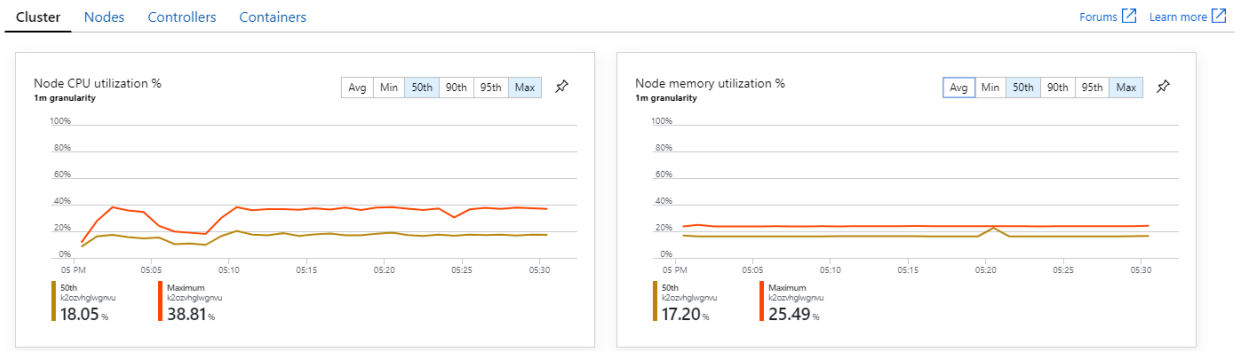
Setting	Value
Cluster nodes	6
Ingestion service	3 replicas
Workflow service	3 replicas
Package, Delivery, Drone Scheduler services	6 replicas each

Unfortunately this load test shows only modest improvement. Outgoing messages are still not keeping up with incoming messages:

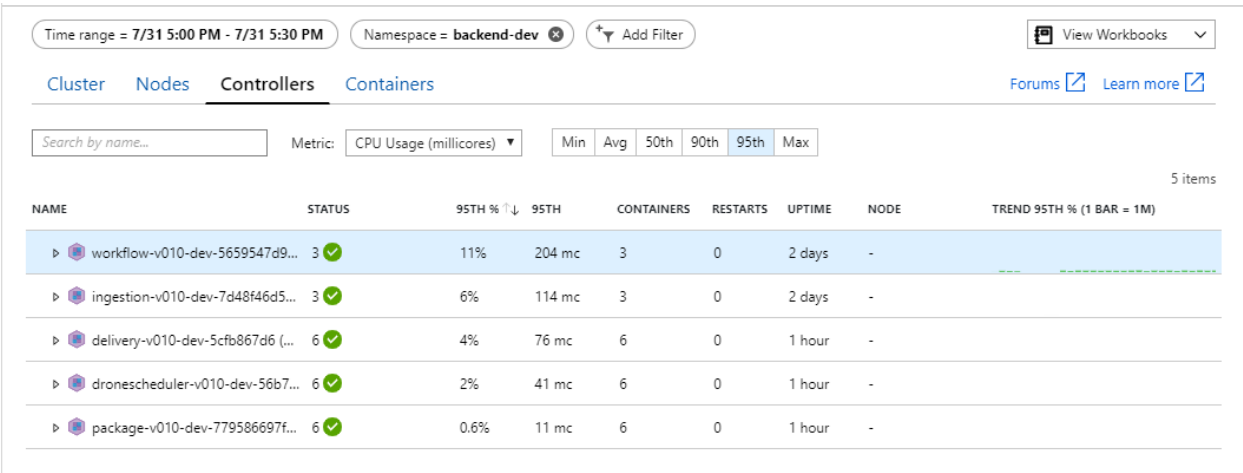


Throughput is more consistent, but the maximum achieved is about the same as the previous test:

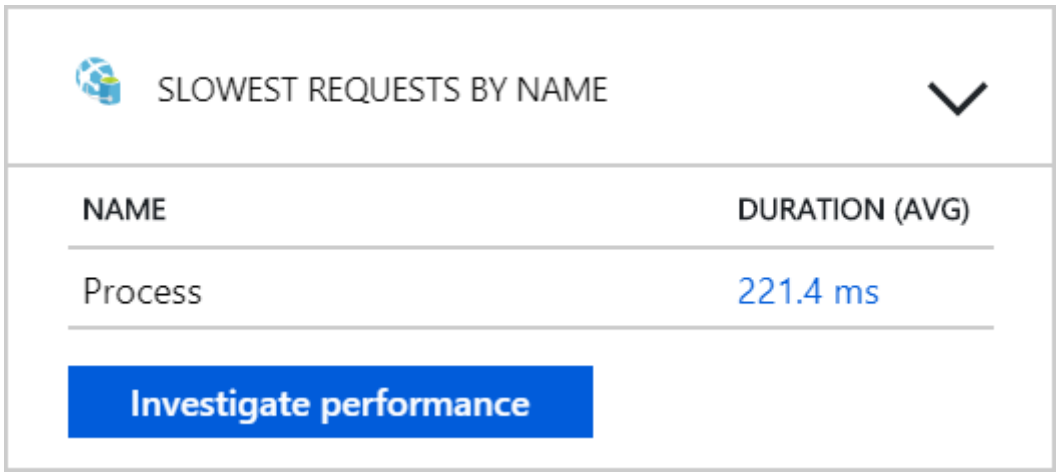
Moreover, looking at Azure Monitor container insights, it appears the problem is not caused by resource exhaustion within the cluster. First, the node-level metrics show that CPU utilization remains under 40% even at the 95th percentile, and memory utilization is about 20%.



In a Kubernetes environment, it's possible for individual pods to be resource-constrained even when the nodes aren't. But the pod-level view shows that all pods are healthy.



From this test, it seems that just adding more pods to the back end won't help. The next step is to look more closely at the Workflow service to understand what's happening when it processes messages. Application Insights shows that the average duration of the Workflow service's `Process` operation is 246 ms.



We can also run a query to get metrics on the individual operations within each transaction:

Expand table

target	percentile_duration_50	percentile_duration_95
`https://dev-i-iuosnlbwkzkau.servicebus.windows.net/ | dev-i-iuosnlbwkzkau`	86.66950203	283.4255578
delivery	37	57
package	12	17
dronescheduler	21	41

The first row in this table represents the Service Bus queue. The other rows are the calls to the backend services. For reference, here's the Log Analytics query for this table:

Kusto

```
let start=datetime("2020-07-31T22:30:00.000Z");
let end=datetime("2020-07-31T22:45:00.000Z");
let dataset=dependencies
| where timestamp > start and timestamp < end
```

```
| where (cloud_RoleName == 'fabrikam-workflow')
| where name == 'Complete' or target in ('package', 'delivery',
'dronescheduler');
dataset
| summarize percentiles(duration, 50, 95) by target
```

```
let start=datetime("2019-08-01T00:00:00.000Z");
let end=datetime("2019-08-01T00:30:00.000Z");
let dataset=dependencies
| where timestamp > start and timestamp < end
| where (cloud_RoleName == 'fabrikam-workflow')
| where name == 'Complete' or target in ('package', 'delivery', 'dronescheduler');
dataset
| summarize percentiles(duration, 50, 95) by target
```

Completed 00:00:03.200 4 records

TABLE CHART Columns

Drag a column header and drop it here to group by that column

target	percentile_duration_50	percentile_duration_95
delivery	37	57
https://dev-i-iuosnlbwkzkau.servicebus.windows.net/ | dev-i-iuosnlbwkzkau	86.6695020315	283.4255577965
package	12	17
dronescheduler	21	41

These latencies look reasonable. But here is the key insight: If the total operation time is ~250 ms, that puts a strict upper bound on how fast messages can be processed in serial. The key to improving throughput, therefore, is greater parallelism.

That should be possible in this scenario, for two reasons:

- These are network calls, so most of the time is spent waiting for I/O completion
- The messages are independent, and don't need to be processed in order.

Test 4: Increase parallelism

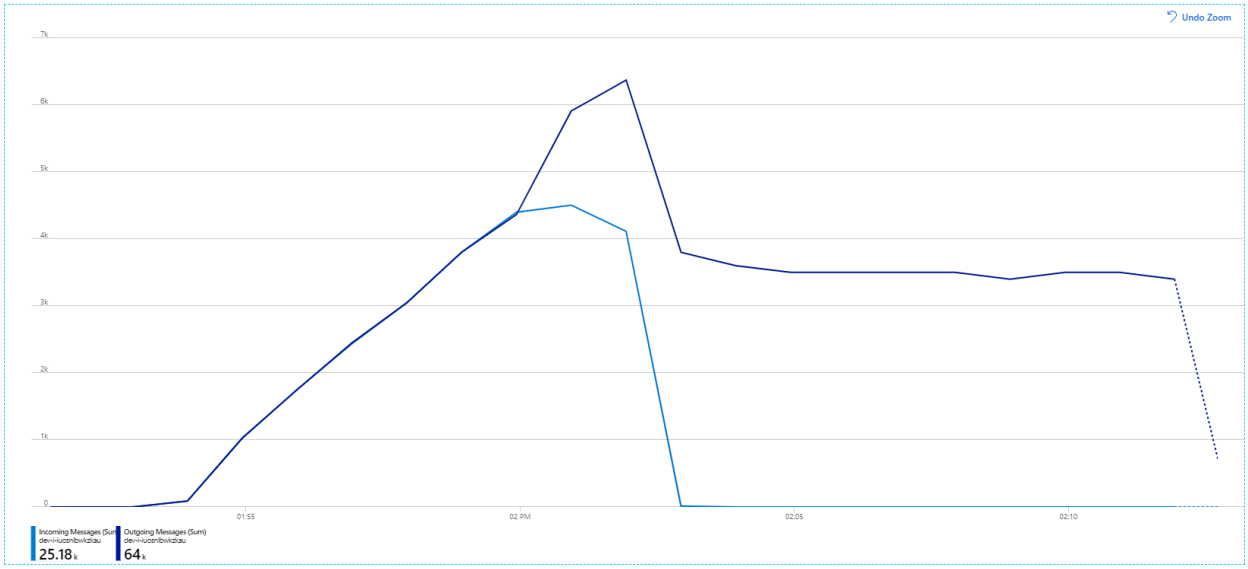
For this test, the team focused on increasing parallelism. To do so, they adjusted two settings on the Service Bus client used by the Workflow service:

Expand table

Setting	Description	Default	New value
`MaxConcurrentCalls`	The maximum number of messages to process concurrently.	1	20
`PrefetchCount`	How many messages the client will	0	3000

Setting	Description	Default	New value
	fetch ahead of time into its local cache.		

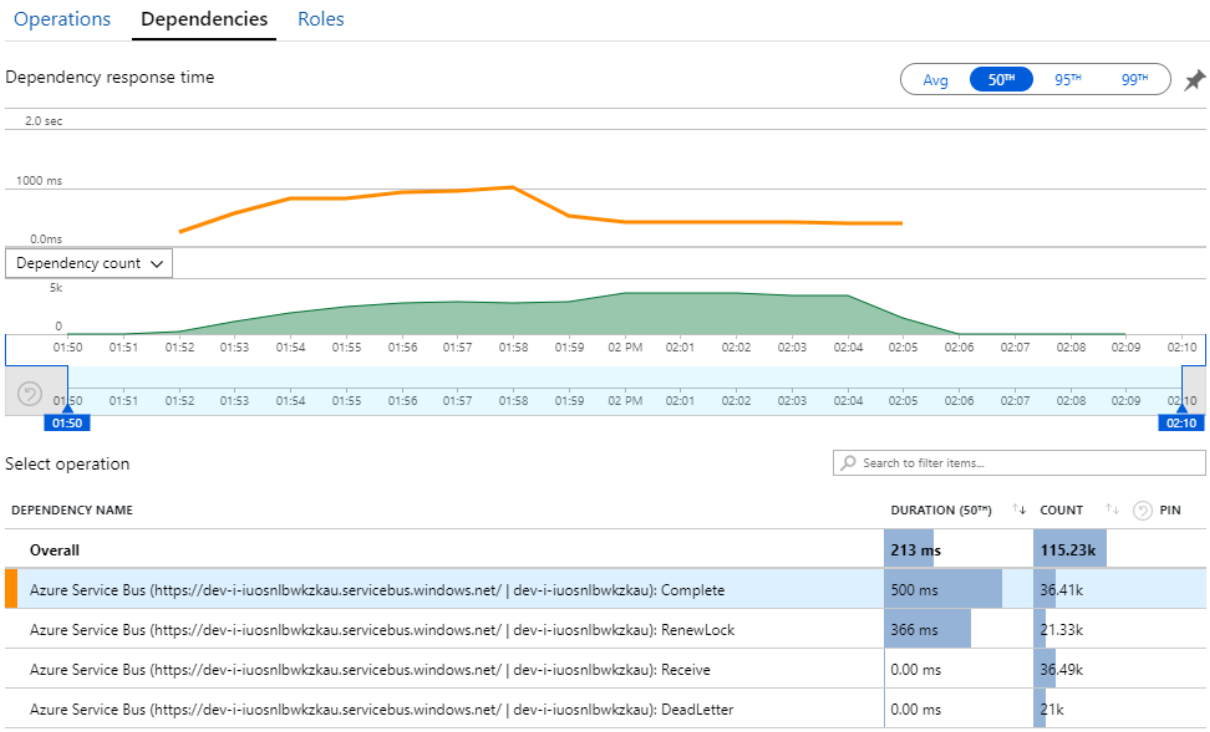
For more information about these settings, see Best Practices for performance improvements using Service Bus Messaging. Running the test with these settings produced the following graph:



Recall that incoming messages are shown in light blue, and outgoing messages are shown in dark blue.

At first glance, this is a very weird graph. For a while, the outgoing message rate exactly tracks the incoming rate. But then, at about the 2:03 mark, the rate of incoming messages levels off, while the number of outgoing messages continues to rise, actually exceeding the total number of incoming messages. That seems impossible.

The clue to this mystery can be found in the **Dependencies** view in Application Insights. This chart summarizes all of the calls that the Workflow service made to Service Bus:

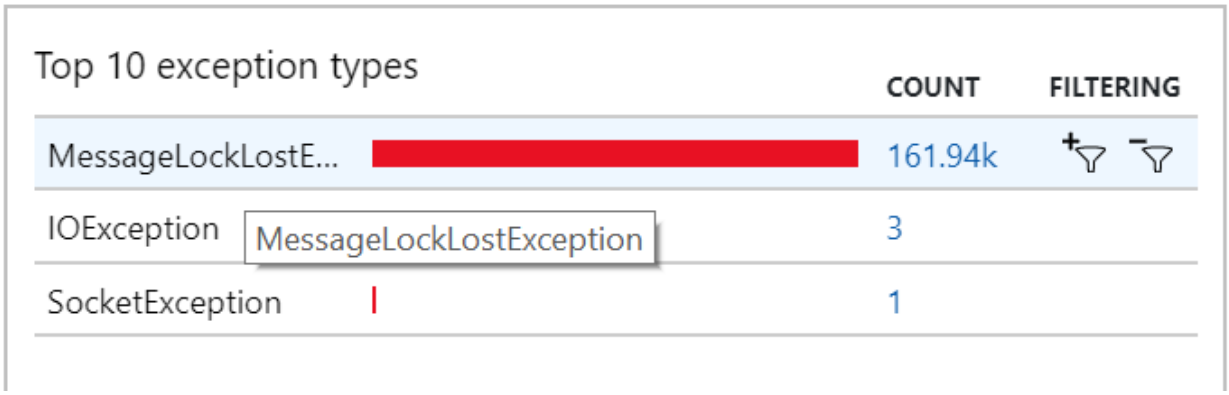


Notice that entry for `DeadLetter`. That calls indicates messages are going into the Service Bus dead-letter queue.

To understand what's happening, you need to understand Peek-Lock semantics in Service Bus. When a client uses Peek-Lock, Service Bus atomically retrieves and locks a message. While the lock is held, the message is guaranteed not to be delivered to other receivers. If the lock expires, the message becomes available to other receivers. After a maximum number of delivery attempts (which is configurable), Service Bus will put the messages onto a dead-letter queue, where it can be examined later.

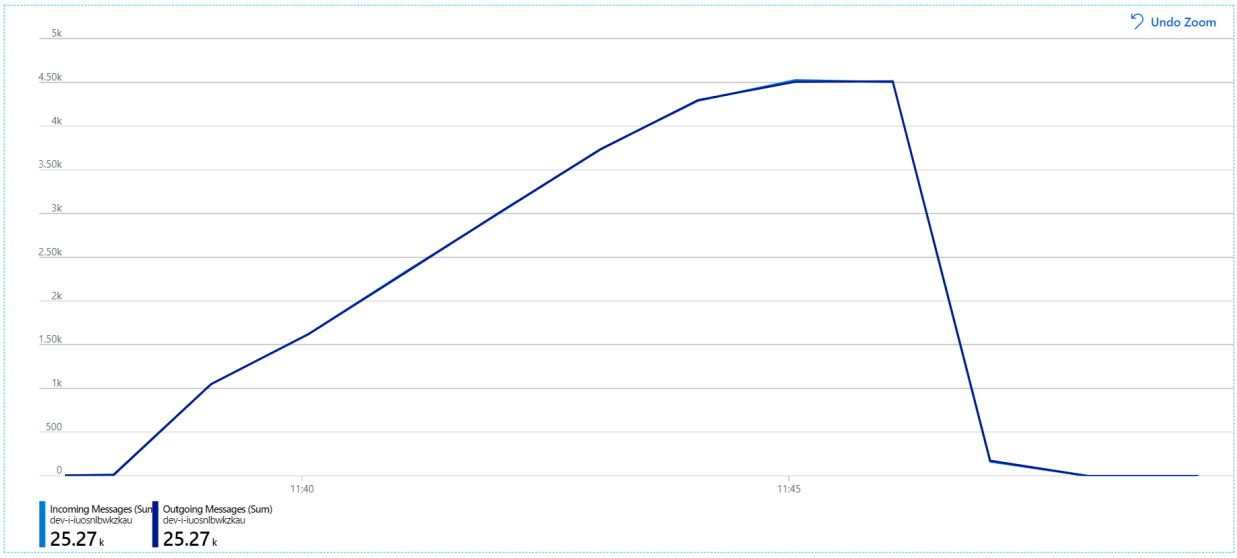
Remember that the Workflow service is prefetching large batches of messages — 3000 messages at a time). That means the total time to process each message is longer, which results in messages timing out, going back onto the queue, and eventually going into the dead-letter queue.

You can also see this behavior in the exceptions, where numerous `MessageLostLockException` exceptions are recorded:

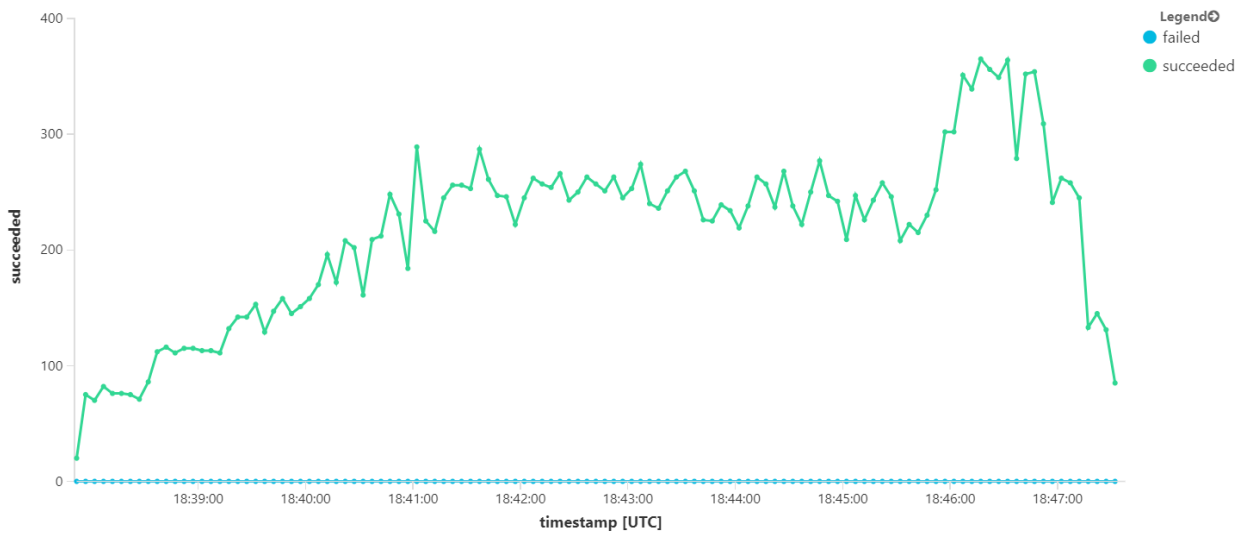


Test 5: Increase lock duration

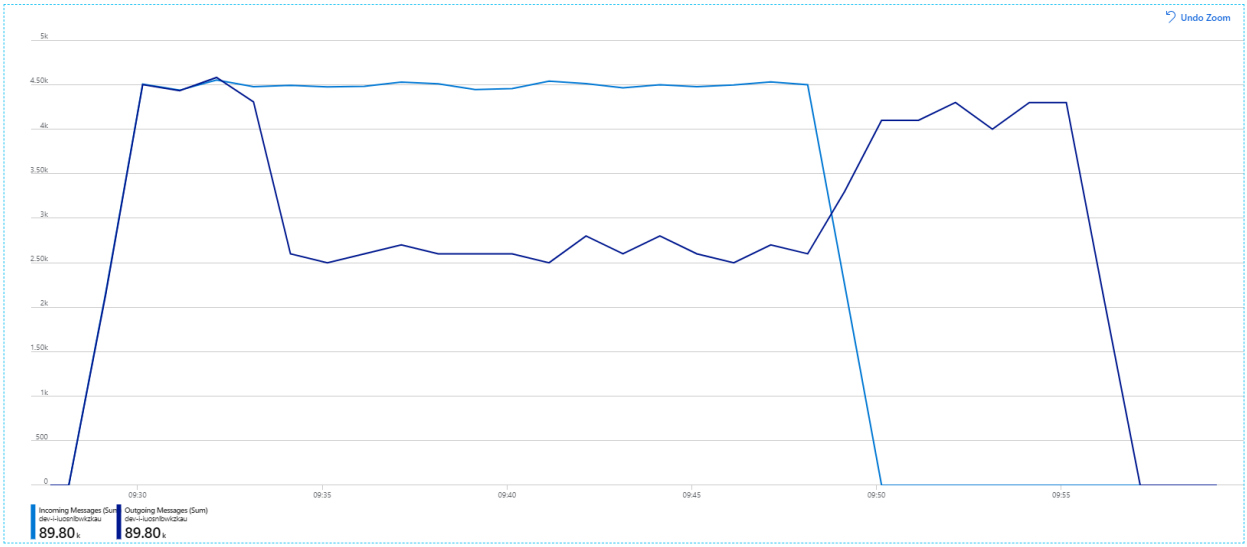
For this load test, the message lock duration was set to 5 minutes, to prevent lock timeouts. The graph of incoming and outgoing messages now shows that the system is keeping up with the rate of incoming messages:



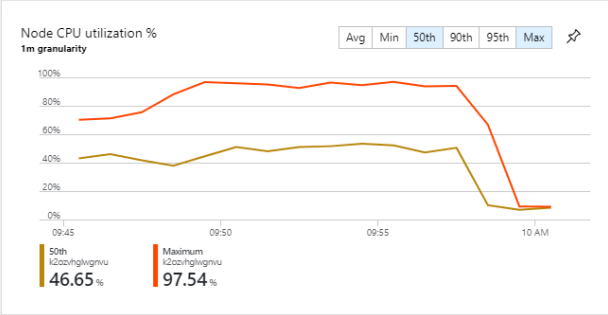
Over the total duration of the 8-minute load test, the application completed 25 K operations, with a peak throughput of 72 operations/sec, representing a 400% increase in maximum throughput.

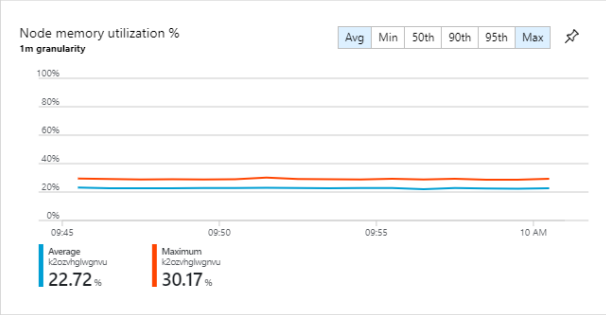


However, running the same test with a longer duration showed that the application could not sustain this rate:



The container metrics show that maximum CPU utilization was close to 100%. At this point, the application appears to be CPU-bound. Scaling the cluster might improve performance now, unlike the previous attempt at scaling out.





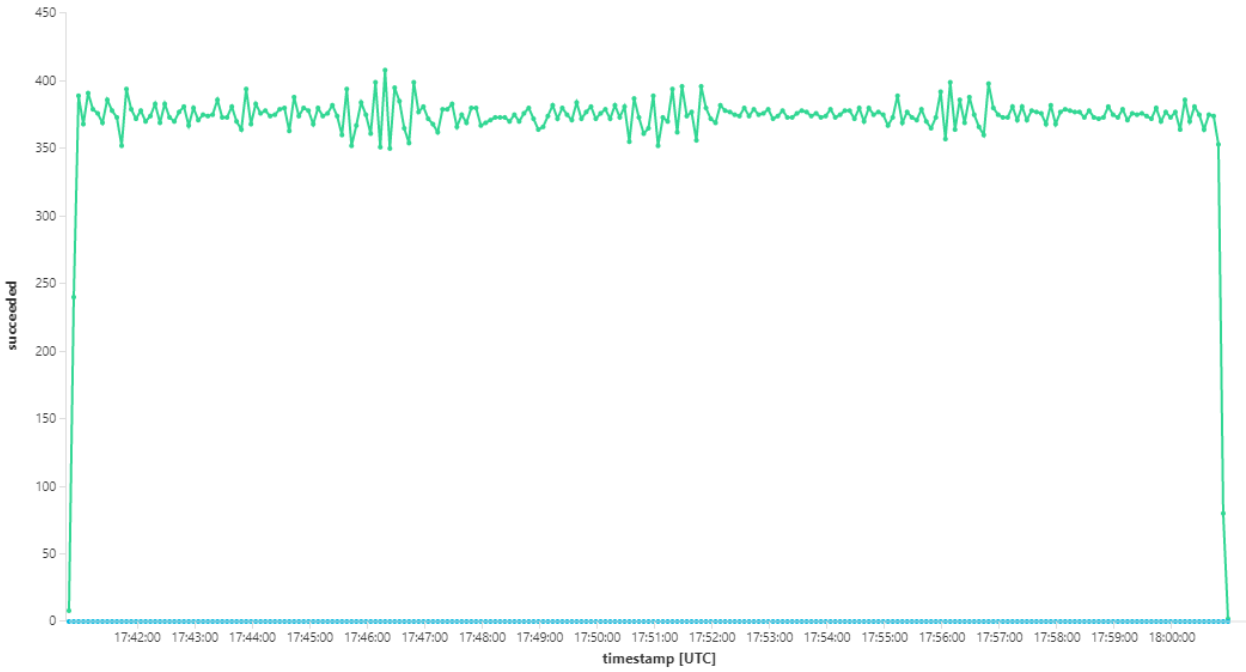
Test 6: Scale out the backend services (again)

For the final load test in the series, the team scaled out the Kubernetes cluster and pods as follows:

Expand table

Setting	Value
Cluster nodes	12
Ingestion service	3 replicas
Workflow service	6 replicas
Package, Delivery, Drone Scheduler services	9 replicas each

This test resulted in a higher sustained throughput, with no significant lags in processing messages. Moreover, node CPU utilization stayed below 80%.



Summary

For this scenario, the following bottlenecks were identified:

- Out-of-memory exceptions in Azure Cache for Redis.
- Lack of parallelism in message processing.
- Insufficient message lock duration, leading to lock timeouts and messages being placed in the dead letter queue.
- CPU exhaustion.

To diagnose these issues, the development team relied on the following metrics:

- The rate of incoming and outgoing Service Bus messages.
- Application Map in Application Insights.
- Errors and exceptions.
- Custom Log Analytics queries.
- CPU and memory utilization in Azure Monitor container insights.

Next steps

For more information about the design of this scenario, see Designing a microservices architecture.

Performance tuning - Multiple backend services

Azure Kubernetes Service (AKS) | Azure Cosmos DB

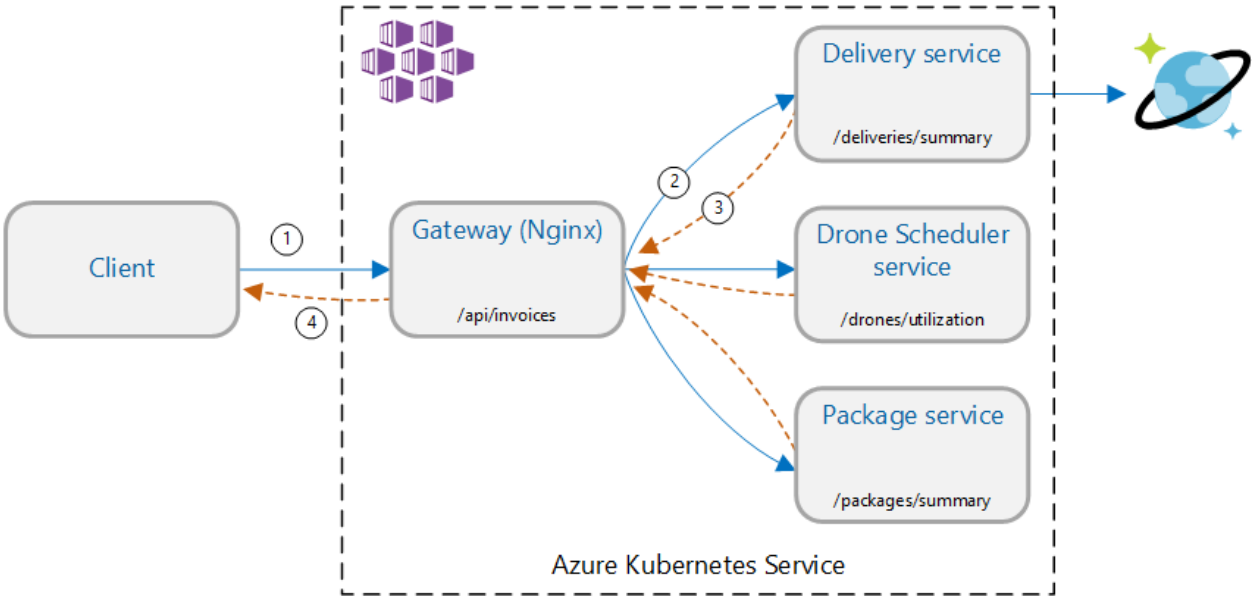
This article describes how a development team used metrics to find bottlenecks and improve the performance of a distributed system. The article is based on actual load testing that was did for a sample application. The application is from the Azure Kubernetes Service (AKS) Baseline for microservices, along with a Visual Studio load test project used to generate the results.

This article is part of a series. Read the first part here.

Scenario: Call multiple backend services to retrieve information and then aggregate the results.

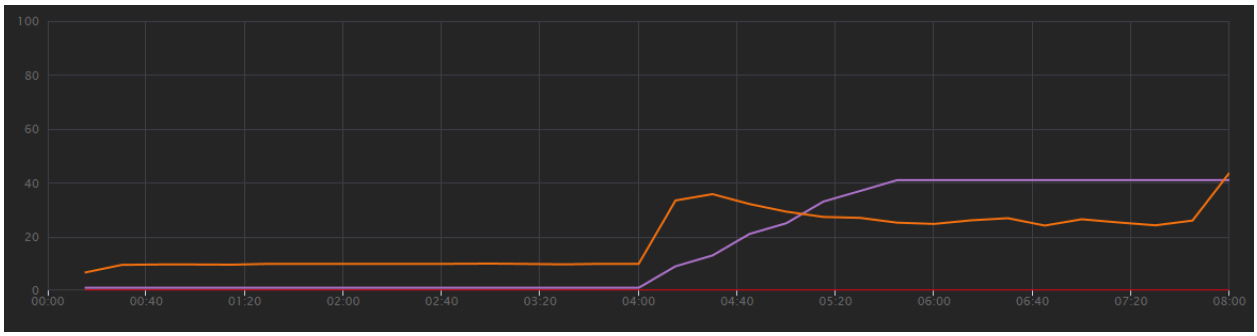
This scenario involves a drone delivery application. Clients can query a REST API to get their latest invoice information. The invoice includes a summary of the customer's deliveries, packages, and total drone utilization. This application uses a microservices architecture running on AKS, and the information needed for the invoice is spread across several microservices.

Rather than the client calling each service directly, the application implements the Gateway Aggregation pattern. Using this pattern, the client makes a single request to a gateway service. The gateway in turn calls the backend services in parallel, and then aggregates the results into a single response payload.



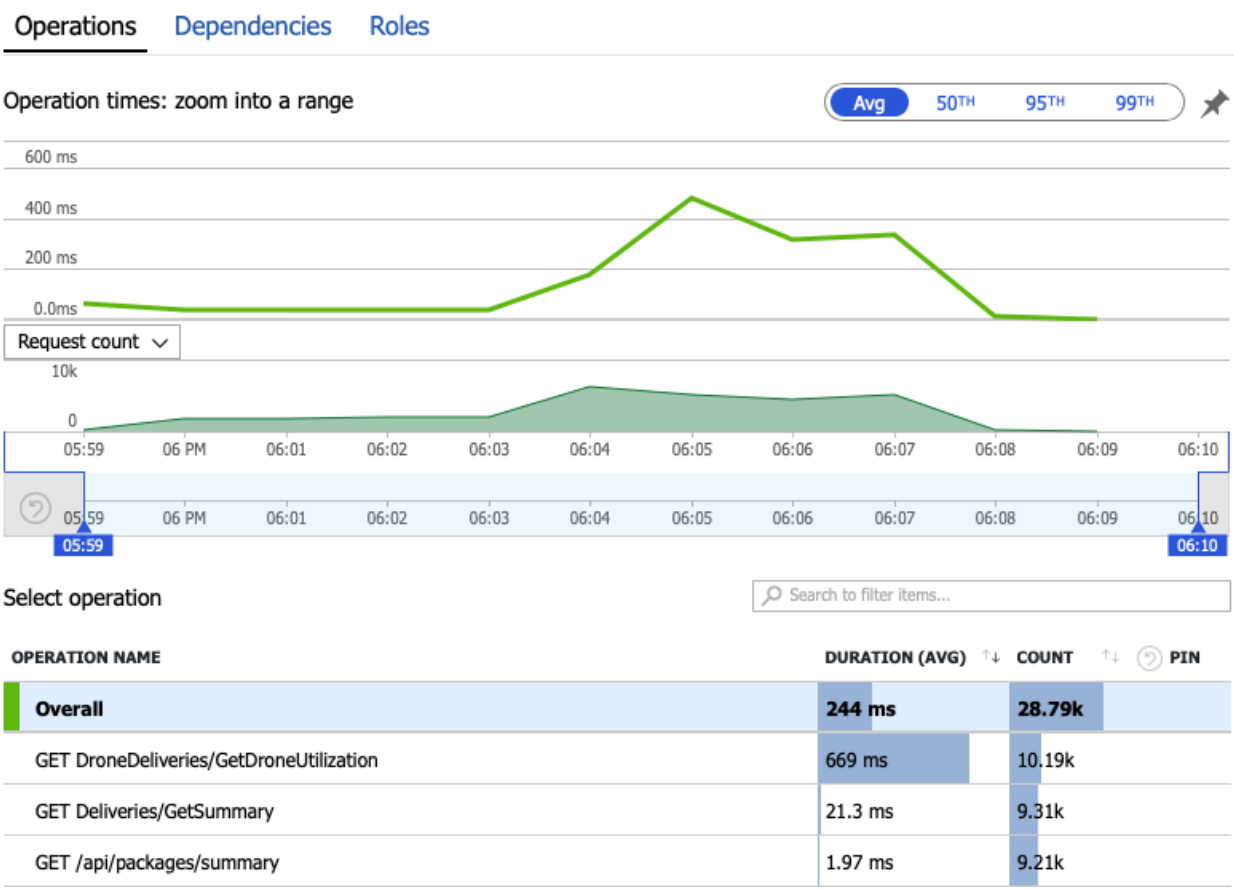
Test 1: Baseline performance

To establish a baseline, the development team started with a step-load test, ramping the load from one simulated user up to 40 users, over a total duration of 8 minutes. The following chart, taken from Visual Studio, shows the results. The purple line shows the user load, and the orange line shows throughput (average requests per second).



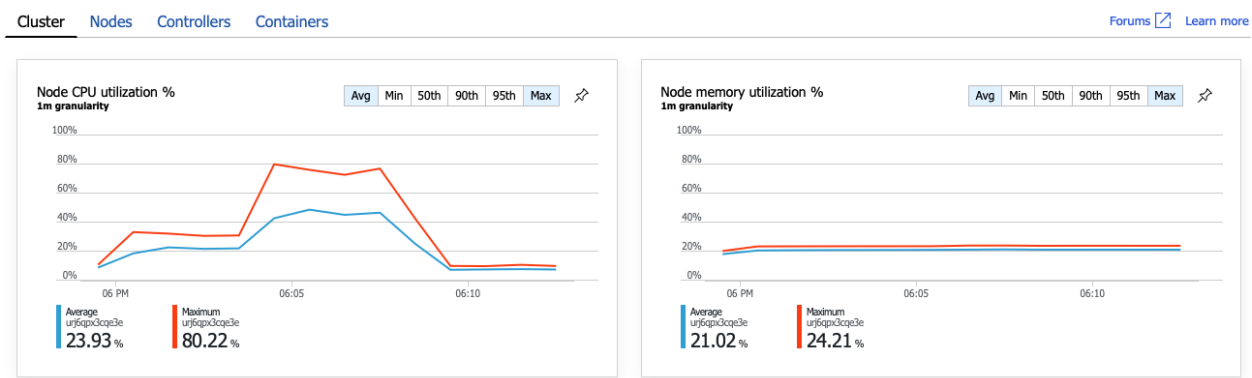
The red line along the bottom of the chart shows that no errors were returned to the client, which is encouraging. However, the average throughput peaks about half way through the test, and then drops off for the remainder, even while the load continues to increase. That indicates the back end is not able to keep up. The pattern seen here is common when a system starts to reach resource limits — after reaching a maximum, throughput actually falls significantly. Resource contention, transient errors, or an increase in the rate of exceptions can all contribute to this pattern.

Let's dig into the monitoring data to learn what's happening inside the system. The next chart is taken from Application Insights. It shows the average durations of the HTTP calls from the gateway to the backend services.



This chart shows that one operation in particular, `GetDroneUtilization`, takes much longer on average — by an order of magnitude. The gateway makes these calls in parallel, so the slowest operation determines how long it takes for the entire request to complete.

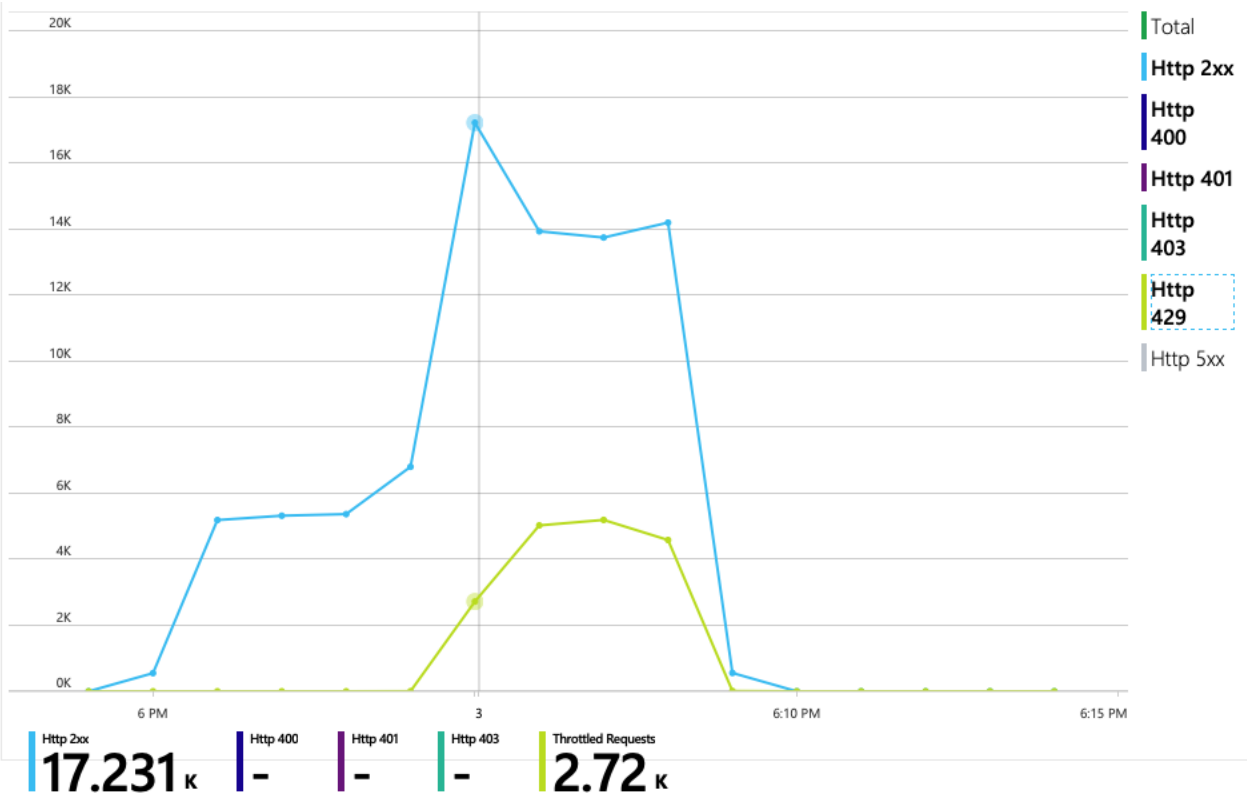
Clearly the next step is dig into the `GetDroneUtilization` operation and look for any bottlenecks. One possibility is resource exhaustion. Perhaps this particular backend service is running out of CPU or memory. For an AKS cluster, this information is available in the Azure portal through the Azure Monitor container insights feature. The following graphs show resource utilization at the cluster level:



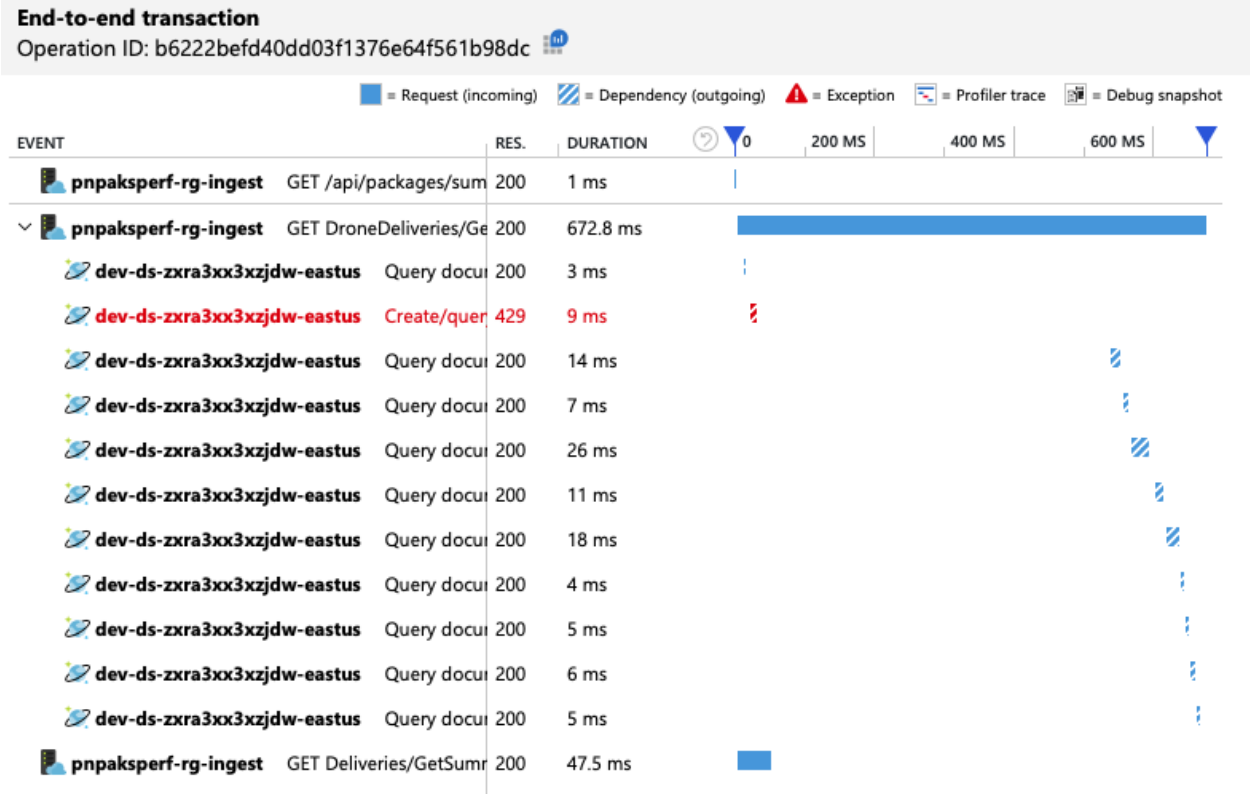
In this screenshot, both the average and maximum values are shown. It's important to look at more than just the average, because the average can hide spikes in the data. Here, the average CPU utilization stays below 50%, but there are a couple of spikes to

80%. That's close to capacity but still within tolerances. Something else is causing the bottleneck.

The next chart reveals the true culprit. This chart shows HTTP response codes from the Delivery service's backend database, which in this case is Azure Cosmos DB. The blue line represents success codes (HTTP 2xx), while the green line represents HTTP 429 errors. An HTTP 429 return code means that Azure Cosmos DB is temporarily throttling requests, because the caller is consuming more resource units (RU) than provisioned.



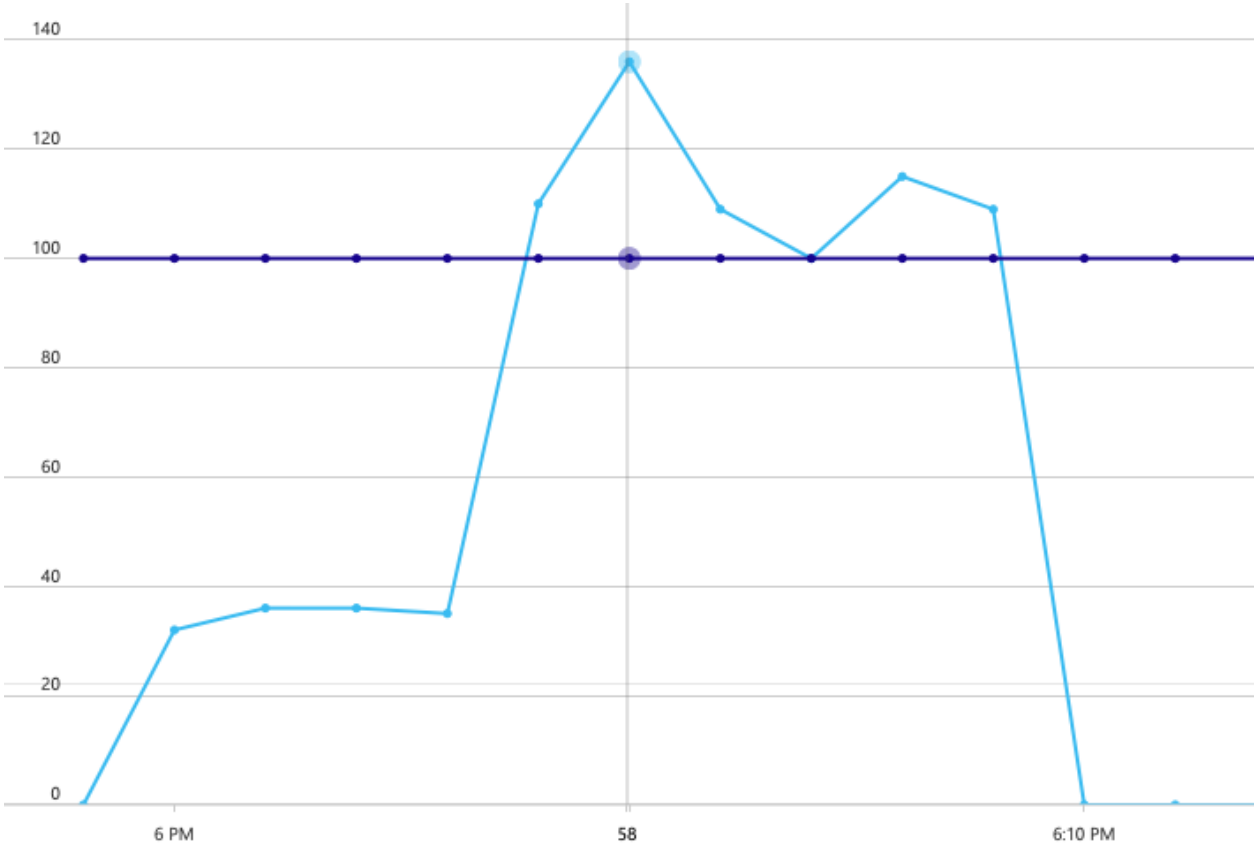
To get further insight, the development team used Application Insights to view the end-to-end telemetry for a representative sample of requests. Here is one instance:



This view shows the calls related to a single client request, along with timing information and response codes. The top-level calls are from the gateway to the backend services. The call to `GetDroneUtilization` is expanded to show calls to external dependencies — in this case, to Azure Cosmos DB. The call in red returned an HTTP 429 error.

Note the large gap between the HTTP 429 error and the next call. When the Azure Cosmos DB client library receives an HTTP 429 error, it automatically backs off and waits to retry the operation. What this view shows is that during the 672 ms this operation took, most of that time was spent waiting to retry Azure Cosmos DB.

Here's another interesting graph for this analysis. It shows RU consumption per physical partition versus provisioned RUs per physical partition:



To make sense of this graph, you need to understand how Azure Cosmos DB manages partitions. Collections in Azure Cosmos DB can have a *partition key*. Each possible key value defines a logical partition of the data within the collection. Azure Cosmos DB distributes these logical partitions across one or more *physical* partitions. The management of physical partitions is handled automatically by Azure Cosmos DB. As you store more data, Azure Cosmos DB might move logical partitions into new physical partitions, in order to spread load across the physical partitions.

For this load test, the Azure Cosmos DB collection was provisioned with 900 RUs. The chart shows 100 RU per physical partition, which implies a total of nine physical partitions. Although Azure Cosmos DB automatically handles the sharding of physical partitions, knowing the partition count can give insight into performance. The development team will use this information later, as they continue to optimize. Where the blue line crosses the purple horizontal line, RU consumption has exceeded the provisioned RUs. That's the point where Azure Cosmos DB will begin to throttle calls.

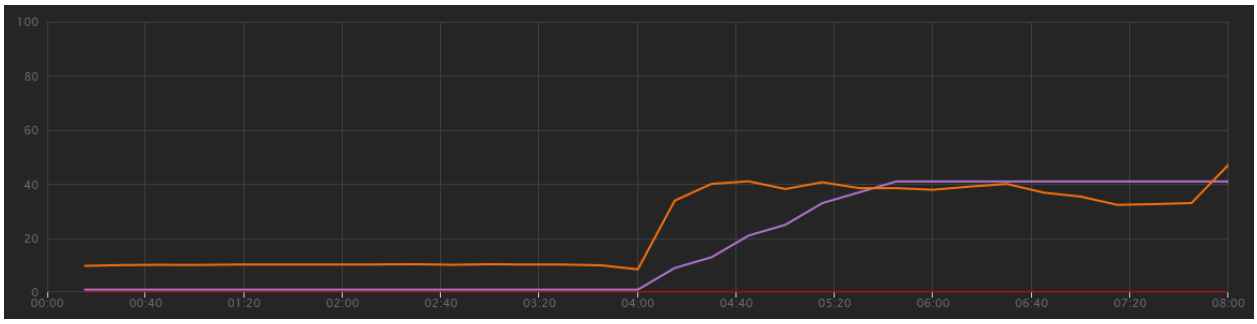
Test 2: Increase resource units

For the second load test, the team scaled out the Azure Cosmos DB collection from 900 RU to 2500 RU. Throughput increased from 19 requests/second to 23 requests/second, and average latency dropped from 669 ms to 569 ms.

Expand table

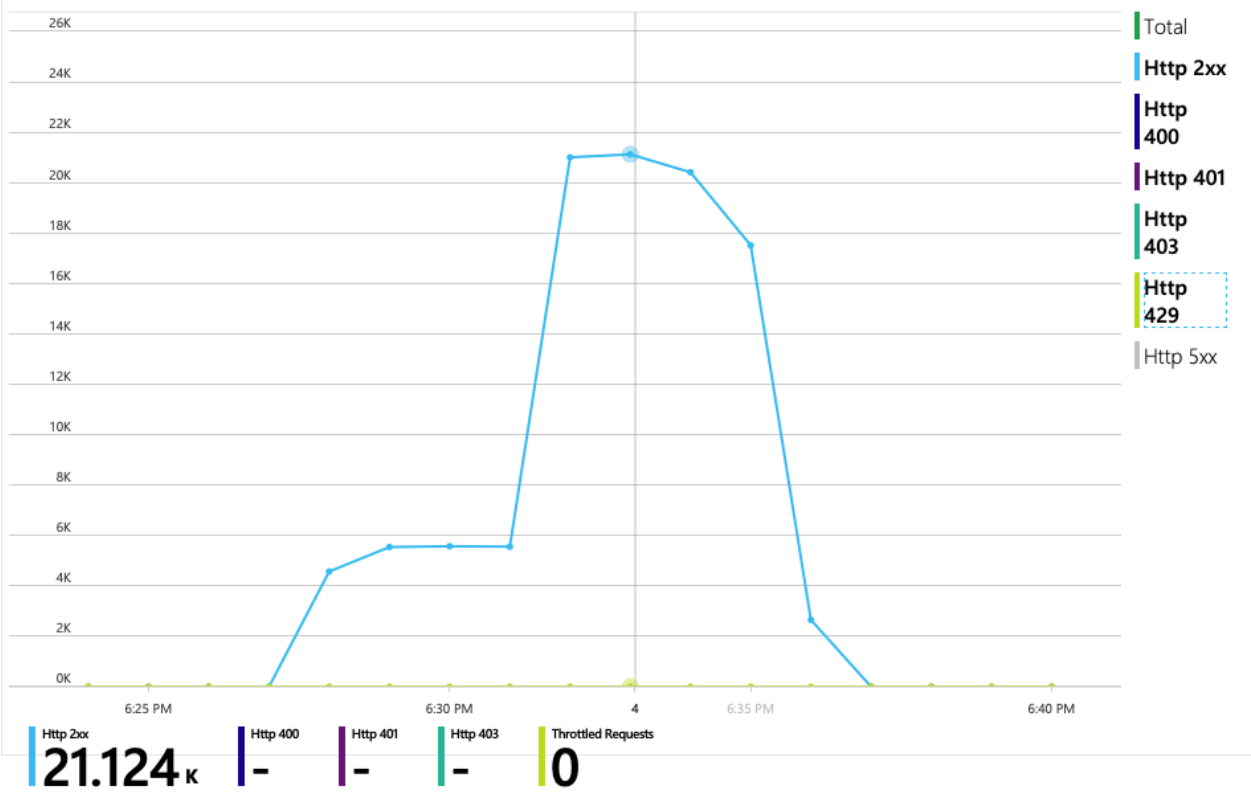
Metric	Test 1	Test 2
Throughput (req/sec)	19	23
Average latency (ms)	669	569
Successful requests	9.8 K	11 K

These aren't huge gains, but looking at the graph over time shows a more complete picture:

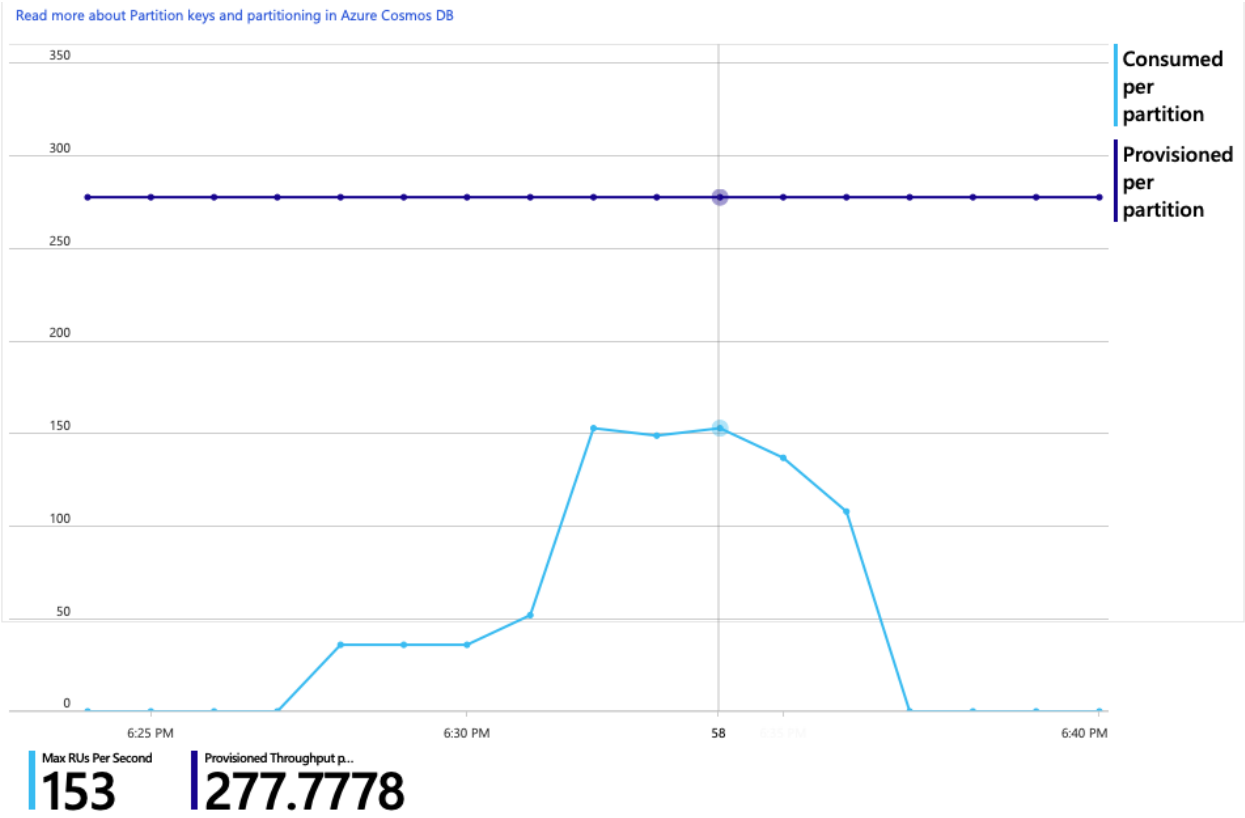


Whereas the previous test showed an initial spike followed by a sharp drop, this test shows more consistent throughput. However, the maximum throughput is not significantly higher.

All requests to Azure Cosmos DB returned a 2xx status, and the HTTP 429 errors went away:



The graph of RU consumption versus provisioned RUs shows there is plenty of headroom. There are about 275 RUs per physical partition, and the load test peaked at about 100 RUs consumed per second.



Another interesting metric is the number of calls to Azure Cosmos DB per successful operation:

Expand table

Metric	Test 1	Test 2
Calls per operation	11	9

Assuming no errors, the number of calls should match the actual query plan. In this case, the operation involves a cross-partition query that hits all nine physical partitions. The higher value in the first load test reflects the number of calls that returned a 429 error.

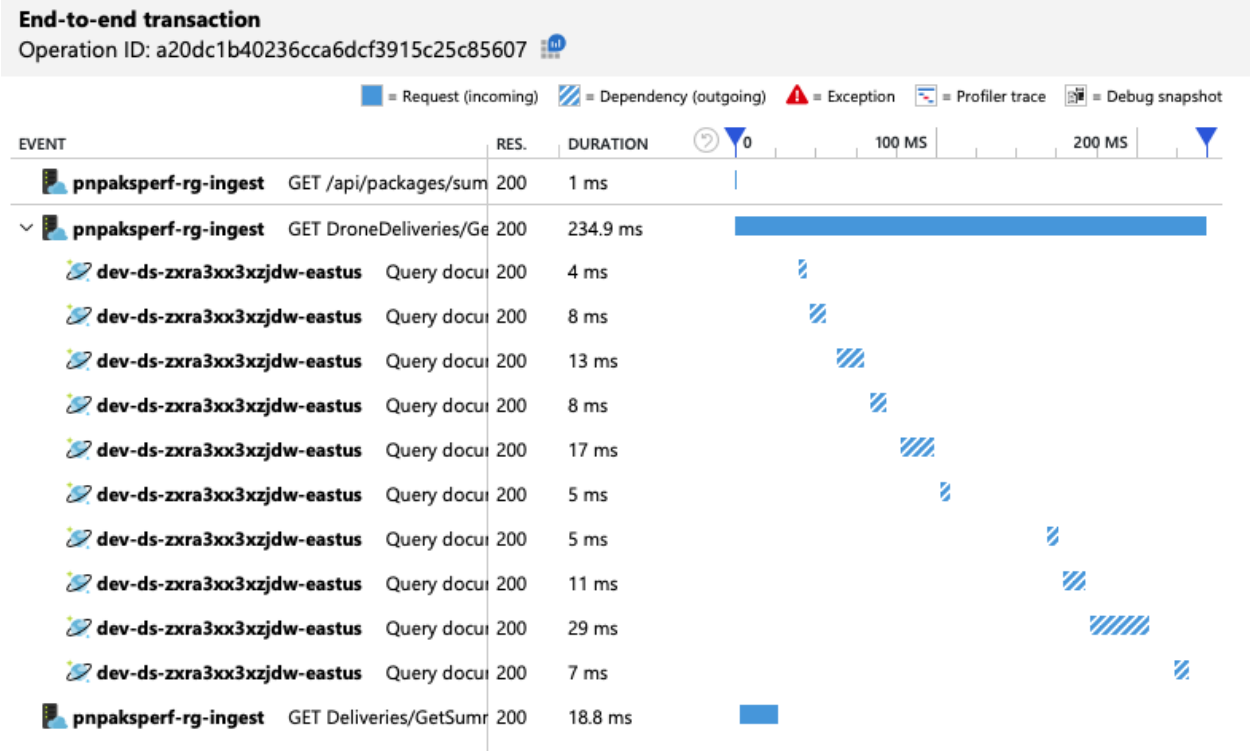
This metric was calculated by running a custom Log Analytics query:

Kusto

```
let start=datetime("2020-06-18T20:59:00.000Z");
let end=datetime("2020-07-24T21:10:00.000Z");
let operationNameToEval="GET DroneDeliveries/GetDroneUtilization";
let dependencyType="Azure DocumentDB";
```

```
let dataset=requests
| where timestamp > start and timestamp < end
| where success == true
| where name == operationNameToEval;
dataset
| project reqOk=itemCount
| summarize
    SuccessRequests=sum(reqOk),
    TotalNumberOfDepCalls=(toscalar(dependencies
    | where timestamp > start and timestamp < end
    | where type == dependencyType
    | summarize sum(itemCount)))
| project
    OperationName=operationNameToEval,
    DependencyName=dependencyType,
    SuccessRequests,
    AverageNumberOfDepCallsPerOperation=
(TotalNumberOfDepCalls/SuccessRequests)
```

To summarize, the second load test shows improvement. However, the `GetDroneUtilization` operation still takes about an order of magnitude longer than the next-slowest operation. Looking at the end-to-end transactions helps to explain why:



As mentioned earlier, the `GetDroneUtilization` operation involves a cross-partition query to Azure Cosmos DB. This means the Azure Cosmos DB client has to fan out the query to each physical partition and collect the results. As the end-to-end transaction view shows, these queries are being performed in serial. The operation takes as long as the sum of all the queries — and this problem will only get worse as the size of the data grows and more physical partitions are added.

Test 3: Parallel queries

Based on the previous results, an obvious way to reduce latency is to issue the queries in parallel. The Azure Cosmos DB client SDK has a setting that controls the maximum degree of parallelism.

Expand table

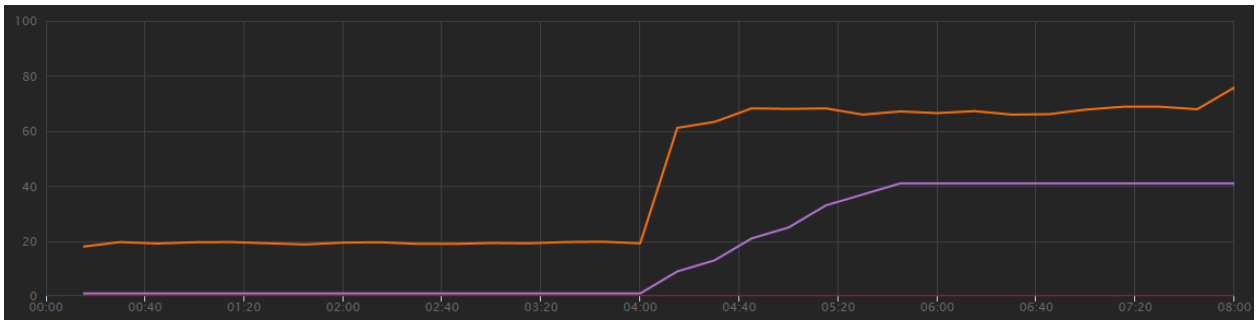
Value	Description
0	No parallelism (default)
> 0	Maximum number of parallel calls
-1	The client SDK selects an optimal degree of parallelism

For the third load test, this setting was changed from 0 to -1. The following table summarizes the results:

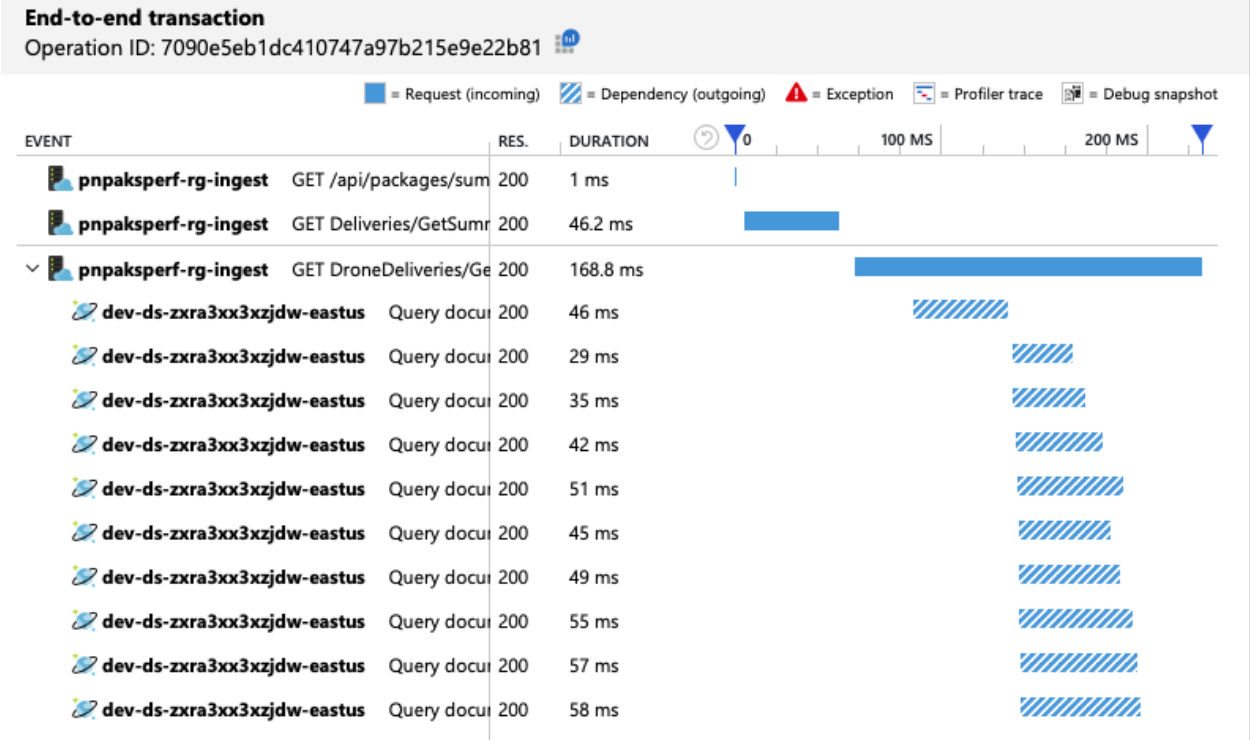
Expand table

Metric	Test 1	Test 2	Test 3
Throughput (req/sec)	19	23	42
Average latency (ms)	669	569	215
Successful requests	9.8 K	11 K	20 K
Throttled requests	2.72 K	0	0

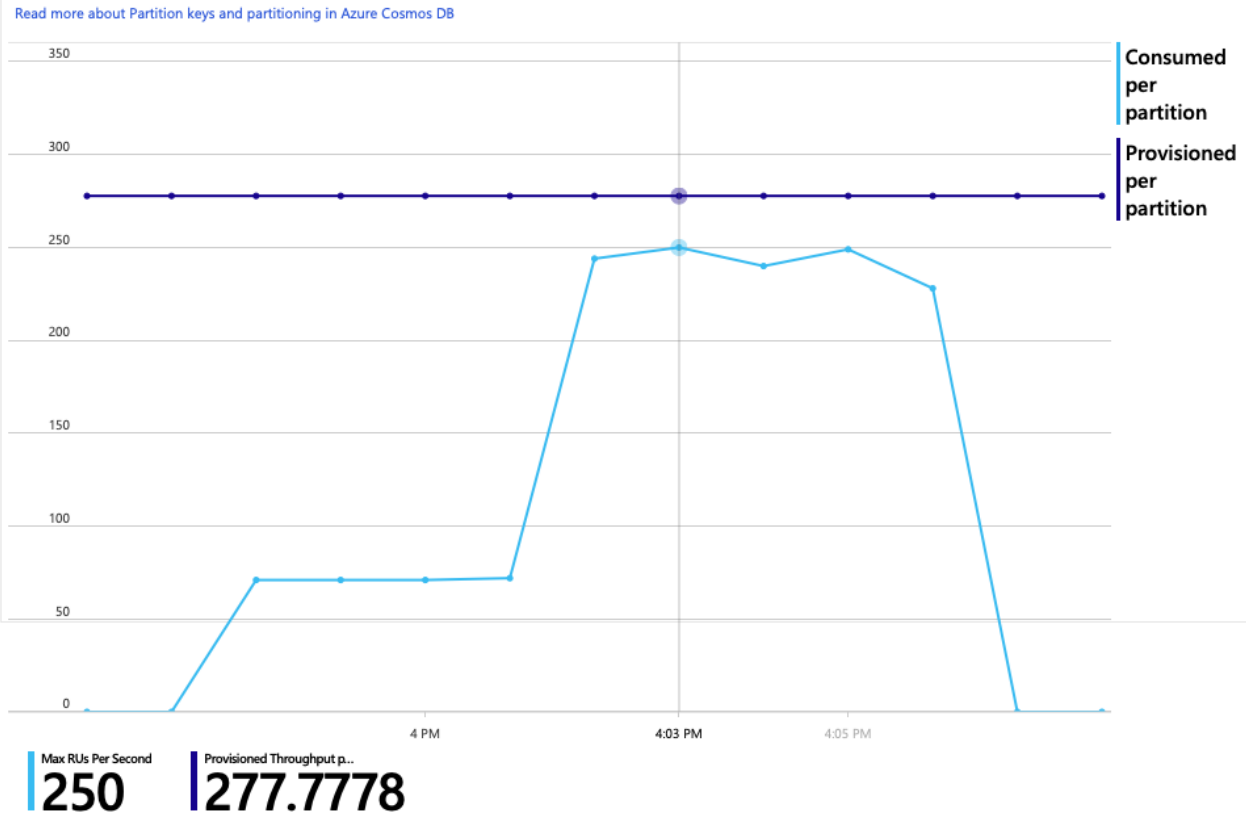
From the load test graph, not only is the overall throughput much higher (the orange line), but throughput also keeps pace with the load (the purple line).



We can verify that the Azure Cosmos DB client is making queries in parallel by looking at the end-to-end transaction view:



Interestingly, a side effect of increasing the throughput is that the number of RUs consumed per second also increases. Although Azure Cosmos DB did not throttle any requests during this test, the consumption was close to the provisioned RU limit:



This graph might be a signal to further scale out the database. However, it turns out that we can optimize the query instead.

Step 4: Optimize the query

The previous load test showed better performance in terms of latency and throughput. Average request latency was reduced by 68% and throughput increased 220%. However, the cross-partition query is a concern.

The problem with cross-partition queries is that you pay for RU across every partition. If the query is only run occasionally — say, once an hour — it might not matter. But whenever you see a read-heavy workload that involves a cross-partition query, you should see whether the query can be optimized by including a partition key. (You might need to redesign the collection to use a different partition key.)

Here's the query for this particular scenario:

SQL

```
SELECT * FROM c
WHERE c.ownerId = <ownerIdValue> and
      c.year = <yearValue> and
      c.month = <monthValue>
```

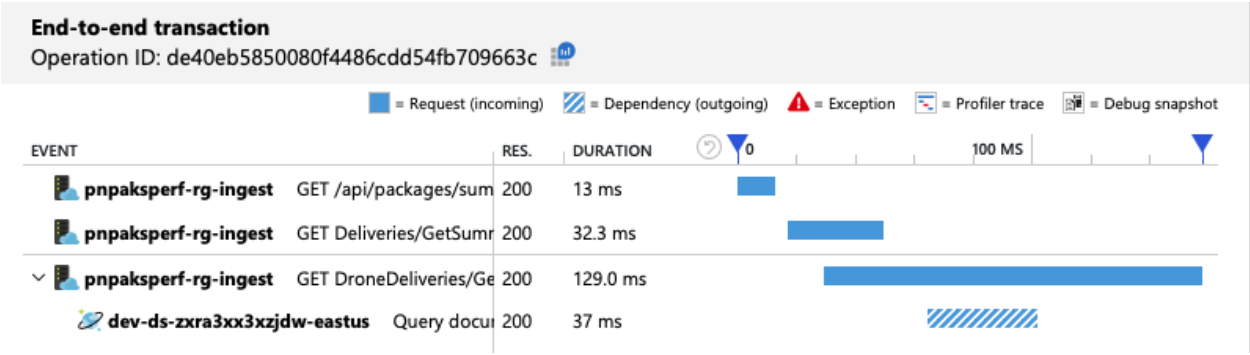
This query selects records that match a particular owner ID and month/year. In the original design, none of these properties is the partition key. That requires the client to fan out the query to each physical partition and gather the results. To improve query performance, the development team changed the design so that owner ID is the partition key for the collection. That way, the query can target a specific physical partition. (Azure Cosmos DB handles this automatically; you don't have to manage the mapping between partition key values and physical partitions.)

After switching the collection to the new partition key, there was a dramatic improvement in RU consumption, which translates directly into lower costs.

Expand table

Metric	Test 1	Test 2	Test 3	Test 4
RUs per operation	29	29	29	3.4
Calls per operation	11	9	10	1

The end-to-end transaction view shows that as predicted, the query reads only one physical partition:



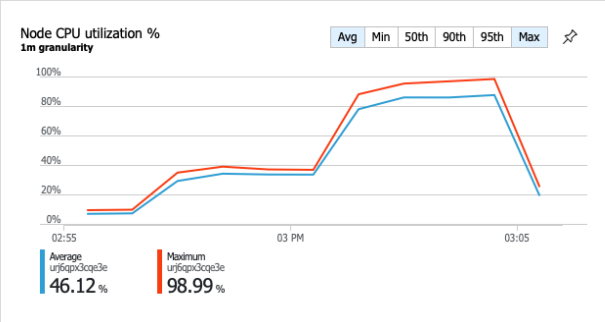
The load test shows improved throughput and latency:

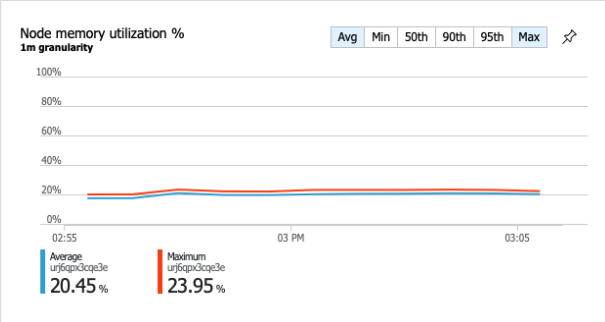
Expand table

Metric	Test 1	Test 2	Test 3	Test 4
Throughput (req/sec)	19	23	42	59
Average latency (ms)	669	569	215	176

Metric	Test 1	Test 2	Test 3	Test 4
Successful requests	9.8 K	11 K	20 K	29 K
Throttled requests	2.72 K	0	0	0

A consequence of the improved performance is that node CPU utilization becomes very high:





Toward the end of the load test, average CPU reached about 90%, and maximum CPU reached 100%. This metric indicates that CPU is the next bottleneck in the system. If higher throughput is needed, the next step might be scaling out the Delivery service to more instances.

Summary

For this scenario, the following bottlenecks were identified:

- Azure Cosmos DB throttling requests due to insufficient RUs provisioned.
- High latency caused by querying multiple database partitions in serial.
- Inefficient cross-partition query, because the query did not include the partition key.

In addition, CPU utilization was identified as a potential bottleneck at higher scale. To diagnose these issues, the development team looked at:

- Latency and throughput from the load test.
- Azure Cosmos DB errors and RU consumption.
- The end-to-end transaction view in Application Insight.
- CPU and memory utilization in Azure Monitor container insights.

Next steps

Review performance antipatterns

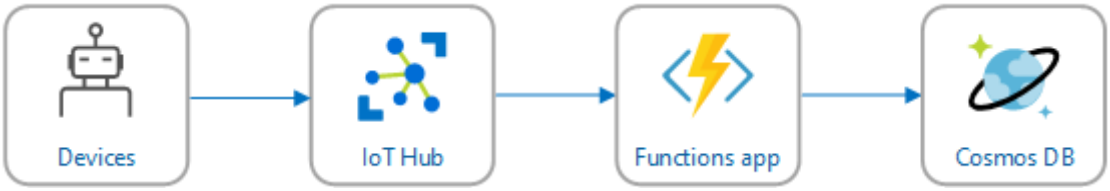
Performance tuning - Event streaming

Azure Functions | Azure IoT Hub | Azure Cosmos DB

This article describes how a development team used metrics to find bottlenecks and improve the performance of a distributed system. The article is based on actual load testing that we did for a sample application.

This article is part of a series. Read the first part here.

Scenario: Process a stream of events using Azure Functions.



In this scenario, a fleet of drones sends position data in real time to Azure IoT Hub. A Functions app receives the events, transforms the data into GeoJSON format, and writes the transformed data to Azure Cosmos DB. Azure Cosmos DB has native support for geospatial data, and Azure Cosmos DB collections can be indexed for efficient spatial queries. For example, a client application could query for all drones within 1 km of a given location, or find all drones within a certain area.

These processing requirements are simple enough that they don't require a full-fledged stream processing engine. In particular, the processing doesn't join streams, aggregate data, or process across time windows. Based on these requirements, Azure Functions is a good fit for processing the messages. Azure Cosmos DB can also scale to support very high write throughput.

Monitoring throughput

This scenario presents an interesting performance challenge. The data rate *per device* is known, but the number of devices might fluctuate. For this business scenario, the latency requirements are not particularly stringent. The reported position of a drone only needs to be accurate within a minute. That said, the function app must keep up with the average ingestion rate over time.

IoT Hub stores messages in a log stream. Incoming messages are appended to the tail of the stream. A reader of the stream — in this case, the function app — controls its own rate of traversing the stream. This decoupling of the read and write paths makes IoT Hub very efficient, but also means that a slow reader can fall behind. To detect this condition, the development team added a custom metric to measure message lateness. This metric records the delta between when a message arrives at IoT Hub, and when the function receives the message for processing.

C#

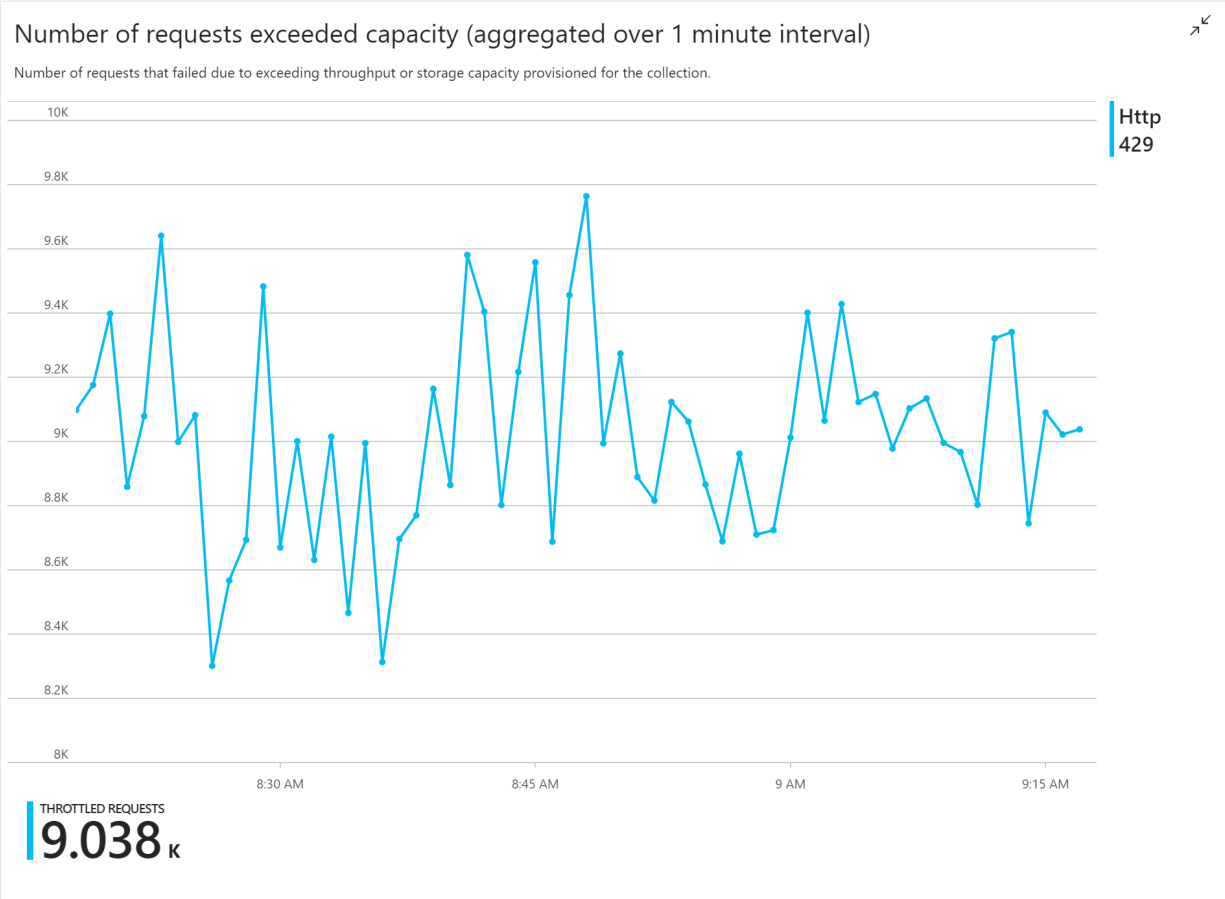
```
var ticksUTCNow = DateTimeOffset.UtcNow;

// Track whether messages are arriving at the function late.
DateTime? firstMsgEnqueuedTicksUtc = messages[0]?.EnqueuedTimeUtc;
if (firstMsgEnqueuedTicksUtc.HasValue)
{
    CustomTelemetry.TrackMetric(
                        context,
                        "IoTHubMessagesReceivedFreshnessMsec",
                        (ticksUTCNow -
firstMsgEnqueuedTicksUtc.Value).TotalMilliseconds);
}
```

The `TrackMetric` method writes a custom metric to Application Insights. For information about using `TrackMetric` inside an Azure Function, see Custom telemetry in C# function.

If the function keeps up with the volume of messages, this metric should stay at a low steady state. Some latency is unavoidable, so the value will never be zero. But if the function falls behind, the delta between enqueued time and processing time will start to go up.

Test 1: Baseline

The first load test showed an immediate problem: The Function app consistently received HTTP 429 errors from Azure Cosmos DB, indicating that Azure Cosmos DB was throttling the write requests.

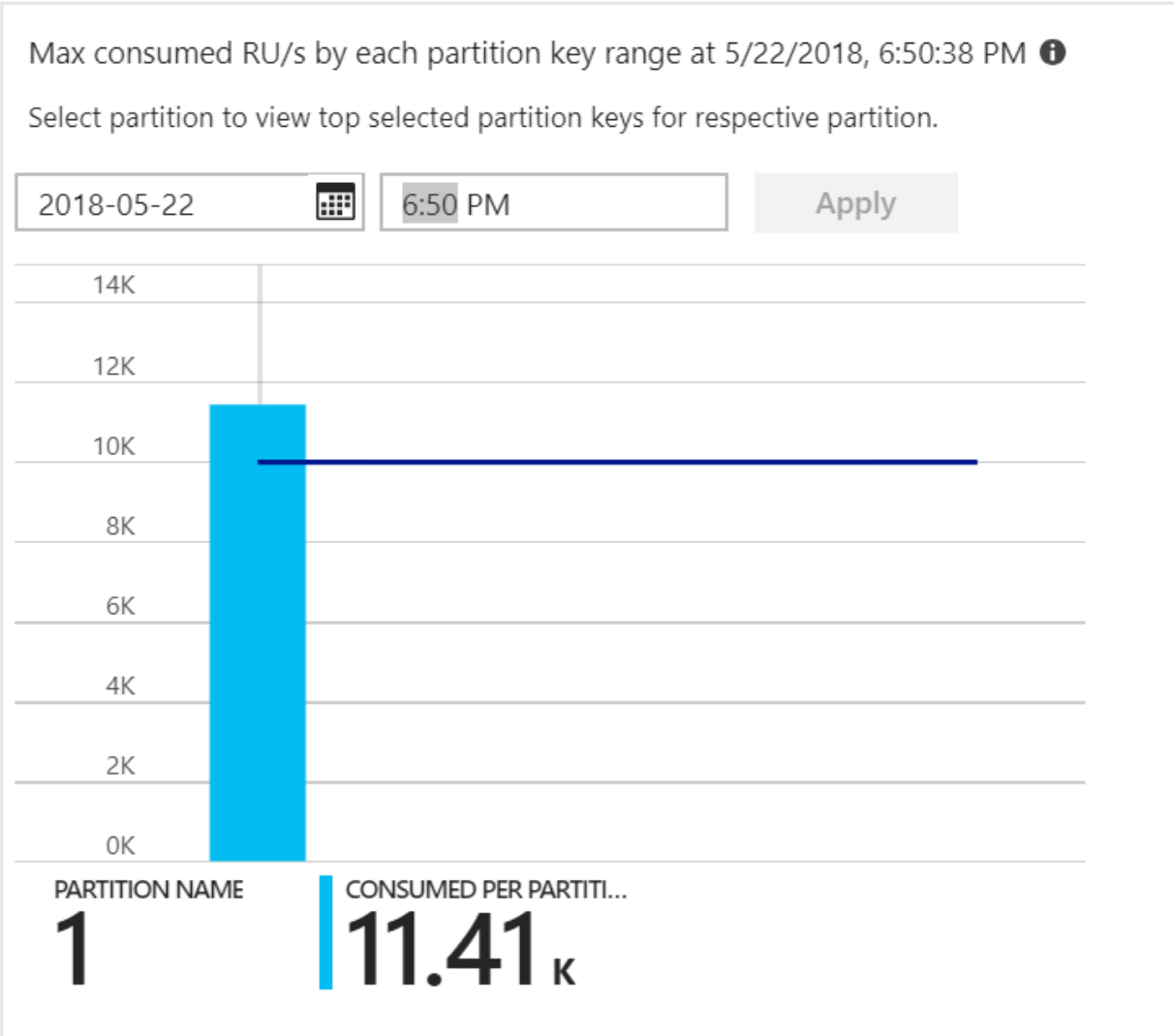


In response, the team scaled Azure Cosmos DB by increasing the number RUs allocated for the collection, but the errors continued. This seemed strange, because their back-of-envelope calculation showed that Azure Cosmos DB should have no problem keeping up with the volume of write requests.

Later that day, one of the developers sent the following email to the team:

> I looked at Azure Cosmos DB for the warm path. There's one thing I don't understand. The partition key is deliveryId, however we don't send deliveryId to Azure Cosmos DB. Am I missing something?

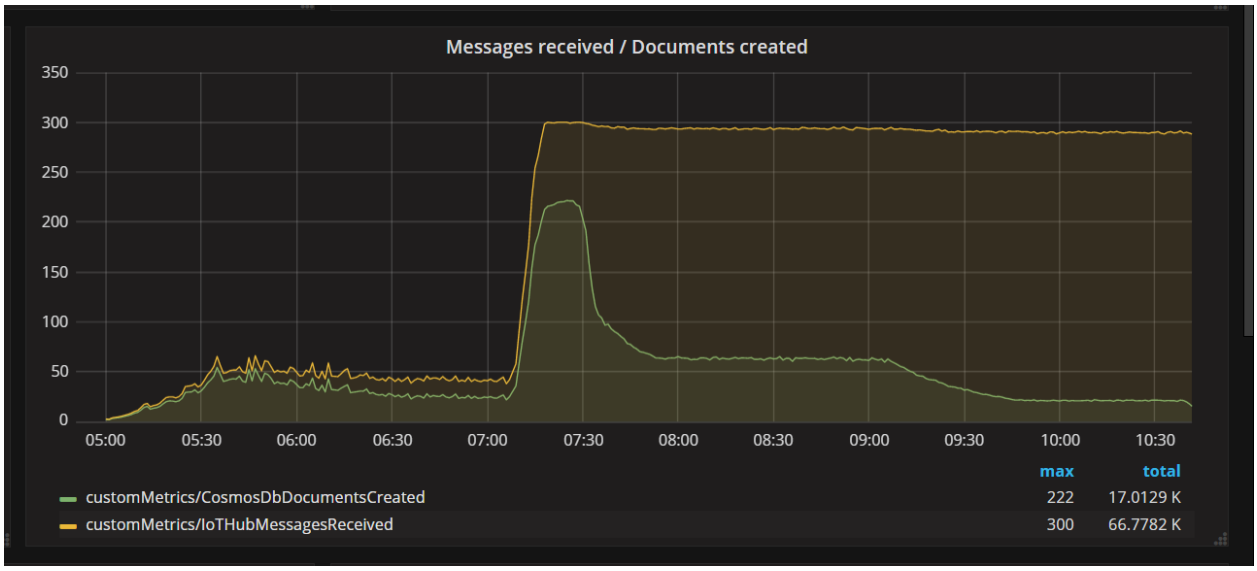
That was the clue. Looking at the partition heat map, it turned out that all of the documents were landing on the same partition.



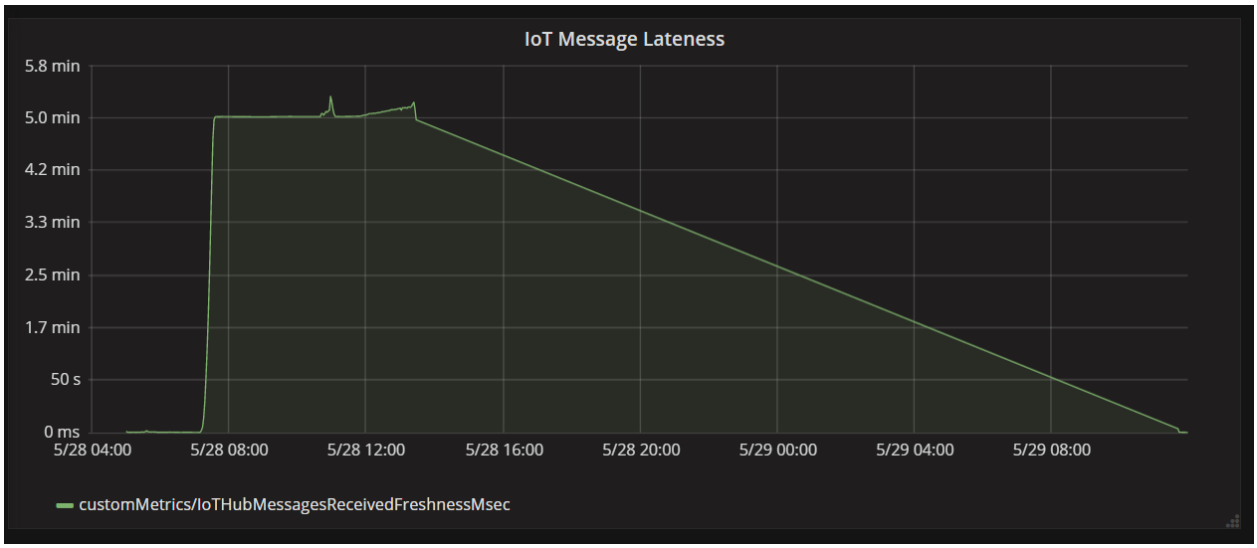
What you want to see in the heat map is an even distribution across all of the partitions. In this case, because every document was getting written to the same partition, adding RUs didn't help. The problem turned out to be a bug in the code. Although the Azure Cosmos DB collection had a partition key, the Azure Function didn't actually include the partition key in the document. For more information about the partition heat map, see Determine the throughput distribution across partitions.

Test 2: Fix partitioning issue

When the team deployed a code fix and re-ran the test, Azure Cosmos DB stopped throttling. For a while, everything looked good. But at a certain load, telemetry showed that the function was writing fewer documents that it should. The following graph shows messages receives from IoT Hub versus documents written to Azure Cosmos DB. The yellow line is number of messages received per batch, and the green is the number of documents written per batch. These should be proportional. Instead, the number of database write operations per batch drops significantly at about 07:30.



The next graph shows the latency between when a message arrives at IoT Hub from a device, and when the function app processes that message. You can see that at the same point in time, the lateness spikes dramatically, levels off, and the declines.



The reason the value peaks at 5 minutes and then drops to zero is because the function app discards messages that are more than 5 minutes late:

C#

```
foreach (var message in messages)
{
    // Drop stale messages,
    if (message.EnqueuedTimeUtc < cutoffTime)
    {
        log.Info($"Dropping late message batch. Enqueued time = {message.EnqueuedTimeUtc}, Cutoff = {cutoffTime}");
        droppedMessages++;
        continue;
    }
}
```

You can see this in the graph when the lateness metric drops back to zero. In the meantime, data has been lost, because the function was throwing away messages.

What was happening? For this particular load test, the Azure Cosmos DB collection had RUs to spare, so the bottleneck was not at the database. Rather, the problem was in the message processing loop. Simply put, the function was not writing documents quickly enough to keep up with the incoming volume of messages. Over time, it fell further and further behind.

Test 3: Parallel writes

If the time to process a message is the bottleneck, one solution is to process more messages in parallel. In this scenario:

- Increase the number of IoT Hub partitions. Each IoT Hub partition gets assigned one function instance at a time, so we would expect throughput to scale linearly with the number of partitions.
- Parallelize the document writes within the function.

To explore the second option, the team modified the function to support parallel writes. The original version of the function used the Azure Cosmos DB output binding. The optimized version calls the Azure Cosmos DB client directly and performs the writes in parallel using Task.WhenAll:

C#

```
private async Task<(long documentsUpserted,
                    long droppedMessages,
                    long cosmosDbTotalMilliseconds)>
                ProcessMessagesFromEventHub(
                    int taskCount,
                    int numberOfDocumentsToUpsertPerTask,
                    EventData[] messages,
                    TraceWriter log)
{
    DateTimeOffset cutoffTime = DateTimeOffset.UtcNow.AddMinutes(-5);

    var tasks = new List<Task>();

    for (var i = 0; i < taskCount; i++)
    {
        var docsToUpsert = messages
                            .Skip(i * numberOfDocumentsToUpsertPerTask)
                            .Take(numberOfDocumentsToUpsertPerTask);
        // client will attempt to create connections to the data
        // nodes on Azure Cosmos DB clusters on a range of port numbers
        tasks.Add(UpsertDocuments(i, docsToUpsert, cutoffTime, log));
```

```
    }

    await Task.WhenAll(tasks);

    return (this.UpsertedDocuments,
            this.DroppedMessages,
            this.CosmosDbTotalMilliseconds);
}
```

Note that race conditions are possible with approach. Suppose that two messages from the same drone happen to arrive in the same batch of messages. By writing them in parallel, the earlier message could overwrite the later message. For this particular scenario, the application can tolerate losing an occasional message. Drones send new position data every 5 seconds, so the data in Azure Cosmos DB is updated continually. In other scenarios, however, it may be important to process messages strictly in order.

After deploying this code change, the application was able to ingest more than 2500 requests/sec, using an IoT Hub with 32 partitions.

Client-side considerations

Overall client experience might be diminished by aggressive parallelization on server side. Consider using Azure Cosmos DB bulk executor library (not shown in this implementation) which significantly reduces the client-side compute resources needed to saturate the throughput allocated to an Azure Cosmos DB container. A single threaded application that writes data using the bulk import API achieves nearly ten times greater write throughput when compared to a multi-threaded application that writes data in parallel while saturating the client machine's CPU.

Summary

For this scenario, the following bottlenecks were identified:

- Hot write partition, due to a missing partition key value in the documents being written.
- Writing documents in serial per IoT Hub partition.

To diagnose these issues, the development team relied on the following metrics:

- Throttled requests in Azure Cosmos DB.
- Partition heat map — Maximum consumed RUs per partition.
- Messages received versus documents created.
- Message lateness.

Next steps

Review performance antipatterns

Advanced Azure Kubernetes Service (AKS) microservices architecture

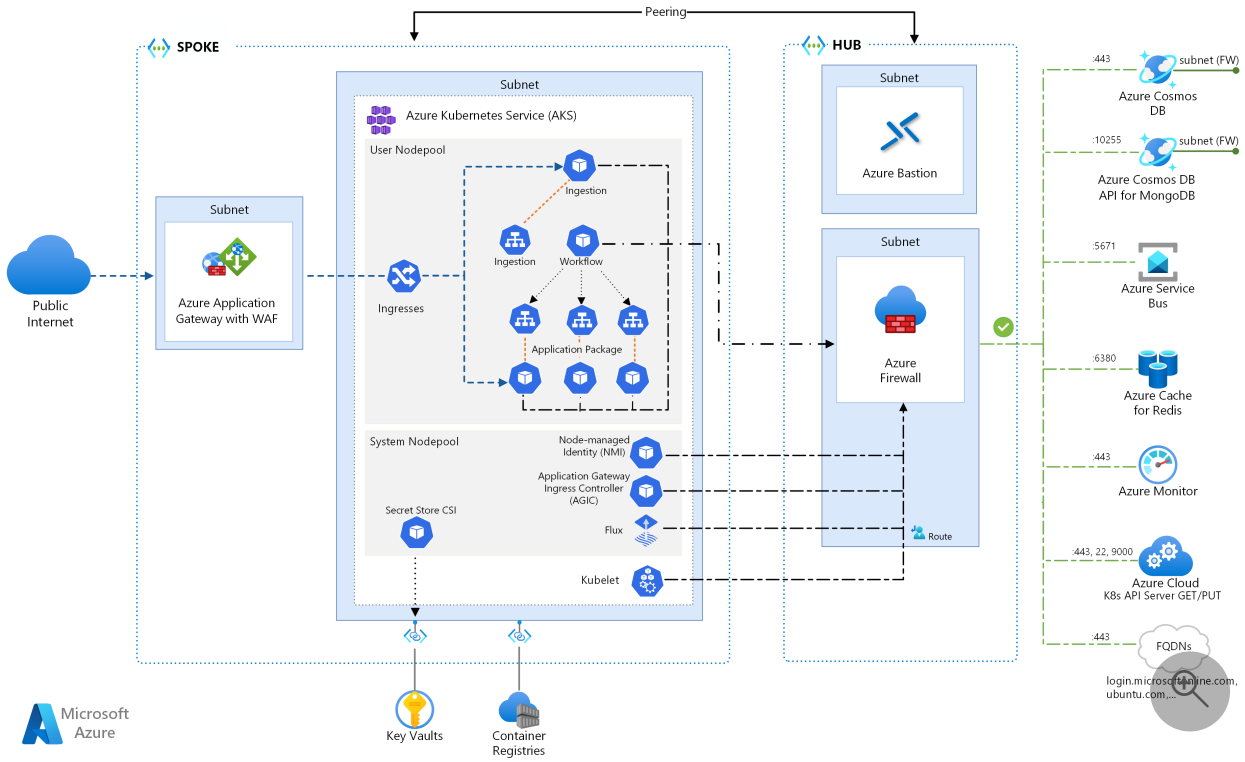
Azure Application Gateway | Azure Container Registry | Azure Kubernetes Service (AKS) | Azure Virtual Network

This reference architecture details several configurations to consider when running microservices on Azure Kubernetes Services. Topics include configuring network policies, pod autoscaling, and distributed tracing across a microservice-based application.

This architecture builds on the AKS Baseline architecture, Microsoft's recommended starting point for AKS infrastructure. The AKS baseline details infrastructural features like Microsoft Entra Workload ID, ingress and egress restrictions, resource limits, and other secure AKS infrastructure configurations. These infrastructural details are not covered in this document. It is recommended that you become familiar with the AKS baseline before proceeding with the microservices content.

A reference implementation of this architecture is available on GitHub.

Architecture



Download a Visio file of this architecture.

If you would prefer to start with a more basic microservices example on AKS, see Microservices architecture on AKS.

Workflow

This request flow implements the Publisher-Subscriber, Competing Consumers, and Gateway Routing cloud design patterns. The messaging flow proceeds as follows:

1. An HTTPS request is submitted to schedule a drone pickup. The requests pass through Azure Application Gateway into the ingestion web application, which runs as an in-cluster microservice in AKS.

2. The ingestion web application produces a message and sends it to the Service Bus message queue.

3. The backend system assigns a drone and notifies the user. The workflow:

   - Consumes message information from the Service Bus message queue.
   - Sends an HTTPS request to the Delivery microservice, which passes data to Azure Cache for Redis external data storage.
   - Sends an HTTPS request to the Drone Scheduler microservice.
   - Sends an HTTPS request to the Package microservice, which passes data to MongoDB external data storage.

4. An HTTPS GET request is used to return delivery status. This request passes through the Application Gateway into the Delivery microservice.

5. The delivery microservice reads data from Azure Cache for Redis.

Components

This architecture uses the following Azure components:

Azure Kubernetes Service is an Azure offering that provides a managed Kubernetes cluster. When using AKS, the Kubernetes API server is managed by Azure. The Kubernetes nodes or node pools are accessible and can be managed by the cluster operator.

The AKS infrastructure features used in this architecture include:

- System and user node pool separation
- AKS-managed Microsoft Entra ID for role-based access control (RBAC)
- Microsoft Entra Workload ID
- Azure Policy Add-on for AKS

- Azure Container Networking Interface (CNI)
- Azure Monitor container insights

Azure Virtual Networks are isolated and highly secure environments for running virtual machines (VMs) and applications. This reference architecture uses a peered hub-spoke virtual network topology. The hub virtual network holds the Azure firewall and Azure Bastion subnets. The spoke virtual network holds the AKS system and user node pool subnets and the Azure Application Gateway subnet.

Azure Private Link allocates specific private IP addresses to access Azure Container Registry and Key Vault from Private Endpoints within the AKS system and user node pool subnet.

Azure Application Gateway with web application firewall (WAF) exposes HTTP(S) routes to the AKS cluster and load balances web traffic to the web application. This architecture uses Azure Application Gateway Ingress Controller (AGIC) as the Kubernetes ingress controller.

Azure Bastion provides secure remote desktop protocol (RDP) and secure shell (SSH) access to VMs in the virtual networks by using a secure socket layer (SSL), without the need to expose the VMs through public IP addresses.

Azure Firewall is a network security service that protects all the Azure Virtual Network resources. The firewall allows only approved services and fully qualified domain names (FQDNs) as egress traffic.

External storage and other components:

Azure Key Vault stores and manages security keys for AKS services.

Azure Container Registry stores private container images that can be run in the AKS cluster. AKS authenticates with Container Registry using its Microsoft Entra managed identity. 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, including application telemetry and Azure platform and service metrics. You can use this data to monitor the application, set up alerts and dashboards, and perform root cause analysis of failures.

Other operations support system (OSS) components:

Helm, a package manager for Kubernetes that bundles Kubernetes objects into a single unit that you can publish, deploy, version, and update.

Azure Key Vault Secret Store CSI provider, gets secrets stored in Azure Key Vault and uses the Secret Store CSI driver interface to mount them into Kubernetes pods.

Flux, an open and extensible continuous delivery solution for Kubernetes, powered by the GitOps Toolkit.

Scenario details

The example Fabrikam Drone Delivery Shipping App shown in the preceding diagram implements the architectural components and practices discussed in this article. In this example, Fabrikam, Inc., a fictitious 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, the 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 drone's location with a continuously updated ETA.

Potential use cases

This solution is ideal for the aircraft, aerospace, and aviation industries.

Recommendations

Implement these recommendations when deploying advanced AKS microservices architectures.

Application Gateway Ingress Controller (AGIC)

API Gateway Routing is a general microservices design pattern. An API gateway acts as a reverse proxy that routes requests from clients to microservices. The Kubernetes *ingress* resource and the *ingress controller* handle most API gateway functionality by:

Routing client requests to the correct backend services provides a single endpoint for clients and help decouple clients from services.

- Aggregating multiple requests into a single request to reduce chatter between the client and the backend.
- Offloading functionality like SSL termination, authentication, IP restrictions, and client rate-limiting or throttling from the backend services.

The state of the AKS cluster is translated to Application Gateway-specific configuration and applied via Azure Resource Manager.

External ingress controllers simplify traffic ingestion into AKS clusters, improve safety and performance, and save resources. This architecture uses the Azure Application Gateway Ingress Controller (AGIC) for ingress control. Using Application Gateway to handle all traffic eliminates the need for an extra load balancer. Because pods establish direct connections against Application Gateway, the number of required hops is reduced, which results in better performance.

Application Gateway has built-in autoscaling capabilities, unlike in-cluster ingress controllers that must be scaled out if they consume an undesired amount of compute resources. Application Gateway can perform layer-7 routing and SSL termination and has end-to-end Transport Layer Security (TLS) integrated with a built-in *web application firewall (WAF)*.

For the AGIC ingress option, you must enable CNI networking when you configure the AKS cluster because Application Gateway is deployed into a subnet of the AKS virtual network. Multi-tenant workloads, or a single cluster that supports development and testing environments, could require more ingress controllers.

Zero-trust network policies

Network policies specify how AKS pods are allowed to communicate with each other and with other network endpoints. By default, all ingress and egress traffic is allowed to and from pods. When designing how your microservices communicate with each other and with other endpoints, consider following a *zero trust principle* where access to any service, device, application, or data repository requires explicit configuration.

One strategy in implementing a zero-trust policy is to create a network policy that denies all ingress and egress traffic to all pods within the target namespace. The following example shows a 'deny all policy' that would apply to all pods located in the backend-dev namespace.

YAML

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
```

```
  name: default-deny-all
  namespace: backend-dev
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

Once a restrictive policy is in place, begin to define specific network rules to allow traffic into and out of each pod in the microservice. In the following example, the network policy is applied to any pod in the backend-dev namespace with a label that matches *app.kubernetes.io/component: backend*. The policy denies any traffic unless sourced from a pod with a label that matches *app.kubernetes.io/part-of: dronedelivery*.

YAML

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: package-v010-dev-np-allow-ingress-traffic
  namespace: backend-dev
spec:
  podSelector:
    matchLabels:
      app.kubernetes.io/component: backend
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app.kubernetes.io/part-of: dronedelivery
    ports:
    - port: 80
      protocol: TCP
```

For more information on Kubernetes network policies and additional examples of potential default policies, see Network Policies in the Kubernetes documentation.

Resource quotas

Resource quotas are a way for administrators to reserve and limit resources across a development team or project. You can set resource quotas on a namespace and use them to set limits on:

- Compute resources, such as CPU and memory, or GPUs.
- Storage resources, including the number of volumes or amount of disk space for a given storage class.

- Object count, such as the maximum number of secrets, services, or jobs that can be created.

Once the cumulative total of resource requests or limits passes the assigned quota, no further deployments are successful.

Resource quotas ensure that the total set of pods assigned to the namespace can't exceed the resource quota of the namespace. The front end can't starve the backend services for resources or vice-versa.

When you define resource quotas, all pods created in the namespace must provide limits or requests in their pod specifications. If they don't provide these values, the deployment is rejected.

The following example shows a pod spec that sets resource quota requests and limits:

```yml
requests:
  cpu: 100m
  memory: 350Mi
limits:
  cpu: 200m
  memory: 500Mi
```

For more information about resource quotas, see:

- Enforce resource quotas
- Resource quotas

Autoscaling

Kubernetes supports *autoscaling* to increase the number of pods allocated to a deployment or increase the nodes in the cluster to increase the total compute resources available. Autoscaling is a self-correcting autonomous feedback system. Although you can scale pods and nodes manually, autoscaling minimizes the chances of services becoming resource-starved at high loads. An autoscaling strategy must take both pods and nodes into account.

Cluster autoscaling

The *cluster autoscaler* (CA) scales the number of nodes. Suppose pods can't be scheduled because of resource constraints; the cluster autoscaler provisions more nodes. You define a minimum number of nodes to keep the AKS cluster and your

workloads operational and a maximum number of nodes for heavy traffic. The CA checks every few seconds for pending pods or empty nodes and scales the AKS cluster appropriately.

The following example shows the CA configuration from the ARM template:

JSON

```
"autoScalerProfile": {
    "scan-interval": "10s",
    "scale-down-delay-after-add": "10m",
    "scale-down-delay-after-delete": "20s",
    "scale-down-delay-after-failure": "3m",
    "scale-down-unneeded-time": "10m",
    "scale-down-unready-time": "20m",
    "scale-down-utilization-threshold": "0.5",
    "max-graceful-termination-sec": "600",
    "balance-similar-node-groups": "false",
    "expander": "random",
    "skip-nodes-with-local-storage": "true",
    "skip-nodes-with-system-pods": "true",
    "max-empty-bulk-delete": "10",
    "max-total-unready-percentage": "45",
    "ok-total-unready-count": "3"
},
```

The following lines in the ARM template set example minimum and maximum nodes for the CA:

JSON

```
"minCount": 2,
"maxCount": 5,
```

Pod autoscaling

The *Horizontal Pod Autoscaler (HPA)* scales pods based on observed CPU, memory, or custom metrics. To configure horizontal pod scaling, you specify target metrics and the minimum and the maximum number of replicas in the Kubernetes deployment pod spec. Load test your services to determine these numbers.

CA and HPA work well together, so enable both autoscaler options in your AKS cluster. HPA scales the application, while CA scales the infrastructure.

The following example sets resource metrics for HPA:

YAML

```yaml
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: delivery-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: delivery
  minReplicas: 2
  maxReplicas: 5
  metrics:
    - type: Resource
      resource:
        name: cpu
        targetAverageUtilization: 60
```

HPA looks at actual resources consumed or other metrics from running pods, but the CA provisions nodes for pods that aren't scheduled yet. Therefore, CA looks at the requested resources, as specified in the pod spec. Use load testing to fine-tune these values.

Health probes

Kubernetes load balances traffic to pods that match a label selector for a service. Only pods that started successfully and are healthy receive traffic. If a container crashes, Kubernetes removes the pod and schedules a replacement.

In Kubernetes, a pod can expose two types of health probe:

- The *liveness probe* tells Kubernetes whether a pod started successfully and is healthy.
- The *readiness probe* tells Kubernetes whether a pod is ready to accept requests.

The liveness probes handle pods that are still running but are unhealthy and should be recycled. For example, if a container serving HTTP requests hangs, the container doesn't crash, but it stops serving requests. The HTTP liveness probe stops responding, which informs Kubernetes to restart the pod.

Sometimes, a pod might not be ready to receive traffic, even though the pod started successfully. For example, the application running in the container might be performing initialization tasks. The readiness probe indicates whether the pod is ready to receive traffic.

Microservices should expose endpoints in their code that facilitate health probes, with delay and timeout tailored specifically to the checks they perform. The HPA formula keys almost exclusively off the Ready phase on a pod, so it's critical that health probes exist and are accurate.

Monitoring

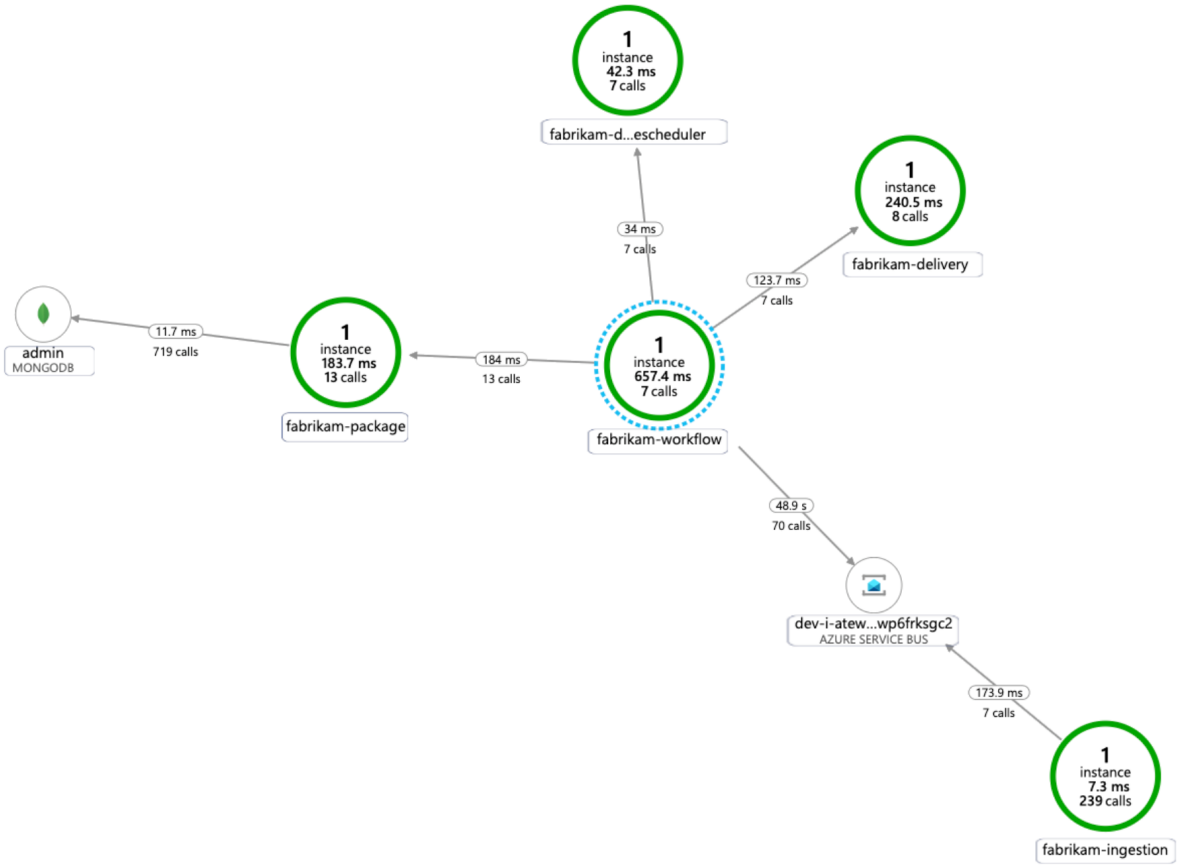
In a microservices application, *Application Performance Management (APM)* monitoring is critical for detecting anomalies, diagnosing issues, and quickly understanding the dependencies between services. Application Insights, which is part of Azure Monitor, provides APM monitoring for live applications written in .NET Core, Node.js, Java, and many other application languages.

Application Insights:

- Logs HTTP requests, including latency and result code.
- Enables distributed tracing by default.
- Includes an operation ID in traces, so you can match all traces for a particular operation.
- Often includes additional contextual information in traces.

To contextualize services telemetry with the Kubernetes world, integrate Azure Monitor telemetry with AKS to collect metrics from controllers, nodes, and containers, as well as container and node logs. If you're using .NET, the Application Insights for Kubernetes library enriches Application Insights telemetry with image, container, node, pod, label, and replica set information.

The following diagram shows an example of the application dependency map that Application Insights generates for an AKS microservices telemetry trace:



For more information on options for instrumenting common languages for application insights integration, see Application monitoring for Kubernetes.

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

Consider the following points when planning for scalability.

- Don't combine autoscaling and imperative or declarative management of the number of replicas. Users and an autoscaler both attempting to modify the number of replicas may cause unexpected behavior. When HPA is enabled, reduce the number of replicas to the minimum number you want to be deployed.

- A side-effect of pod autoscaling is that pods may be created or evicted frequently, as scale-out and scale-in events happen. To mitigate these effects:
  - Use readiness probes to let Kubernetes know when a new pod is ready to accept traffic.

    - Use pod disruption budgets to limit how many pods can be evicted from a service at a time.

- You can't change the VM size after creating a cluster, so do initial capacity planning to choose an appropriate VM size for the agent nodes when you create the cluster.

- Multi-tenant or other advanced workloads might have node pool isolation requirements that demand more and likely smaller subnets. For more information about creating node pools with unique subnets, see Add a node pool with a unique subnet. Organizations have different standards for their hub-spoke implementations. Be sure to follow your organizational guidelines.

Manageability

Consider the following points when planning for manageability.

- Manage the AKS cluster infrastructure via an automated deployment pipeline. The reference implementation for this architecture provides a GitHub Actions workflow that you can reference when building your pipeline.

- The workflow file deploys the infrastructure only, not the workload, into the already-existing virtual network and Microsoft Entra configuration. Deploying infrastructure and workload separately lets you address distinct lifecycle and operational concerns.

- Consider your workflow as a mechanism to deploy to another region if there is a regional failure. Build the pipeline so that you can deploy a new cluster in a new region with parameter and input alterations.

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.

Consider the following points when planning for security.

- An AKS pod authenticates itself by using a *workload identity* stored in Microsoft Entra ID. Using a workload identity is preferable because it doesn't require a client secret.

- With managed identities, the executing process can quickly get Azure Resource Manager OAuth 2.0 tokens; there is no need for passwords or connection strings.

In AKS, you can assign identities to individual pods by using Microsoft Entra Workload ID.

- Each service in the microservice application should be assigned a unique workload identity to facilitate least-privileged RBAC assignments. You should only assign identities to services that require them.
- In cases where an application component requires Kubernetes API access, ensure that application pods are configured to use a service account with appropriately scoped API access. For more information on configuring and managing Kubernetes service account, see Managing Kubernetes Service Accounts.
- Not all Azure services support data plane authentication using Microsoft Entra ID. To store credentials or application secrets for those services, for third-party services, or for API keys, use Azure Key Vault. Azure Key Vault provides centralized management, access control, encryption at rest, and auditing of all keys and secrets.
- In AKS, you can mount one or more secrets from Key Vault as a volume. The pod can then read the Key Vault secrets just like a regular volume. For more information, see the secrets-store-csi-driver-provider-azure project on GitHub.

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 section in the Microsoft Azure Well-Architected Framework describes cost considerations. Use the Azure pricing calculator to estimate costs for your specific scenario.
- AKS has no costs associated with deployment, management, and operations of the Kubernetes cluster. You only pay for the VM instances, storage, and networking resources the cluster consumes. Cluster autoscaling can significantly reduce the cost of the cluster by removing empty or unused nodes.
- To estimate the cost of the required resources, see the Container Services calculator.
- Consider enabling AKS cost analysis for granular cluster infrastructure cost allocation by Kubernetes specific constructs.

Next steps

- Introduction to Azure Kubernetes Service
- What is Azure Virtual Networks?
- What is Azure Private Link?
- What is Azure Application Gateway?
- What is Azure Bastion?
- About Azure Key Vault
- Introduction to Azure Container Registry
- Welcome to Azure Cosmos DB
- Azure Monitor overview

Related resources

- Baseline architecture for an Azure Kubernetes Service (AKS) cluster
- Design, build, and operate microservices on Azure with Kubernetes
- Microservices architecture on AKS
- Monitor a microservices architecture in AKS
- Performance tuning scenario: Distributed business transactions
- Building a CI/CD pipeline for microservices on Kubernetes

End-to-end geospatial storage, analysis, and visualization

Article • 10/12/2022

Geospatial data comes in various forms and requires a wide range of capabilities to process, analyze and visualize the data. While Geographic Information System (GIS) is common, it also is largely not cloud-native. Most GIS run on the desktop, which limits their scale and performance. While there have been advances in moving the data to the backend, these systems remain IaaS bound which makes them difficult to scale.

This article will provide a high-level approach to use cloud-native capabilities along with some open-source software options and commercial options. Three personas will be considered. The personas are architects that are looking for a high-level flow without getting into the specifics of an implementation. The personas include the following:

- General geospatial architect. This architect is looking for a means to implement geospatial but may not have a background in GIS or Remote Sensing.
- OSS geospatial architect. This architect is dedicated to an open-source software (OSS) solution but takes advantage of the cloud for compute and storage.
- COTS geospatial architect. This architect is dedicated to COTS but also takes advantage of cloud compute and storage.

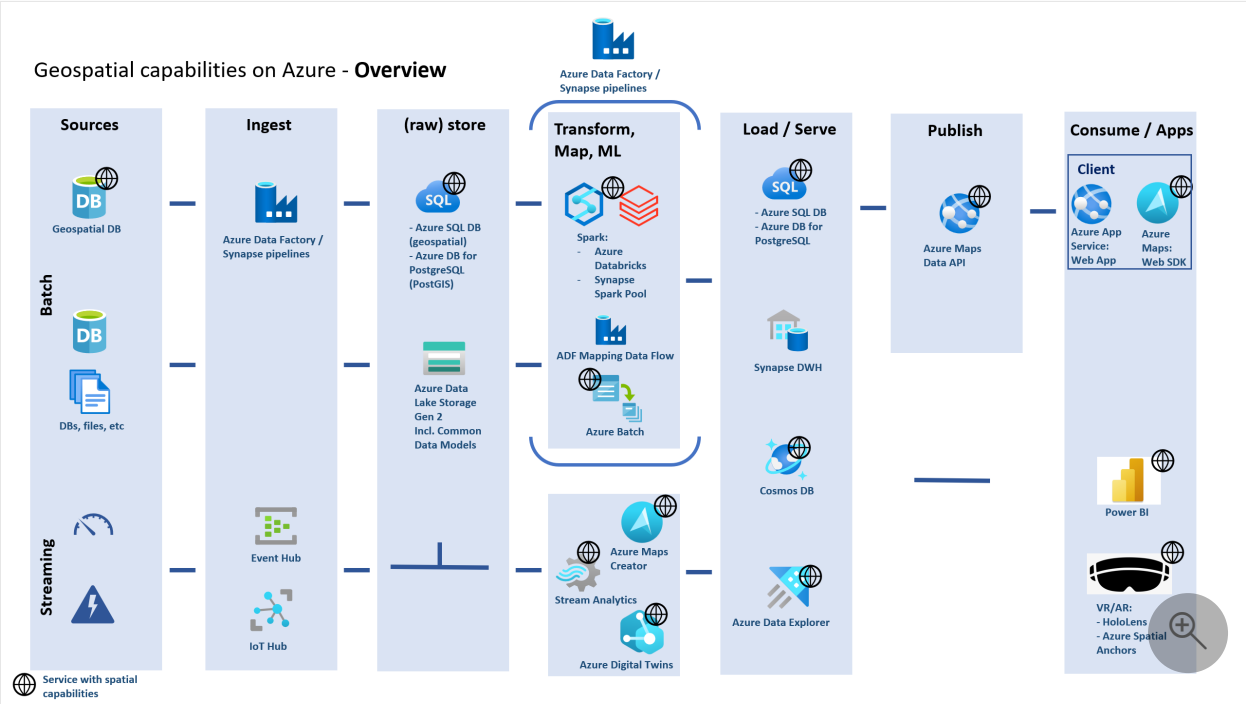
Potential use cases

The solutions provided in these architectures apply to many use cases:

- Processing, storing, and providing access to large amounts of raster data, such as layers or climate data.
- Combining entity location data from ERP systems with GIS reference data or include vector data, arrays, point clouds, etc.
- Storing Internet of Things (IoT) telemetry from moving devices, and analyze in real time or in batch
- Running analytical geospatial queries.
- Embedding curated and contextualized geospatial data in web apps.
- Processing data from drones, aerial photography, satellite imagery, LiDAR, gridded model results etc.

General geospatial architecture

Azure has many native geospatial capabilities. In this diagram and the ones that follow, you'll find high-level stages in which geospatial data undergoes. First, you have the data source, an ingestion step, a place where the data is stored, transformed, served, published, and finally consumed. Note the globe icon beside the services with native geospatial capabilities. Also, these diagrams aren't to be considered linear processes. One may start in the Transforms column, Publish and Consume, and then create some derived datasets, which requires going back to a previous column.



This architecture flow assumes that the data may be coming from databases, files or streaming sources and not stored in a native GIS format. Once the data is ingested with Azure Data Factory, or via Azure IoT, Event Hubs and Stream Analytics, it could then be stored permanently in warm storage with Azure SQL, Azure SQL Managed Instance, Azure Database for PostgreSQL or Azure Data Lake Storage. From there, the data can be transformed and processed in batch with Azure Batch or Synapse Spark Pool, of which both can be automated through the usage of an Azure Data Factory or Synapse pipeline. For real-time data, it can be further transformed or processed with Stream Analytics, Azure Maps or brought into context with Azure Digital Twins. Once the data is transformed, it can then once again be served for additional uses in Azure SQL DB or Azure Database for PostgreSQL, Synapse SQL Pool (for abstracted non-geospatial data), Azure Cosmos DB, or Azure Data Explorer. Once ready, the data can be queried directly through the data base API, but frequently a publish layer is used. The Azure Maps Data API would suffice for small datasets, otherwise a non-native service can be introduced based on OSS or COTS, for accessing the data through web services or desktop applications. Finally, the Azure Maps Web SDK hosted in Azure App Service would allow for geovisualization. Another option is to use Azure Maps in Power BI. Lastly, HoloLens

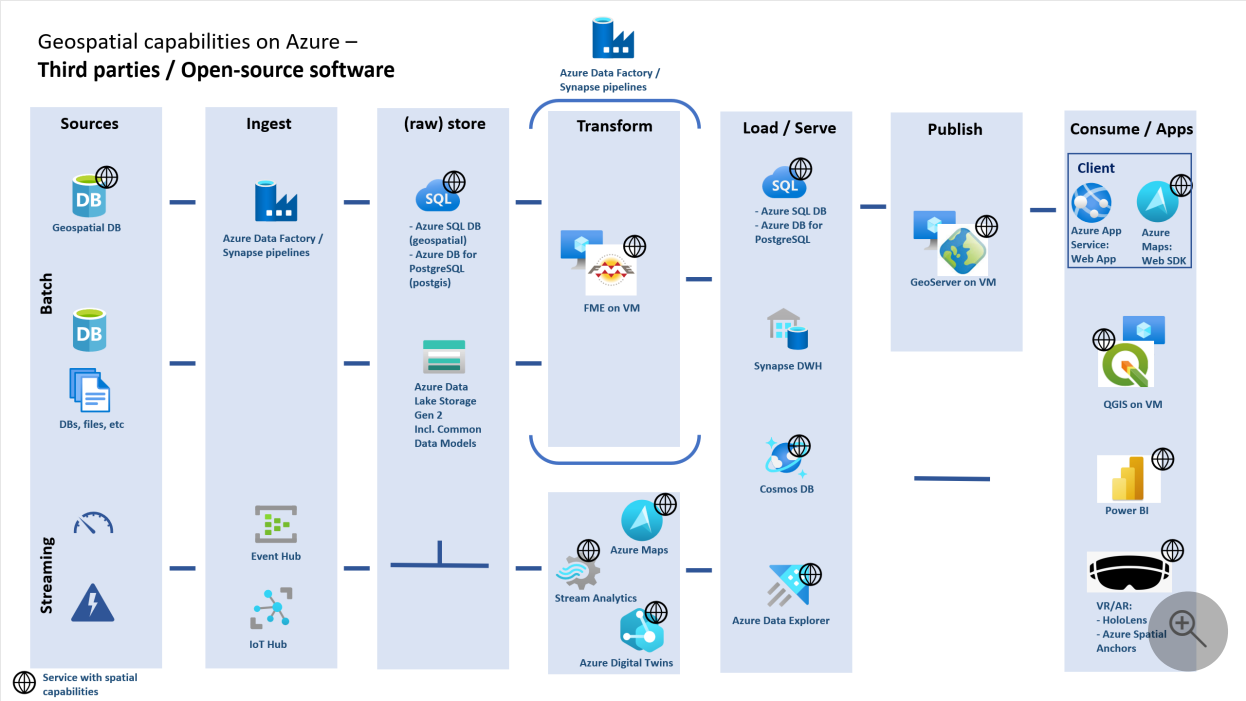
and Azure Spatial Anchors can be used to view the data and place it in the real-world for virtual reality (VR) and augmented reality (AR) experiences.

It should be noted as well that many of these options are optional and could be supplemented with OSS to reduce cost while also maintaining scalability, or 3rd party tools to utilize their specific capabilities. The next session addresses this need.

3rd Party and Open-source software geospatial architecture

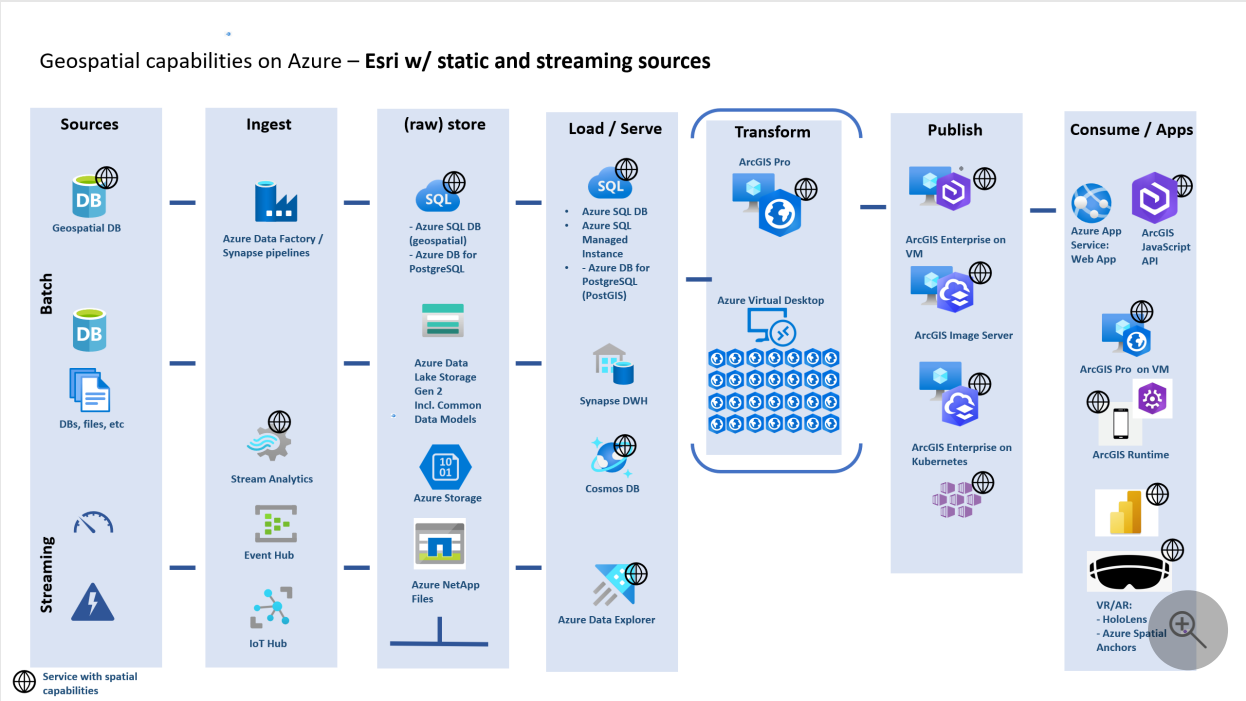
This pattern takes the approach of using Azure native geospatial capabilities while at the same time taking advantage of some 3rd party tools and open-source software tools.

The most significant difference between this approach and the previous flow diagram is the use of FME from Safe Software Inc., which can be acquired from the Azure Marketplace. FME allows geospatial architects to integrate various type of geospatial data which includes CAD (for Azure Maps Creator), GIS, BIM, 3D, point clouds, LIDAR, etc. There are 450+ integration options, and can speed up the creation of many data transformations through its functionality. Implementation, however, is based on the usage of a virtual machine, and has therefore limits in its scaling capabilities. The automation of FME transformations might be reached using FME API calls with the use of Azure Data Factory and/or with Azure Functions. Once the data is loaded in Azure SQL, for example, it can then be served in GeoServer and published as a Web Feature Service (vector) or Web Mapping Tile Service (raster) and visualized in Azure Maps web SDK or analyzed with QGIS for the desktop along with the other Azure Maps base maps.



COTS geospatial architecture: Esri with static and streaming sources

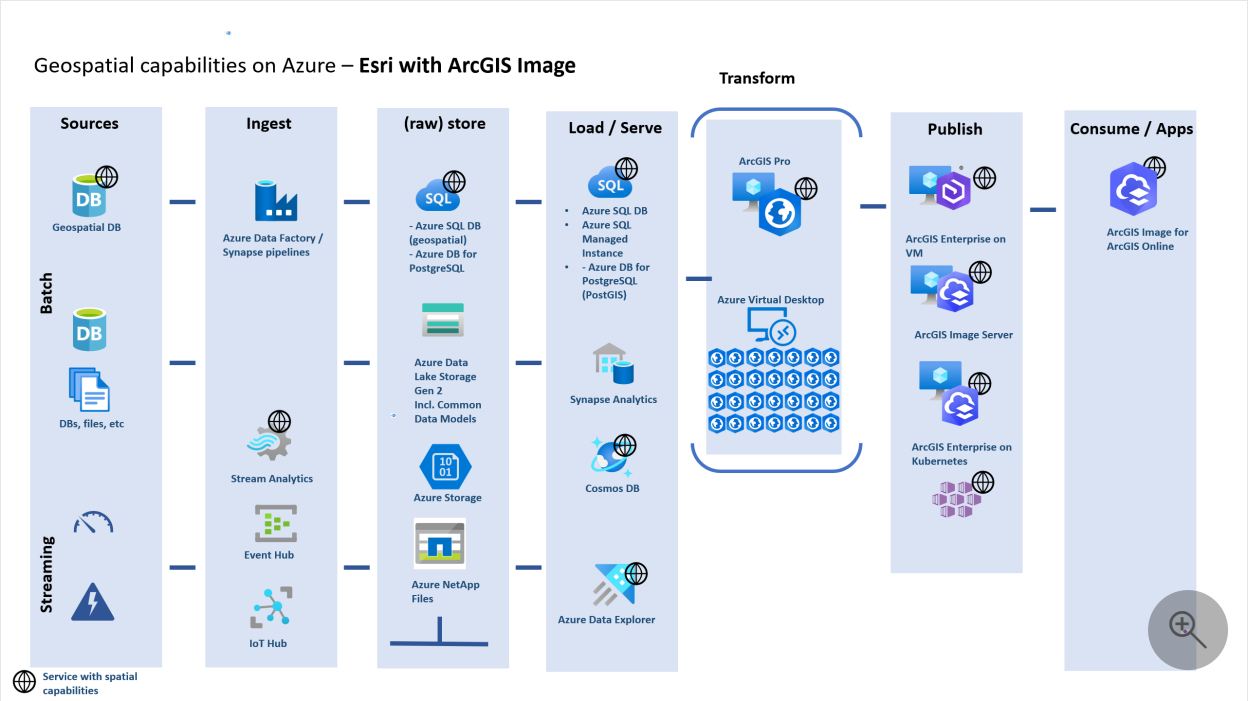
The next approach we'll look at uses commercial GIS as the basis for the solution. Esri's technology, available from the Azure Marketplace, will be the foundation for this architecture, although other commercial software can fit the same patterns. As before, the sources, ingestion, (raw) store, Load/Serve largely remain the same. The data can also be transformed with ArcGIS Pro on a standalone computer (VM) or as part of a larger solution with Azure Virtual Desktop. The data can be published via ArcGIS Enterprise or with the ArcGIS Enterprise on Kubernetes (Azure Kubernetes Service). Imagery can be processed on IaaS with ArcGIS Image as part of the ArcGIS Enterprise deployment. The data can be consumed in web apps hosted in Azure App Service with the ArcGIS JavaScript SDK, an ArcGIS Pro end user, ArcGIS Runtime mobile SDK, or with ArcGIS for Power BI. Likewise, users can consume the data with ArcGIS Online.



COTS geospatial imagery architecture: Esri's ArcGIS Image and Azure Orbital

The next architecture involves Azure Orbital and Esri's ArcGIS Image. With this end-to-end flow, Azure Orbital allows you to schedule contacts with satellites and downlink the data into a VM or stream to Azure Event Hubs. Besides direct streamed satellite data, drone or other imagery data can be brought on the platform, and processed. The raw data can be stored in Azure NetApp Files, an Azure Storage account (blob), or in a database such as Azure Database for PostgreSQL. Depending on the satellite and sensor

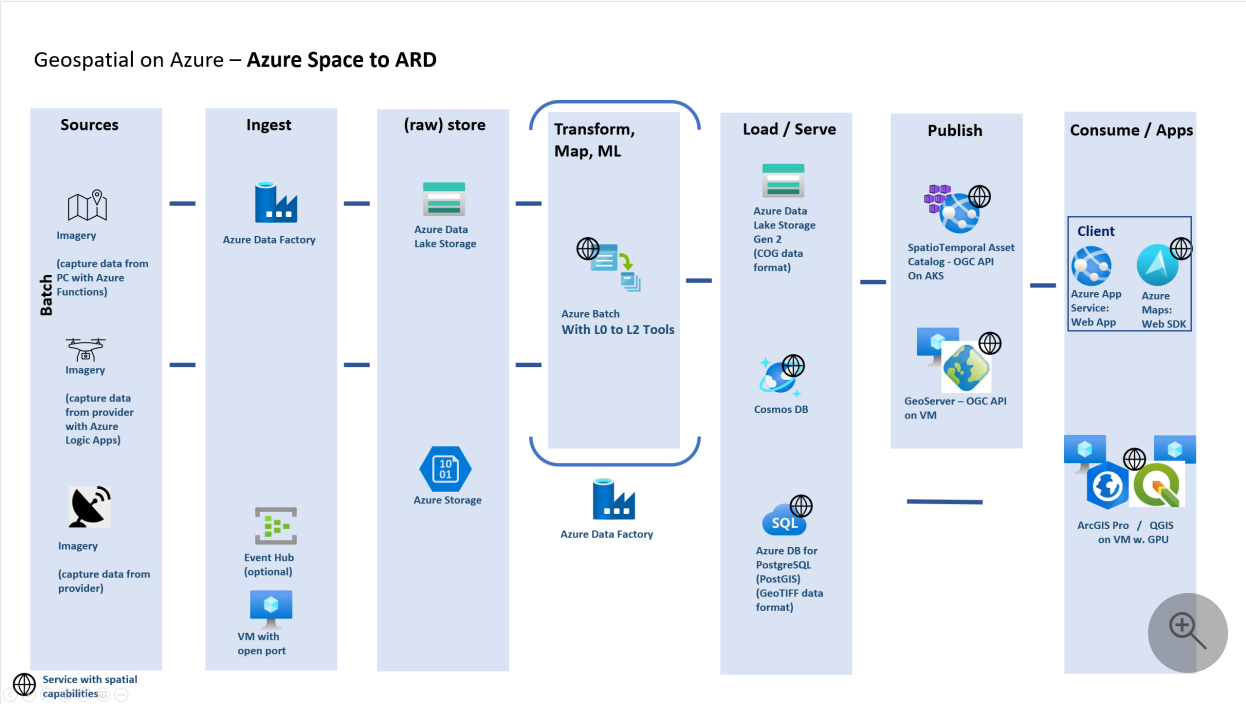
platform, the data is transformed from Level 0 to Level 2 dataset. See NASA Data Processing Levels. To which level is required, is dependent on the satellite and sensor. Next, ArcGIS Pro can transform the data into a Mosaic Dataset. The Mosaic Dataset is then turned into an image service with ArcGIS Enterprise (on VMs or Kubernetes). ArcGIS Image Server can serve the data directly as an image service or a user can consume the image service via ArcGIS Image for ArcGIS Online.



COTS/Open-source software geospatial imagery architecture: Azure Space to Analysis Ready Dataset

When Analysis Ready Datasets are made available through APIs that enable search and query capabilities, like with Microsoft's Planetary Computer, there's no need to first download the data from a satellite. However, if low lead times are required for imagery, acquiring the data directly from Azure Space is ideal because a satellite operator or mission driven organization can schedule a contact with a satellite via Azure Orbital. The process for going from Level 0 to Level 2 Analysis Ready Dataset varies by the satellite and the imagery products. Multiple tools and intermediate steps are often required. Azure Batch or another compute resource can process the data in a cluster and store the resulting data. The data may go through multiple steps before it's ready for being used in ArcGIS or QGIS or some other geovisualization tool. For example, once the data is in a Cloud Optimized GeoTIFF (COG) format, it's served up via a Storage Account or Azure Data Lake and made accessible and queryable via the STAC API, which can be deployed on Azure as a service, with AKS among others. Alternatively, the data is published as a Web Mapping Tile Service with GeoServer. Consumers can then access

the data in ArcGIS Pro or QGIS or via a web app with Azure Maps or Esri's mobile and web SDKs.



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 Orbital is a fully managed, cloud-based ground station as a service that allows you to streamline your operations by ingesting space data directly into Azure.
- 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 Cosmos DB is a fully managed NoSQL database service for modern app development.
- Azure Synapse Analytics is an enterprise analytics service that accelerates time to insight across data warehouses and big data systems.
- Azure Digital Twins is a platform as a service offering that enabled the creation of twin graphs based on digital models of entire environments, which could be buildings, factories, farms, energy networks, railways, stadiums, or entire cities.
- Azure Virtual Desktop is a desktop and app virtualization service that runs on the cloud.
- 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.

- Azure Batch allows you to run large-scale parallel and high-performance computing jobs.
- Azure 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 SQL Database is a PaaS version of SQL Server and is an intelligent, scalable, relational database service.
- 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.
- 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.
- 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 NetApp Files is an enterprise-class, high-performance, metered file Network Attached Storage (NAS) service.
- Quantum GIS is a free and open-source desktop GIS that supports editing, analysis, geovisualization of geospatial data.
- ArcGIS Enterprise is a platform for mapping and geovisualization, analytics and data management, which hosts data, applications, and custom low-code/no-code applications. It works along with the desktop GIS called ArcGIS Pro or ArcGIS Desktop (not included here because it has been supplanted by ArcGIS Pro).
- ArcGIS Pro is Esri's professional desktop GIS application. It allows power users to explore, geovisualize, and analyze data. It includes 2D and 3D capabilities and runs best on Azure High Performance Compute VMs such as the NV series. The use of ArcGIS can be scaled using Azure Virtual Desktop.

- ArcGIS Image for ArcGIS Online is an extension to ArcGIS Online (SaaS) which allows for geovisualization, hosting, publishing, and analysis.
- STAC API specification allows you to query and retrieve raster data via a catalog.

Although not shown in the diagrams above, Azure Monitor, Log Analytics and Key Vault would also be part of a broader solution.

- 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.
- Azure 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

Various Spark libraries are available for working with geospatial data on Azure Databricks and Synapse Spark Pools. See these libraries:

- Apache Sedona (GeoSpark)
- GeoPandas
- GeoTrellis

But other solutions also exist for processing and scaling geospatial workloads with Azure Databricks.

- Other Python libraries to consider include PySAL, Rasterio, WhiteboxTools, Turf.js, Pointpats, Raster Vision, EarthPy, Planetary Computer, PDAL, etc.

- Vector tiles provide an efficient way to display GIS data on maps. A solution could use 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.

Related resources

Related architectures

- Big data analytics with Azure Data Explorer
- Health data consortium on Azure
- DataOps for the modern data warehouse
- Azure Data Explorer interactive analytics

Related guides

- Geospatial data processing and analytics
- 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
- Monitor Azure Databricks

Next steps

- Connect a WFS to Azure Maps
- Geospatial clustering
- Explore ways to display data with Azure Maps.

Ingest FAA SWIM content to analyze flight data

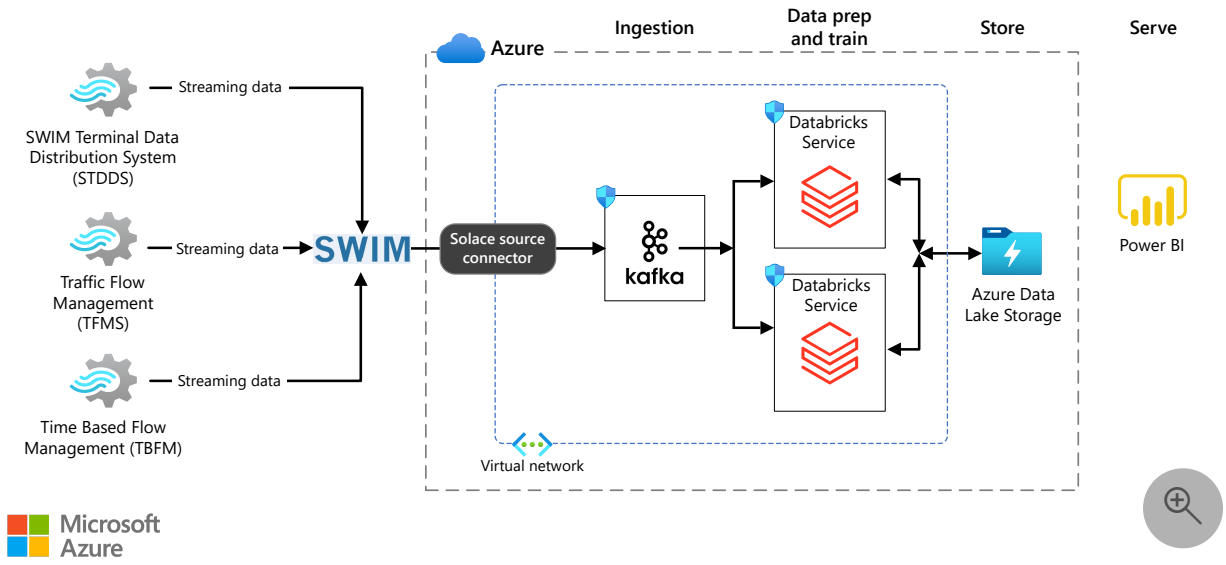
Azure Databricks | Azure Data Lake Storage | Power BI

This solution shows how to integrate Chef Infra, Chef InSpec, Test Kitchen, Terraform, Terraform Cloud, and GitHub Actions to automate and to create data analytics environments. It uses an Azure Databricks cluster to analyze the data.

Architecture

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.

Progress Chef and HashiCorp Terraform are trademarks of their respective companies. No endorsement is implied by the use of these marks.



Download a Visio file of this architecture.

The left side of the diagram shows the Federal Aviation Administration (FAA) System Wide Information Management (SWIM) system and three of its data producers. The right side shows the Azure architecture that connects, ingests, and analyzes data coming from these sources.

This solution uses Apache Kafka because of its event-driven architecture. Apache Kafka is an open-source platform that lets you collect, process, store, and integrate data from

various sources in real-time. It's based on the concept of a commit log, which is a record of events that happen in a system. You can use Kafka to publish or subscribe to streams of events, and also to transform and analyze them by using the Streams API or other external tools like Databricks.

Another important aspect of this architecture is that SWIM uses Solace, so Kafka uses a Solace source connector to connect and ingest the data. Solace provides support for open source and sink connectors that you can build and deploy in your Kafka cluster.

Workflow

1. SWIM provides data. The architecture diagram shows three of the most common data sources (TFMS, TBFM, and STDDS). The type of information that you want to analyze dictates the data source from SWIM that you need to subscribe to.
2. A Solace source connector connects and ingests the data into Kafka.
3. Messages from Kafka are cleaned, prepped, and parsed in a workspace in Azure Databricks. This Azure Databricks workspace is where data scientists do their work. They use notebooks written in Python, Scala, or R that contain the logic they need to parse the data or even train models based on it.
4. Azure Data Lake provides storage.
5. Power BI or Tableau displays the final data.

Components

- Azure Databricks. A data analytics platform that's optimized for the Azure cloud.
- Azure Data Lake Storage. A massively scalable and highly secure data lake for high-performance analytics workloads.
- Power BI. An analytics and BI platform that can help you discover insights in your data.
- SWIM. A National Airspace System (NAS) information system that provides publicly available FAA data to FAA-approved consumers via Solace Java Message Service (JMS) messaging.
- Apache Kafka. An Apache event streaming platform.

Alternatives

For ingestion, the solution uses Apache Kafka in a single VM. This design configuration creates a single point of failure, but for a more robust solution, you can use one of these options:

- Apache Kafka in Azure HDInsight

- Apache Kafka for Confluent Cloud

Both are managed services and offer multiple benefits, like SLAs, simplified configuration, and scalability. They're also more expensive.

As an alternative to Power BI, you can use Tableau or another visualization option.

Scenario details

This solution relies on the Federal Aviation Administration (FAA) System Wide Information Management (SWIM) system. SWIM provides a single point of access for near real-time, relevant, and reliable aeronautical, flight, weather, and surveillance information. It delivers the infrastructure, standards, and services needed to optimize the secure exchange of relevant data across the National Airspace System (NAS) and the aviation community. As the digital data-sharing backbone of NextGen, SWIM enables both operational excellence and innovation.

This solution connects to the following FAA SWIM data sources via an Apache Kafka server:

- Traffic Flow Management Service (TFMS) distributes Traffic Flow Management (TFM) data to users via SWIM's NAS Enterprise Messaging Service (NEMS).
- Time-Based Flow Management (TBFM) enhances NAS efficiency by using the capabilities of the TBFM decision-support tool, a system already deployed at all high-altitude air traffic control centers in the contiguous United States.
- SWIM Terminal Data Distribution System (STDDS) converts legacy terminal data collected from airport towers and Terminal Radar Approach Control (TRACON) facilities into easily accessible information, which is published via NEMS.

For information about SWIM, see the FAA SWIM page.

Potential use cases

This solution consumes multiple data sources for flight data patterns. It's ideal for the aerospace, aircraft, and aviation industries. It can be used to analyze flight data patterns and to predict future flight patterns. For example, it can also be used to analyze flight data to improve flight safety.

The solution environment is flexible, so it can be extended to analyze other SWIM data sources or similar streamed data sources.

This solution also shows how to automate and create data analytics environments by using Chef Infra, Chef InSpec, Test Kitchen, Terraform, Terraform Cloud, and GitHub

Actions. This automation can be extended to other data analytics environments. For example, you can use it to automate the deployment of an Azure Databricks cluster that analyzes data from a different source.

SWIM architecture

SWIM is a NAS information system. It's an FAA cloud-based service that provides publicly available FAA SWIM content to FAA-approved consumers via Solace JMS messaging.

SWIM provides a single point of access for aviation data. Data producers publish data once, and users access the information they need through a single connection. SWIM provides multiple data producers. Depending on the type of data you need, you can subscribe to one or more of them. SWIM has a typical Publisher-Subscriber architecture.

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.

A key requirement for this architecture is that all traffic must be internal and highly secure. To meet this requirement:

- Virtual network injection is used to deploy Azure Databricks. This deployment method keeps communication between the cluster and Kafka internal.
- The Azure Databricks workspace uses your Azure identity for authentication.
- Network security groups filter network traffic to and from Azure Databricks and Kafka VMs.

For more information about improving the security of your solution, see Overview of the 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.

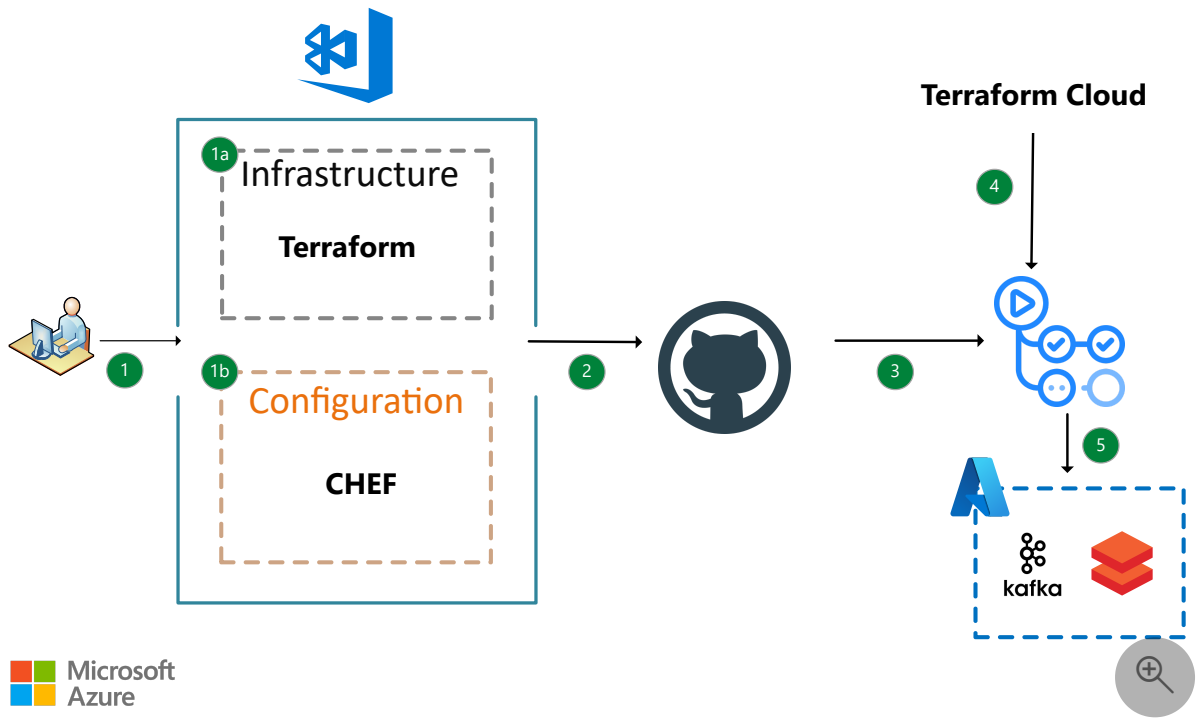
If you run this project, your account is billed. For information, see Cost optimization documentation.

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.

CI/CD pipeline architecture

This architecture uses GitHub Actions to orchestrate the CI/CD pipeline:



Download a Visio file of this architecture.

Dataflow

1. Developers work on either the Infrastructure or the Configuration code.

    a. Infrastructure code uses Terraform.

b. Configuration code uses Chef.

2. Developers push their code to the GitHub repository using a GitHub pull request.

3. The pull request triggers a GitHub Actions workflow. The workflow runs the code through a series of tests. If the changes are configuration-related, the workflow runs Chef InSpec and Test Kitchen to test the code. Once the tests are completed, it reports the results back to GitHub. If the changes are infrastructure-related, the workflow also creates a Terraform plan and posts it as a comment in the pull request. The plan shows the changes that are applied to the infrastructure if the pull request is approved.

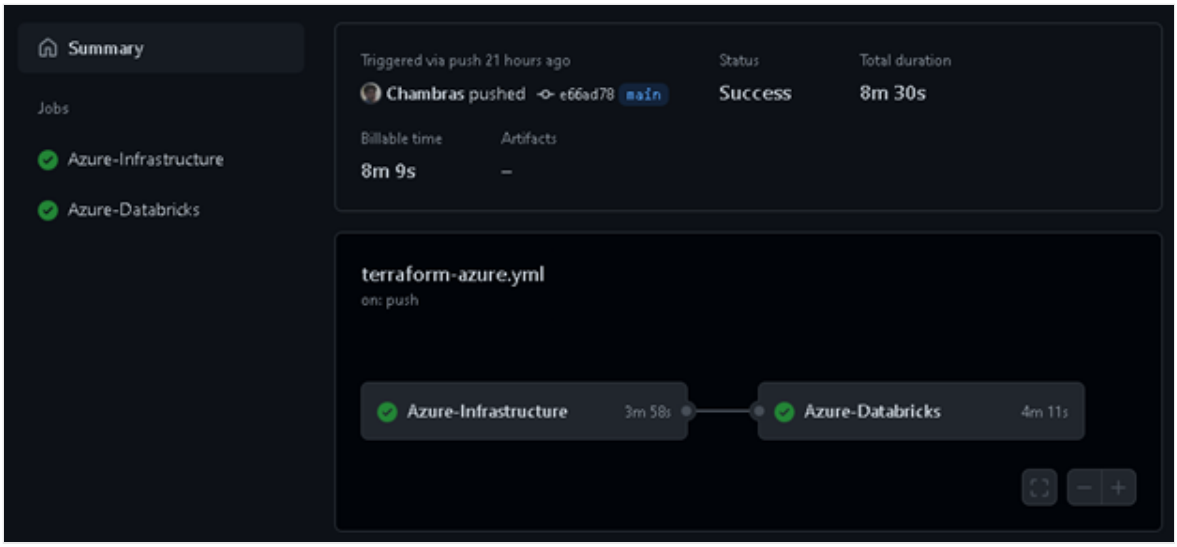
4. If the changes are infrastructure-related, the workflow runs Terraform Cloud's remote state. Terraform Cloud is a hosted service that provides remote state management, locking, and other features.

5. If the tests pass, and the pull request is approved, the workflow merges the code into the `main` branch. After it's merged, the Actions workflow pushes changes in either the infrastructure or in the configurations for Kafka, Azure Databricks, and so on.

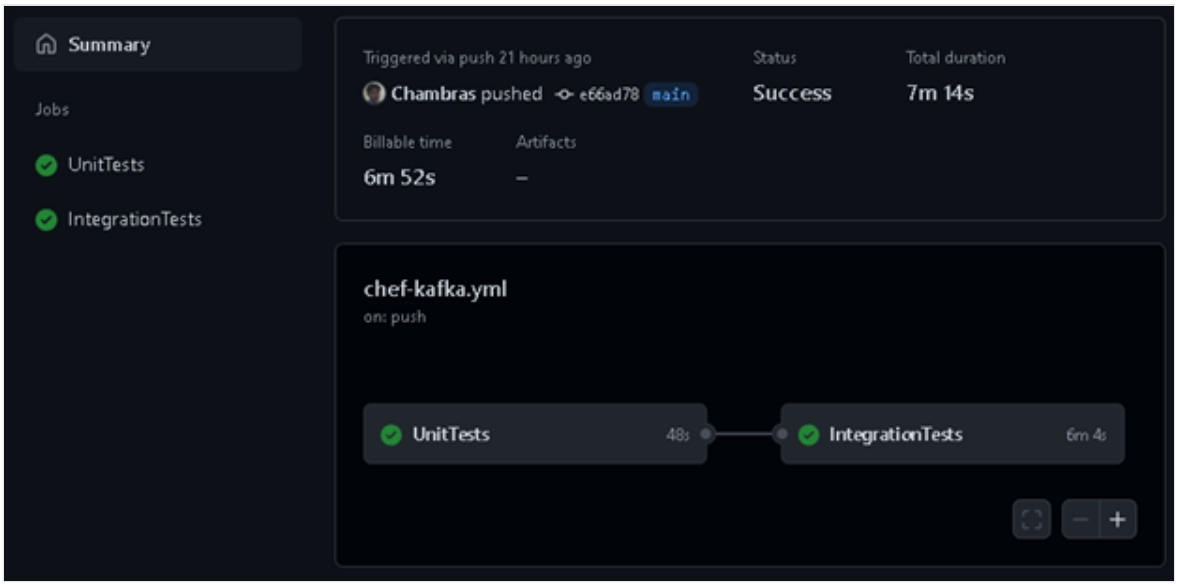
GitHub workflows

In this solution, two GitHub Actions workflows automate the infrastructure that hosts the data analytics environment. Terraform deploys the infrastructure. After provisioning is complete, Chef configures the resources that are required to connect to TFMS, TBFM, STDDS, and others if needed.

- **terraform-azure.yml** performs Terraform deployment. It uses Terraform Cloud's remote state. It also creates an Azure Databricks cluster, deploys some starter Python notebooks to test connectivity to the Kafka server, and retrieves messages. It creates all infrastructure with proper naming conventions and tagging.



- **chef-kafka.yml** performs static code analysis by using ***Cookstyle***, performs unit tests by using ***Chef InSpec***, and performs integration tests by using ***Test Kitchen***. These tests ensure the cookbook is properly tested before it's uploaded to Chef Server.



Deploy this scenario

For information about deploying this solution, workflows, cookbooks, sample notebooks, Terraform files, and more, see the Azure/SWIMDataIngestion GitHub repository. The repository contains step-by-step instructions to deploy the solution. The deployment process is fully automated using GitHub Actions, but for it to work, it requires that some secrets are set up in GitHub.

The deployment process is divided into three parts: infrastructure, configuration, and visualization.

- The infrastructure part deploys the following Azure resources:

    - A virtual network
    - Subnets
    - A resource group
    - Kafka server
    - An Azure storage account
    - Azure Data Lake Storage on top of the storage account
    - Network security groups
    - An Azure Databricks workspace created with virtual network injection, which keeps the traffic internal

- The configuration part configures the resources that are required to connect to TFMS, TBFM, STDDS, and others if needed. To connect Kafka to SWIM, request access to SWIM and specify the data source to connect to. After it's approved, FAA sends a link to the data source endpoint, user name, password, and port for connection. Here are three of the most common data sources:
    - STDDS. SWIM Terminal Data Distribution System.
    - TFMS. Traffic Flow Management Service.
    - TBFM. Time-Based Flow Management.

- Lastly, connect a visualization dashboard like Power BI or Tableau to Azure Databricks.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- Marcelo Zambrana | Senior Software Engineer

Other contributor:

- Mick Alberts | Technical Writer

Next steps

- What is Azure Databricks?
- Introduction to Azure Data Lake Storage Gen2
- Introduction to Power BI

Related resources

- All aerospace architectures
- Publisher-Subscriber pattern
- Advanced analytics architecture

Organize spaceborne geospatial data with SpatioTemporal Asset Catalog (STAC)

Article • 04/17/2023

This reference architecture shows an end-to-end implementation of SpatioTemporal Asset Catalog (STAC) creation to structure geospatial data. In this document, we'll use the publicly available National Agriculture Imagery Program (NAIP) data set using geospatial libraries on Azure. The architecture can be adapted to take data sets from other sources such as Satellite imagery providers, Azure Orbital Ground Station (AOGS), or Bring Your Own Data (BYOD).

The implementation consists of four stages: Data acquisition, Metadata generation, Cataloging, and Data discovery via STAC FastAPI. This article also shows how to build STAC based on a new data source or on bring-your-own-data.

An implementation of this architecture is available on GitHub.

This article is intended for users with intermediate levels of skill in working with spaceborne geospatial data. Refer to the table in the glossary for the definition of commonly used STAC terms. For more details visit the official stacspec page.

Scenario details

Spaceborne data collection is becoming increasingly common. There are various data providers of spatiotemporal assets such as Imagery, Synthetic Aperture Radar (SAR), Point Clouds, and so forth. Data providers don't have a standard way of providing users access to their spatiotemporal data. Users of spatiotemporal data are often burdened with building unique workflows for each different collection of data they want to consume. Developers are required to develop new tools and libraries to interact with the spatiotemporal data.

The STAC community has defined a specification to remove these complexities and spur common tooling. The STAC specification is a common language to describe geospatial information, so it can more easily be worked with, indexed, and discovered. There are many deployed products that are built on top of STAC and one such is Microsoft Planetary Computer providing a multi-petabyte STAC catalog of global environmental data for sustainability research.

Our sample solution uses open source tools such as STAC FastAPI, pystac, Microsoft Planetary Computer APIs and open standard geospatial libraries (listed in the Components section) to run the solution on Azure.

Potential use cases

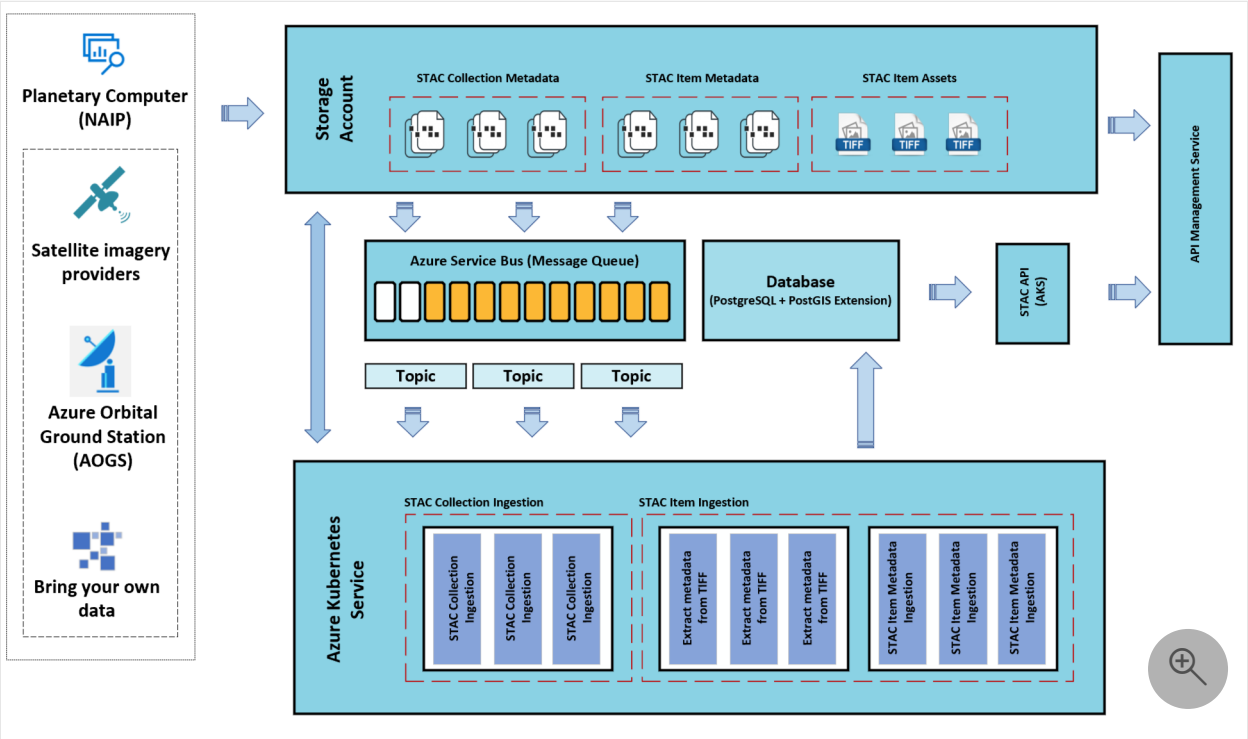
STAC has become an industry standard to support how geospatial data should be structured and queried. It has been used in many production deployments for various use cases.

Here are a couple of examples:

- A satellite data provider company needs to make their data easy to discover and access. The provider builds STAC Catalogs to index all of its historic archive data sets and also the incoming refresh data on a daily basis. A web client UI is built on top of STAC APIs that allows users to browse the catalogs and search for their desired images based on the area of interest (AOI), date/time range, and other parameters.

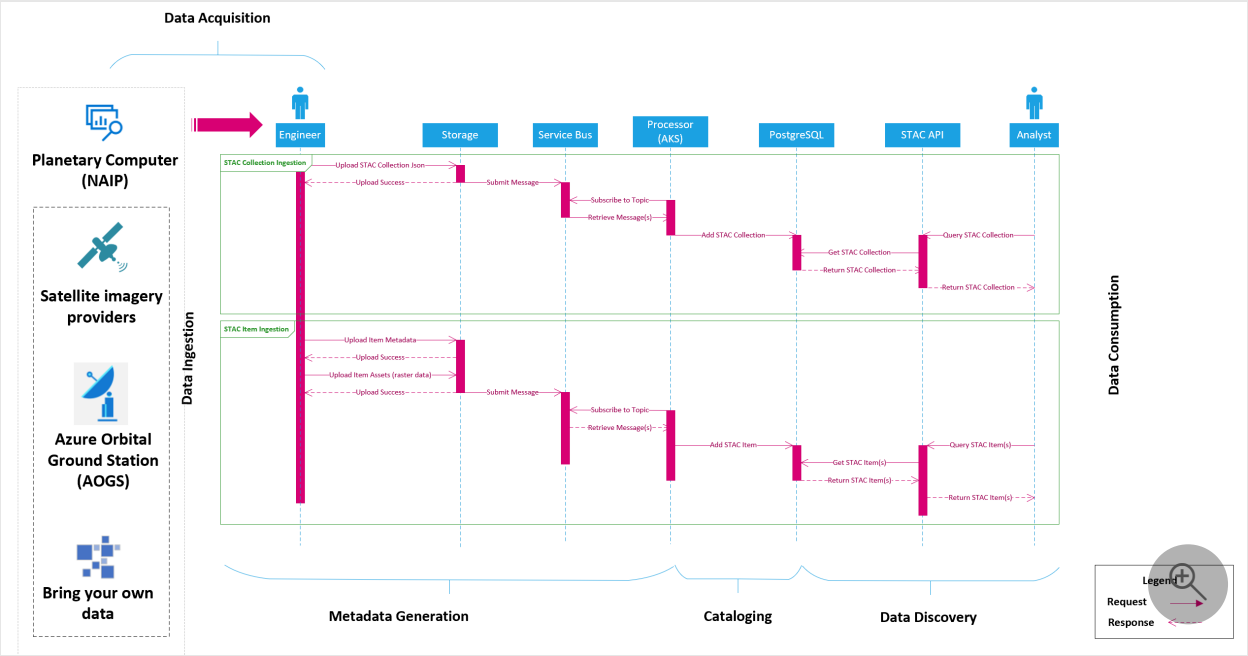
- A geospatial data analysis company needs to build a database of spaceborne data including imagery, Digital Elevation Model (DEM), and 3D types that it has acquired from various data sources. The database will serve its geographic information system (GIS) analysis solution to aggregate different data sets for machine learning model-based object detection analysis. To support a standard data access layer, the company decides to implement an open source compatible STAC API interface for the GIS analysis solution to interact with the database in a scalable and performing way.

Architecture



Download a Visio file for this architecture.

Dataflow



Download a Visio file for this dataflow.

The following sections describe the four stages in the architecture.

Data acquisition

- Spaceborne data is provided by various data providers including Airbus, NAIP/USDA (via the Planetary Computer API), and Maxar.

- In the sample solution we use the NAIP dataset provided by Microsoft Planetary Computer.

Metadata generation

- Data providers define the metadata describing provider, license terms, keywords, etc. This metadata forms the STAC Collection.
- Data providers may provide metadata describing the geospatial assets. In our sample, we use metadata provided by NAIP & FGDC. More metadata is extracted from the assets using standard geospatial libraries. This metadata forms the STAC Items.
- This STAC Collection and Items are used to build the STAC Catalog that helps users discover the spatiotemporal assets using STAC APIs.

Cataloging

- STAC Catalog
  - STAC Catalog is a top-level object that logically groups other Catalog, Collection, and Item Objects. As part of the deployment of this solution, we create a STAC Catalog under which all the collections and items are organized.

- STAC Collection
  - It's a related group of STAC Items that is made available by a data provider.
  - Search queries for discovering the assets are scoped at the STAC Collection level.
  - It's generated for a data provider, NAIP in this case and this JSON metadata is uploaded to an Azure Storage container.
  - The upload of a STAC Collection metadata file triggers a message to Azure Service Bus.
  - The processor processes this metadata on Azure Kubernetes Cluster and ingests to the STAC Catalog database (PostgreSQL database). There are different processors for different data providers and each processor subscribes to the respective Service Bus Topic.

- STAC Item and asset
  - An asset that is to be cataloged (raster data in the form of GeoTiff, Cloud Optimized GeoTiff and so forth). Metadata describing the asset and the metadata extracted from the asset is uploaded to the Storage Account under the appropriate storage container.
  - The asset(s) (GeoTiff) are then uploaded to the Storage Account under the appropriate storage container following the successful upload of their corresponding metadata.

    - Each asset and its associated metadata uploaded to the Storage Account triggers a message to the Service Bus. This metadata forms the STAC Item in the catalog database.
    - The processor processes this metadata on Azure Kubernetes Cluster and ingests to the STAC Catalog database (PostgreSQL database).

Data discovery

- STAC API is based on open source STAC FastAPI.
- STAC API layer is implemented on Azure Kubernetes Service and the APIs are exposed using API Management Service.
- STAC APIs are used to discover the geospatial data in your Catalog. These APIs are based on STAC specifications and understand the STAC metadata defined and indexed in the STAC Catalog database (PostgreSQL server).
- Based on the search criteria, you can quickly locate your data from a large dataset.
    - Querying the STAC Collection, Items & Assets:
        - A query is submitted by a user to look up one or more STAC Collection, Items & Assets through the STAC FastAPI.
        - STAC FastAPI queries the data in the PostgreSQL database to retrieve the STAC Collection, Items & references to Assets.
        - The result is served back to the user by the STAC FastAPI.

Components

The following Azure services are used in this architecture.

- 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.
- Service Bus is part of a broader Azure messaging infrastructure that supports queueing, publish/subscribe, and more advanced integration patterns.
- Azure Data Lake Storage is dedicated to big data analytics, and is built on Azure Blob Storage.
- Azure Virtual Network enables Azure resources to securely communicate with each other, the internet, and on-premises networks.
- Azure Database for PostgreSQL - Flexible Server is a fully managed database service designed to provide more granular control and flexibility over database management functions and configuration settings. It has richer capabilities such as zone resilient high availability (HA), predictable performance, maximum control, custom maintenance window, cost optimization controls, and simplified developer experience suitable for your enterprise workloads.

- API Management Services offers a scalable, multicloud API management platform for securing, publishing and analyzing APIs.
- Azure Kubernetes Services offers the quickest way to start developing and deploying cloud-native apps, with built-in code-to-cloud pipelines and guardrails.
- Container Registry to store and manage your container images and related artifacts.
- Virtual Machine (VM) gives you the flexibility of virtualization for a wide range of computing solutions. In a fully secured deployment, a user connects to a VM via Azure Bastion (described in the next item below) to perform a range of operations like copying files to storage accounts, running Azure CLI commands, and interacting with other services.
- Azure Bastion enables you to securely and seamlessly RDP & SSH to your VMs in Azure virtual network, without the need of public IP on the VM, directly from the Azure portal, and without the need of any other client/agent or any piece of software.
- Application Insights provides extensible application performance management and monitoring for live web apps.
- Log Analytics is a tool to edit and run log queries from data collected by Azure Monitor logs and interactively analyze the results.

The following Geospatial libraries are also used:

- GDAL is a library of tools for manipulating spaceborne data. GDAL works on raster and vector data types. It's a good tool to know if you're working with spaceborne data.
- Rasterio is a module for raster processing. You can use it to read and write several different raster formats in Python. Rasterio is based on GDAL. When the module is imported, Python automatically registers all known GDAL drivers for reading supported formats.
- Shapely is a Python package for set-theoretic analysis and manipulation of planar features. It uses (via Python's ctypes module) functions from the widely deployed GEOS library.
- pyproj performs cartographic transformations. It converts from longitude and latitude to native map projection x, y coordinates, and vice versa, by using PROJ.

Considerations

- The sample solution demonstrates STAC's core JSON support that is needed to interact with any geospatial data collection. While STAC standardizes metadata fields, naming conventions, query language, and catalog structure, users should

additionally consider STAC Extensions to support metadata fields specific to their Assets.

- In the sample implementation, components that process the asset to extract metadata have a set number of replicas. Scaling this component allows you to process your assets faster. However, scaling isn't dynamic. If large number of assets to be cataloged, consider scaling these replicas.

Adding a new data source

To catalog more data sources or to catalog your own data source, consider the following options.

- Define the STAC Collection for your data source. Search queries are scoped at the STAC Collection level. Consider how the user will search STAC Items and Assets in your collection.
- Generate the STAC Item metadata. More metadata may be derived from geospatial assets using standard tools and libraries. Define and implement the process to capture supplemental metadata for the assets that will be useful in making STAC items rich and in turn make discovery of data using APIs easier.
- Once this metadata (in the form STAC Collection and STAC Items) is available for a data source, this sample solution can be used to build your STAC Catalog using the same flow. The data once cataloged will be queryable using standard STAC APIs.
- The processor component of this architecture is extensible to include custom code that can be developed and run as containers in Azure Kubernetes Cluster. It's intended to provide a way for different representation of geospatial data to be cataloged as assets.

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.

- Azure Kubernetes Service Container Security implementation ensures the processors are built and run as containers are secure.
- API Management Service Security baseline provides recommendations on how to secure your cloud solutions on Azure.
- Azure Database for PostgreSQL Security covers in-depth the security at multiple layers when data is stored in PostgreSQL Flexible Server including data at rest and data in transit scenarios.

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.

As this solution is intended for learning and development, we have used minimal configuration for the Azure resources. This minimal configuration runs a sample solution on a sample dataset.

Users can also adjust the configurations to meet their workload and scaling needs to be performant. For instance, you can swap Standard HDDs for Premium SSDs in your AKS cluster or scale API Management Services to premium SKUs.

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. Additionally, the following guidance can be useful in maximizing performance efficiency:

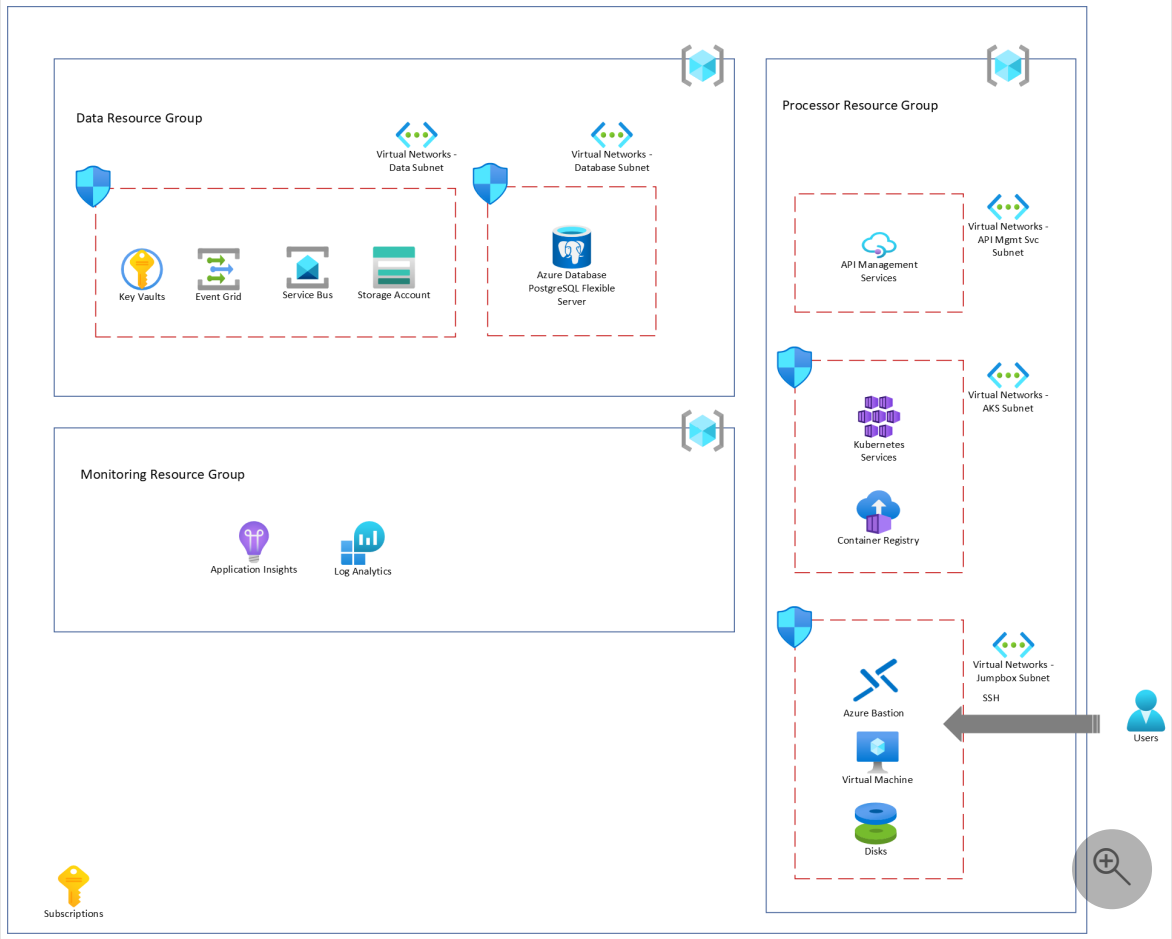
- Monitor and tune provides a way to monitor your data and tune your database to improvement performance.
- Performance tuning a distributing application walks through a few different scenarios, how to identify key metrics and improve performance.
- Baseline architecture for an Azure Kubernetes Service (AKS) cluster recommends baseline infrastructure architecture to deploy an Azure Kubernetes Service (AKS) cluster on Azure.
- Improve the performance of an API by adding a caching policy in Azure API Management is a training module on improving performance through Caching Policy.

Deploy this scenario

We built a sample solution that can be deployed into your subscription. This solution enables users to validate the overall data flow from STAC metadata, ingestion to discovering the assets using standard STAC APIs. The deployment instructions and the validation steps are documented in the README file.

At a high level, this deployment does the following:

- Deploys various infrastructure components such as Azure Kubernetes Services, Azure PostgreSQL Server, Azure Key Vault, Azure Storage account, Azure Service Bus, and so forth, in the private network.
- Deploys Azure API Management service and publishes the endpoint for STAC FastAPI.
- Packages the code and its dependencies, builds the Docker container images, and pushes them to Azure Container Registry.



Download a Visio file for this implementation.

Next steps

If you want to start building this, we have put together a sample solution discussed briefly above. Below are some useful links to get started on STAC & model implementation.

- STAC Overview
- STAC tutorial
- Microsoft Planetary Computer API

Related resources

- Microsoft Planetary Computer lets users apply the power of the cloud to accelerate environmental sustainability and Earth science. Many of the Planetary Computer components are also open source.
- The STAC specification
- STAC FastAPI
- PySTAC
- PgSTAC
- pyPgSTAC
- NAIP
- FGDC

Glossary

STAC term	Definition
Asset	Any file that represents spaceborne data captured in a certain space and time.
STAC Specification	Allows you to describe the geospatial data so it can be easily indexed and discovered.
STAC Item	The core atomic unit, representing a single spatiotemporal asset as a GeoJSON feature plus metadata like datetime and reference links.
STAC Catalog	A simple, flexible JSON that provides a structure and organized the metadata like STAC items, collections and other catalogs.
STAC Collection	Provides additional information such as the extents, license, keywords, providers, and so forth, that describe STAC Items within the Collection.
STAC API	Provides a RESTful endpoint that enables search of STAC Items, specified in OpenAPI.

Process Synthetic Aperture Radar (SAR) data in Azure

Article • 10/20/2022

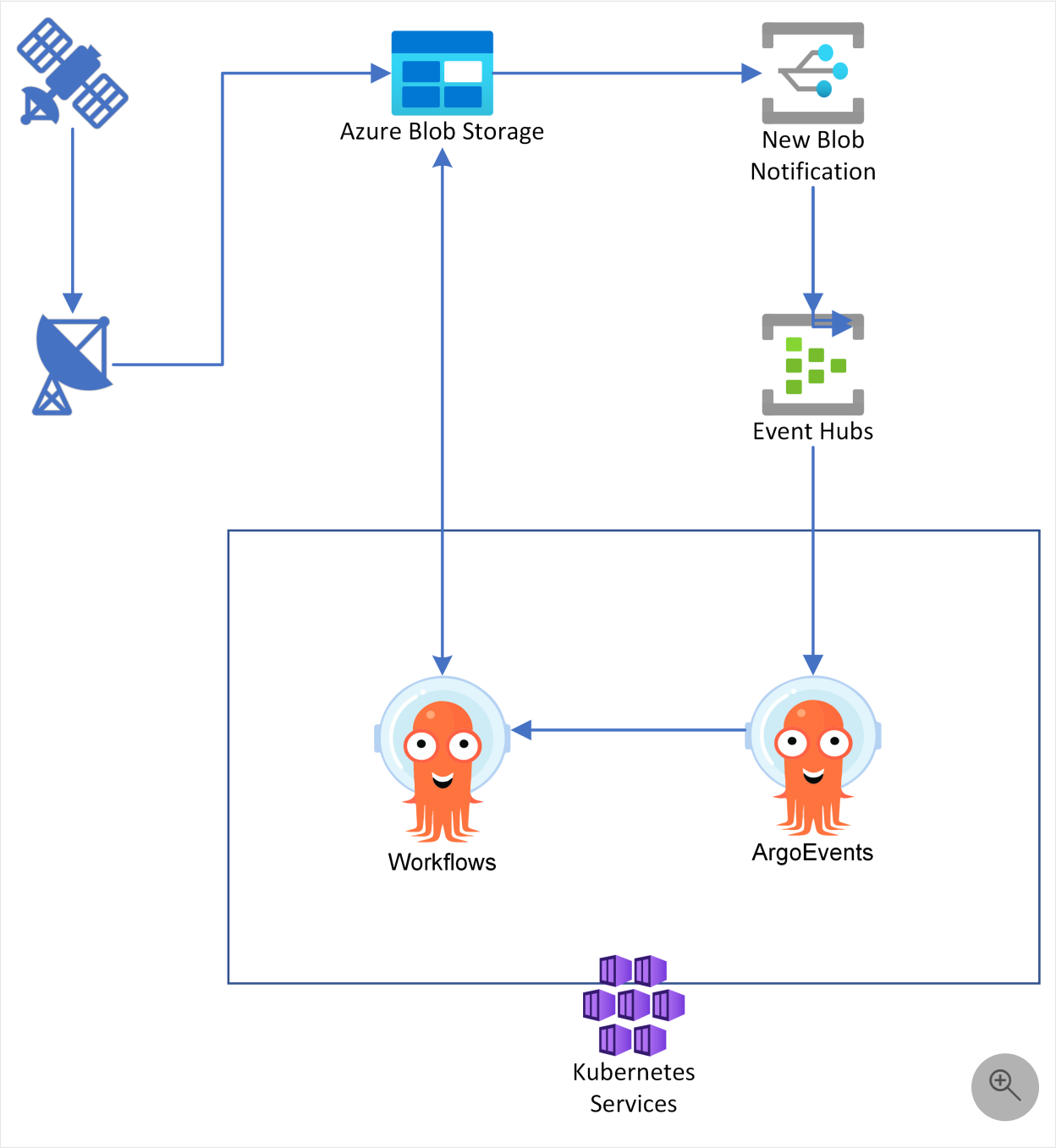
SAR is a form of radar that is used to create two-dimensional images of three-dimensional reconstructions of objects, such as landscapes. SAR uses the motion of the radar antenna over a target to provide finer spatial resolution than conventional stationary beam-scanning radars.

Processing

Remote Sensing and or SAR data has always been processed in a linear way because of the way the algorithms were written. Historically, the data was processed on single, and or power machines, which could only be scaled vertically. There was limited way to scale this process vertically and horizontally since the machines that were used to process the data were expensive. Due to increased cost, it wasn't possible to process this data in real-time or near real-time. After looking into the problem space, we were able to come up with alternative ways to scale this process horizontally.

Scaling using AKS and Argo Workflows

SAR data processing, especially raw L0 processing, typically involves vendor-specific tooling rather than open-source software. As such, a scalable processing pipeline must be able to execute vendor-specific binaries as-is instead of relying on access to source code to change the algorithm and scale out using technologies such as Apache Spark. Containerization allows for vendor supplied binaries to be wrapped in a container, and then run at scale. While the performance of processing a given image won't increase, many images can be processed in parallel. Azure Kubernetes Service is a natural fit for executing containerized software at scale. Argo Workflows provides a low overhead Kubernetes-native approach to execute pipelines on any Kubernetes cluster. This architecture allows for horizontal scaling of a processing pipeline that utilizes vendor provided binaries and/or open-source software. While processing of any individual file or image won't occur any faster, many files can be processed simultaneously in parallel. With the flexibility of AKS, each step in the pipeline can execute on the hardware best suited for the tool, for example, GPU, high core count, or increased memory.



Raw products are received by a ground station application, which, in turn, writes the data into Azure Blob Storage. Using an Azure Event Grid subscription, a notification is supplied to Azure Event Hubs when a new product image is written to blob storage. Argo Events, running on Azure Kubernetes Service, subscribes to the Azure Event Hubs notification and upon receipt of the event, triggers an Argo Workflows workflow to process the image.

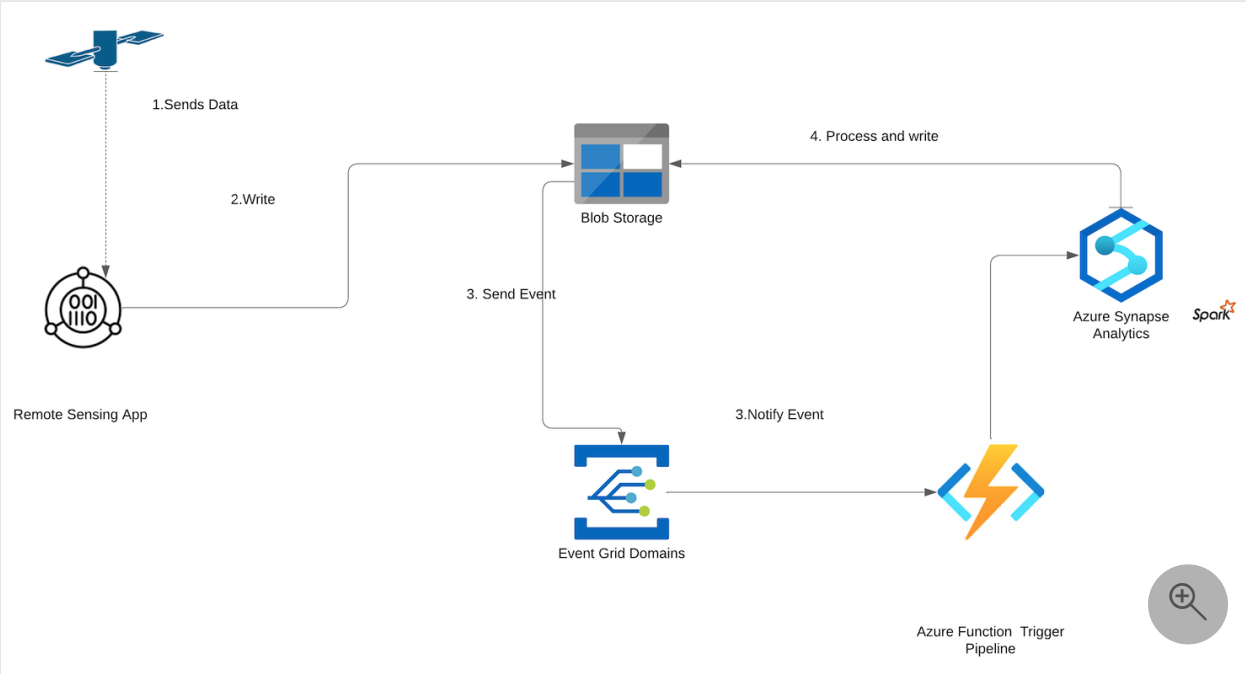
Argo Workflows are specified using a Kubernetes custom resource definition that allows a DAG or simple pipeline to be created by defining Kubernetes objects. Each step in the pipeline/DAG can run a Kubernetes Pod that performs work. The Pod may run a simple shell script or execute code in a custom container including executing vendor-specific tools to process the remote sensor products. Since each step in the pipeline is a different Kubernetes object, normal Kubernetes resource requests are used to specify

the requirements of the step. For example, a vendor-specific tool may require a GPU or node with high memory and/or cores to complete its work. These requirements can be specified using Kubernetes resource requests, and Kubernetes affinity and/or nodeSelectors. Kubernetes will map these requests to nodes that are able to satisfy the needs, provided such nodes exist.

With Azure Kubernetes Service, this typically involves creating node pools with the appropriate Azure compute SKU to meet the needs of potential pipelines. These node pools can be set to auto scale so that resources aren't consumed when pipeline steps requiring them aren't running.

Processing using Azure Synapse

The approach to use Azure Synapse is slightly different than a normal pipeline. Typically, lots of data processing firms already have algorithms that are processing the data. They may not want to rewrite the algorithms that are already written but they may need a way to scale those algorithms horizontally. What we are showing here's an approach using which they can easily run their code on distributed framework like Apache Spark and not have to worry about dealing with all the complexities one would when working with Distributed system. We're taking advantage of vectorization and SIMD architecture where we're processing more than one row at a time instead of processing one row at a time. These features are specific to Apache Spark DataFrame and JVM



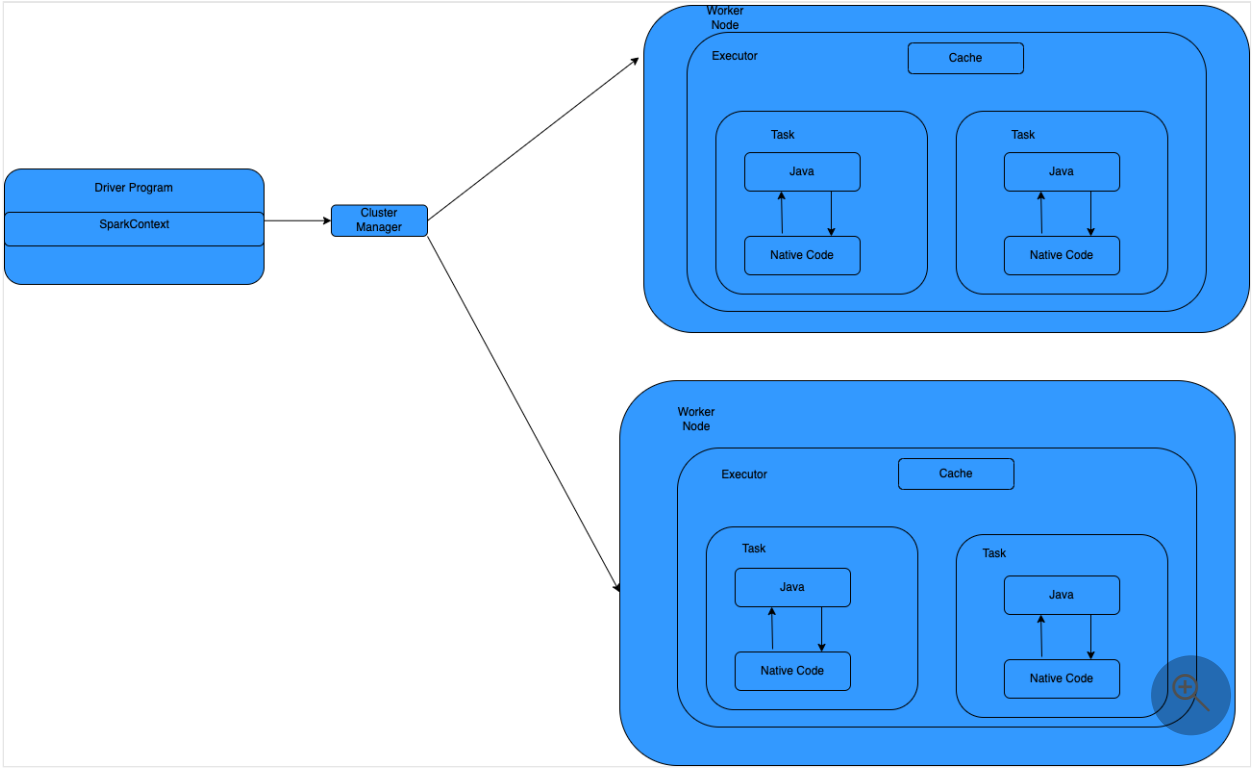
Data ingestion

Remote Sensing Data is sent to a ground station. The ground station app collects the data and Writes to Blob Storage.

Data transformation

1. Blob Storage sends an event to Event-Grid about the file being created.
2. Event-Grid Notifies the function registered to receive the event.
3. The function triggers an Azure Synapse Spark Pipeline. This pipeline has the native library and the configuration required to run the spark job. The Spark Job performs the heavy computation and writes the result to the blob storage where it can be further used by any downstream processes.

Under this approach using Apache Spark, we're gluing the library that has algorithms with JNA. JNA requires you to define the interfaces for your native code and does the heavy lifting to converting your data to and from the native library to usable Java Types. Now without any major rewriting, we can distribute the computation of data on nodes vs a single machine. Typical Spark Execution under this model looks like as follows.



Considerations

Pool size consideration

The following section outlines in detail as to how to choose a pool size for the job.

Size	Cores	Memory (GB)	Nodes	Executor	Cost (USD)
Small	4	32	20-200	2-100	11.37 to 113.71
Medium	8	64	20-200	2-100	22.74 to 227.42
Large	16	128	20-200	2-100	45.48 to 454.85
XLarge	32	256	20-200	2-100	90.97 to 909.70
XXLarge	64	512	20-200	2-100	181.94 to 1819.39

To process 1 year's worth of data, which is around 610 GB of remote sensing data, following are the metrics that were captured. These metrics are specific to the processing algorithm that was used. It only showcases and exhibits how the process can be horizontally scaled for Batch processing and for Real time processing.

Size	Time(mins)
Small	120
Medium	80
Large	67
XLarge	50
XXLarge	40

Spark configuration

Property Name	Value
Spark.driver.maxResultSize	2g
Spark.kryoserializer.buffer.max	2000
Spark.sql.shuffle.partitions	1000

The above configuration was used in the BYOLB use case as there was lot of data that was moved from the executor and the driver nodes. The default configurations weren't enough to handle the use case where we were moving the results as part of DataFrame. We could have tried broadcasting the data but since these were processed as a part of DataFrame broadcasting the values wasn't chosen as we wanted to transform each row of the DataFrame.

Spark version

We were using Apache Spark 3.1 with Scala 2.12 to develop our pipelines. This version is compatible with Java 11 which has the Garbage collector improvements over Java 8.

Data abstraction

DataFrames

- Best choice in most situations.
- Provides query optimization through Catalyst.
- Whole-stage code generation.
- Direct memory access.
- Low garbage collection (GC) overhead.
- Not as developer-friendly as Datasets, as there 's no compile-time checks or domain object programming.

RDDs

- You don't need to use RDDs, unless you need to build a new custom RDD.
- No query optimization through Catalyst.
- No whole-stage code generation.
- High GC overhead.
- Must use Spark 1.x legacy APIs.

Potential use cases

- Digital Signal Processing
- Operations on Raw Satellite Data.
- Image manipulation and processing.
- Compute heavy tasks that want to be distributed.

Contributors

This article is being updated and maintained by Microsoft. It was originally written by the following contributors

- Harjit Singh | Senior Engineering Architect
- Brian Loss | Principal Engineering Architect

Additional contributors:

- Nikhil Manchanda | Principal Engineering Manager
- Billie Rinaldi | Principal Engineering Manager
- Joey Frazee | Principal Engineering Manager
- Katy Smith | Data Scientist
- Steve Truitt | Principal Program Manager

Next steps

- Azure Maps Geospatial Services
- Getting geospatial insights from big data using SynapseML
- Get started with Azure Synapse Analytics
- Explore Azure Synapse Studio
- SpaceBorne Data Analysis

See also

- Geospatial data processing and analytics
- Geospatial analysis for the telecommunications industry
- Big data architectures

References

- Azure Synapse
- Apache Spark
- Argo

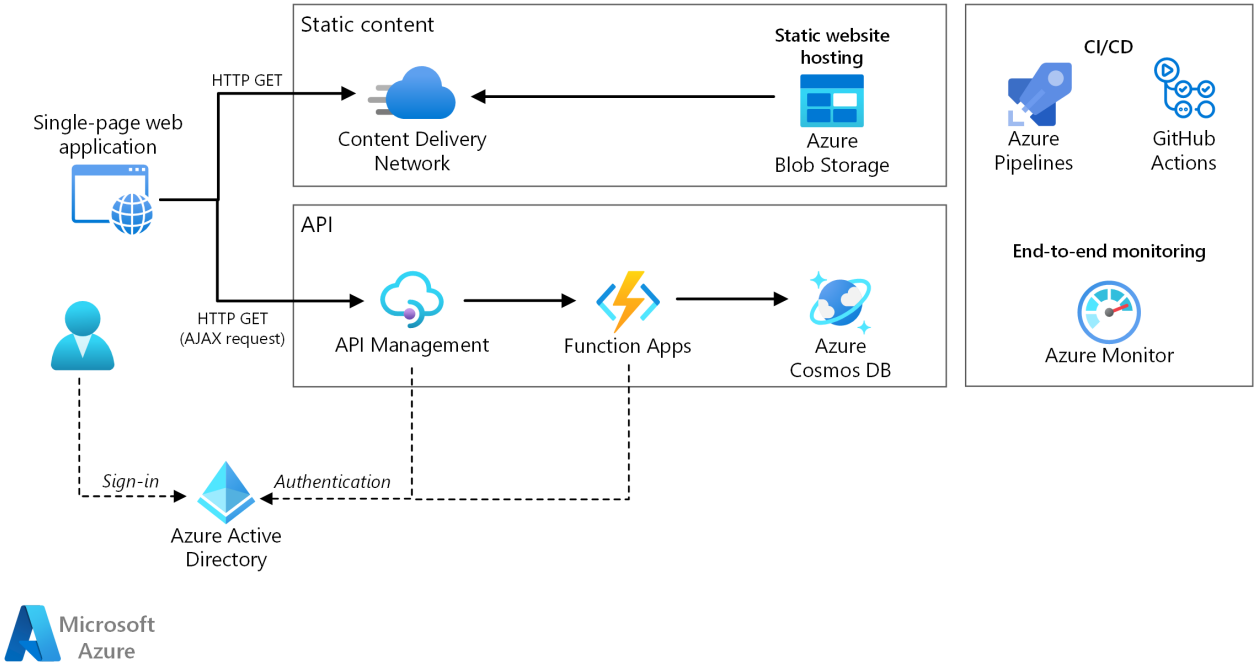
Serverless web application

Microsoft Entra ID | Azure API Management | Azure Blob Storage | Azure Content Delivery Network | Azure Functions | Azure Monitor

This reference architecture shows a serverless web application. The application serves static content from Azure Blob Storage, and implements an API using Azure Functions. The API reads data from Azure Cosmos DB and returns the results to the web app.

Two reference implementations for this architecture are available on GitHub: Drone Delivery App (ARM & Azure Pipelines) and To Do App (Bicep & GitHub Actions).

Architecture



Download a Visio file of this architecture.

The term serverless has two distinct but related meanings:

- **Backend as a service** (BaaS). Back-end cloud services, such as databases and storage, provide APIs that enable client applications to connect directly to these services.
- **Functions as a service** (FaaS). In this model, a "function" is a piece of code that is deployed to the cloud and runs inside a hosting environment that completely abstracts the servers that run the code.

Both definitions have in common the idea that developers and DevOps personnel don't need to deploy, configure, or manage servers. This reference architecture focuses on FaaS using Azure Functions, although serving web content from Azure Blob Storage could be an example of BaaS. Some important characteristics of FaaS are:

1. Compute resources are allocated dynamically as needed by the platform.
2. Consumption-based pricing: You are charged only for the compute resources used to execute your code.
3. The compute resources scale on demand based on traffic, without the developer needing to do any configuration.

Functions are executed when an external trigger occurs, such as an HTTP request or a message arriving on a queue. This makes an event-driven architecture style natural for serverless architectures. To coordinate work between components in the architecture, consider using message brokers or pub/sub patterns. For help with choosing between messaging technologies in Azure, see Choose between Azure services that deliver messages.

Components

The architecture consists of the following components:

Blob Storage. Static web content, such as HTML, CSS, and JavaScript files, are stored in Azure Blob Storage and served to clients by using static website hosting. All dynamic interaction happens through JavaScript code making calls to the back-end APIs. There is no server-side code to render the web page. Static website hosting supports index documents and custom 404 error pages.

CDN. Use Azure Content Delivery Network (CDN) to cache content for lower latency and faster delivery of content, as well as providing an HTTPS endpoint.

Function Apps. 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, the function is invoked when a client makes an HTTP request. The request is always routed through an API gateway, described below.

API Management. Azure API Management provides an API gateway that sits in front of the HTTP function. You can use API Management to publish and manage APIs used by client applications. Using a gateway helps to decouple the front-end application from the back-end APIs. For example, API Management can rewrite URLs, transform requests before they reach the back end, set request or response headers, and so forth.

API Management can also be used to implement cross-cutting concerns such as:

- Enforcing usage quotas and rate limits
- Validating OAuth tokens for authentication
- Enabling cross-origin requests (CORS)
- Caching responses
- Monitoring and logging requests

If you don't need all of the functionality provided by API Management, another option is to use Functions Proxies. This feature of Azure Functions lets you define a single API surface for multiple function apps, by creating routes to back-end functions. Function proxies can also perform limited transformations on the HTTP request and response. However, they don't provide the same rich policy-based capabilities of API Management.

Azure Cosmos DB. Azure Cosmos DB is a multi-model database service. For this scenario, the function application fetches documents from Azure Cosmos DB in response to HTTP GET requests from the client.

Microsoft Entra ID (Microsoft Entra ID). Users sign into the web application by using their Microsoft Entra ID credentials. Microsoft Entra ID returns an access token for the API, which the web application uses to authenticate API requests (see Authentication).

Azure Monitor. Azure 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. It also collected application logs.

Azure Pipelines. Azure Pipelines is a continuous integration (CI) and continuous delivery (CD) service that builds, tests, and deploys the application.

GitHub Actions. Workflow is an automated process (CI/CD) that you set up in your GitHub repository. You can build, test, package, release, or deploy any project on GitHub with a workflow.

Scenario details

Potential use cases

This drone-delivery solution is ideal for the aircraft, aviation, aerospace, and robotics industries.

Recommendations

Function App plans

Azure Functions supports two hosting models. With the **consumption plan**, compute power is automatically allocated when your code is running. With the **App Service** plan, a set of VMs are allocated for your code. The App Service plan defines the number of VMs and the VM size.

Note that the App Service plan is not strictly *serverless*, according to the definition given above. The programming model is the same, however — the same function code can run in both a consumption plan and an App Service plan.

Here are some factors to consider when choosing which type of plan to use:

- **Cold start**. With the consumption plan, a function that hasn't been invoked recently will incur some additional latency the next time it runs. This additional latency is due to allocating and preparing the runtime environment. It is usually on the order of seconds but depends on several factors, including the number of dependencies that need to be loaded. For more information, see Understanding Serverless Cold Start. Cold start is usually more of a concern for interactive workloads (HTTP triggers) than asynchronous message-driven workloads (queue or event hubs triggers), because the additional latency is directly observed by users.
- **Timeout period**. In the consumption plan, a function execution times out after a configurable period of time (to a maximum of 10 minutes)
- **Virtual network isolation**. Using an App Service plan allows functions to run inside of an App Service Environment, which is a dedicated and isolated hosting environment.
- **Pricing model**. The consumption plan is billed by the number of executions and resource consumption (memory × execution time). The App Service plan is billed hourly based on VM instance SKU. Often, the consumption plan can be cheaper than an App Service plan, because you pay only for the compute resources that you use. This is especially true if your traffic experiences peaks and troughs. However, if an application experiences constant high-volume throughput, an App Service plan may cost less than the consumption plan.
- **Scaling**. A big advantage of the consumption model is that it scales dynamically as needed, based on the incoming traffic. While this scaling occurs quickly, there is still a ramp-up period. For some workloads, you might want to deliberately overprovision the VMs, so that you can handle bursts of traffic with zero ramp-up time. In that case, consider an App Service plan.

Function App boundaries

A *function app* hosts the execution of one or more *functions*. You can use a function app to group several functions together as a logical unit. Within a function app, the functions share the same application settings, hosting plan, and deployment lifecycle. Each function app has its own hostname.

Use function apps to group functions that share the same lifecycle and settings. Functions that don't share the same lifecycle should be hosted in different function apps.

Consider taking a microservices approach, where each function app represents one microservice, possibly consisting of several related functions. In a microservices architecture, services should have loose coupling and high functional cohesion. *Loosely* coupled means you can change one service without requiring other services to be updated at the same time. *Cohesive* means a service has a single, well-defined purpose. For more discussion of these ideas, see Designing microservices: Domain analysis.

Function bindings

Use Functions bindings when possible. Bindings provide a declarative way to connect your code to data and integrate with other Azure services. An input binding populates an input parameter from an external data source. An output binding sends the function's return value to a data sink, such as a queue or database.

For example, the `GetStatus` function in the reference implementation uses the Azure Cosmos DB input binding. This binding is configured to look up a document in Azure Cosmos DB, using query parameters that are taken from the query string in the HTTP request. If the document is found, it is passed to the function as a parameter.

C#

```
[FunctionName("GetStatusFunction")]
public static Task<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,
    ILogger log)
{
    ...
}
```

By using bindings, you don't need to write code that talks directly to the service, which makes the function code simpler and also abstracts the details of the data source or sink. In some cases, however, you may need more complex logic than the binding provides. In that case, use the Azure client SDKs directly.

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

Functions. For the consumption plan, the HTTP trigger scales based on the traffic. There is a limit to the number of concurrent function instances, but each instance can process more than one request at a time. For an App Service plan, the HTTP trigger scales according to the number of VM instances, which can be a fixed value or can autoscale based on a set of autoscaling rules. For information, see Azure Functions scale and hosting.

Azure Cosmos DB. Throughput capacity for Azure Cosmos DB is measured in Request Units (RU). A 1-RU throughput corresponds to the throughput need to GET a 1KB document. In order to scale an Azure Cosmos DB container past 10,000 RU, you must specify a partition key when you create the container and include the partition key in every document that you create. For more information about partition keys, see Partition and scale in Azure Cosmos DB.

API Management. API Management can scale out and supports rule-based autoscaling. The scaling process takes at least 20 minutes. If your traffic is bursty, you should provision for the maximum burst traffic that you expect. However, autoscaling is useful for handling hourly or daily variations in traffic. For more information, see Automatically scale an Azure API Management instance.

Disaster recovery

The deployment shown here resides in a single Azure region. For a more resilient approach to disaster-recovery, take advantage of the geo-distribution features in the various services:

- API Management supports multi-region deployment, which can be used to distribute a single API Management instance across any number of Azure regions.

For more information, see How to deploy an Azure API Management service instance to multiple Azure regions.

- Use Traffic Manager to route HTTP requests to the primary region. If the Function App running in that region becomes unavailable, Traffic Manager can fail over to a secondary region.

- 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.

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.

Authentication

The `GetStatus` API in the reference implementation uses Microsoft Entra ID to authenticate requests. Microsoft Entra ID supports the OpenID Connect protocol, which is an authentication protocol built on top of the OAuth 2 protocol.

In this architecture, the client application is a single-page application (SPA) that runs in the browser. This type of client application cannot keep a client secret or an authorization code hidden, so the implicit grant flow is appropriate. (See Which OAuth 2.0 flow should I use?). Here's the overall flow:

1. The user clicks the "Sign in" link in the web application.
2. The browser is redirected the Microsoft Entra sign-in page.
3. The user signs in.
4. Microsoft Entra ID redirects back to the client application, including an access token in the URL fragment.
5. When the web application calls the API, it includes the access token in the Authentication header. The application ID is sent as the audience ('aud') claim in the access token.
6. The back-end API validates the access token.

To configure authentication:

- Register an application in your Microsoft Entra tenant. This generates an application ID, which the client includes with the login URL.

- Enable Microsoft Entra authentication inside the Function App. For more information, see Authentication and authorization in Azure App Service.
- Add the validate-jwt policy to API Management to pre-authorize the request by validating the access token.

For more details, see the GitHub readme.

It's recommended to create separate app registrations in Microsoft Entra ID for the client application and the back-end API. Grant the client application permission to call the API. This approach gives you the flexibility to define multiple APIs and clients and control the permissions for each.

Within an API, use scopes to give applications fine-grained control over what permissions they request from a user. For example, an API might have `Read` and `Write` scopes, and a particular client app might ask the user to authorize `Read` permissions only.

Authorization

In many applications, the back-end API must check whether a user has permission to perform a given action. It's recommended to use claims-based authorization, where information about the user is conveyed by the identity provider (in this case, Microsoft Entra ID) and used to make authorization decisions. For example, when you register an application in Microsoft Entra ID, you can define a set of application roles. When a user signs into the application, Microsoft Entra ID includes a `roles` claim for each role that the user has been granted, including roles that are inherited through group membership.

The ID token that Microsoft Entra ID returns to the client contains some of the user's claims. Within the function app, these claims are available in the X-MS-CLIENT-PRINCIPAL header of the request. However, it's simpler to read this information from binding data. For other claims, use Microsoft Graph to query Microsoft Entra ID. (The user must consent to this action when signing in.)

For more information, see Working with client identities.

CORS

In this reference architecture, the web application and the API do not share the same origin. That means when the application calls the API, it is a cross-origin request. Browser security prevents a web page from making AJAX requests to another domain.

This restriction is called the *same-origin policy* and prevents a malicious site from reading sensitive data from another site. To enable a cross-origin request, add a Cross-Origin Resource Sharing (CORS) policy to the API Management gateway:

```XML
<cors allow-credentials="true">
    <allowed-origins>
        <origin>[Website URL]</origin>
    </allowed-origins>
    <allowed-methods>
        <method>GET</method>
    </allowed-methods>
    <allowed-headers>
        <header>*</header>
    </allowed-headers>
</cors>
```

In this example, the **allow-credentials** attribute is **true**. This authorizes the browser to send credentials (including cookies) with the request. Otherwise, by default the browser does not send credentials with a cross-origin request.

> **Note**
>
> Be very careful about setting **allow-credentials** to **true**, because it means a website can send the user's credentials to your API on the user's behalf, without the user being aware. You must trust the allowed origin.

Enforce HTTPS

For maximum security, require HTTPS throughout the request pipeline:

- **CDN**. Azure CDN supports HTTPS on the `*.azureedge.net` subdomain by default. To enable HTTPS in the CDN for custom domain names, see Tutorial: Configure HTTPS on an Azure CDN custom domain.
- **Static website hosting**. Enable the "Secure transfer required" option on the Storage account. When this option is enabled, the storage account only allows requests from secure HTTPS connections.
- **API Management**. Configure the APIs to use HTTPS protocol only. You can configure this in the Azure portal or through a Resource Manager template:

  ```JSON
  ```

```
{
    "apiVersion": "2018-01-01",
    "type": "apis",
    "name": "dronedeliveryapi",
    "dependsOn": [
        "[concat('Microsoft.ApiManagement/service/',
variables('apiManagementServiceName'))]"
    ],
    "properties": {
        "displayName": "Drone Delivery API",
        "description": "Drone Delivery API",
        "path": "api",
        "protocols": [ "HTTPS" ]
    },
    ...
}
```

- **Azure Functions**. Enable the "HTTPS Only" setting.

Lock down the function app

All calls to the function should go through the API gateway. You can achieve this as follows:

- Configure the function app to require a function key. The API Management gateway will include the function key when it calls the function app. This prevents clients from calling the function directly, bypassing the gateway.
- The API Management gateway has a static IP address. Restrict the Azure Function to allow only calls from that static IP address. For more information, see Azure App Service Static IP Restrictions. (This feature is available for Standard tier services only.)

Protect application secrets

Don't store application secrets, such as database credentials, in your code or configuration files. Instead, use App settings, which are stored encrypted in Azure. For more information, see Security in Azure App Service and Azure Functions.

Alternatively, you can store application secrets in Key Vault. This allows you to centralize the storage of secrets, control their distribution, and monitor how and when secrets are being accessed. For more information, see Configure an Azure web application to read a secret from Key Vault. However, note that Functions triggers and bindings load their configuration settings from app settings. There is no built-in way to configure the triggers and bindings to use Key Vault secrets.

DevOps

Front-end deployment

The front end of this reference architecture is a single page application, with JavaScript accessing the serverless back-end APIs, and static content providing a fast user experience. The following are some important considerations for such an application:

- Deploy the application uniformly to users over a wide geographical area with a global-ready CDN, with the static content hosted on the cloud. This avoids the need for a dedicated web server. Read Integrate an Azure storage account with Azure CDN to get started. Secure your application with HTTPS. Read the Best practices for using content delivery networks for additional recommendations.
- Use a fast and reliable CI/CD service such as Azure Pipelines or GitHub Actions, to automatically build and deploy every source change. The source must reside in an online version control system. For more details on Azure Pipelines, read Create your first pipeline. To learn more on GitHub Actions for Azure, see Deploy apps to Azure.
- Compress your website files to reduce the bandwidth consumption on the CDN and improve performance. Azure CDN allows compression on the fly on the edge servers. Alternatively, the deploy pipeline in this reference architecture compresses the files before deploying them to the Blob storage. This reduces the storage requirement, and gives you more freedom to choose the compression tools, regardless of any CDN limitations.
- The CDN should be able to purge its cache to ensure all users are served the freshest content. A cache purge is required if the build and deploy processes are not atomic, for example, if they replace old files with newly built ones in the same origin folder.
- A different cache strategy such as versioning using directories, may not require a purge by the CDN. The build pipeline in this front-end application creates a new directory for each newly built version. This version is uploaded as an atomic unit to the Blob storage. The Azure CDN points to this new version only after a completed deployment.
- Increase the cache TTL by caching resource files for a longer duration, spanning months. To make sure the cached files are updated when they do change, fingerprint the filenames when they are rebuilt. This front-end application fingerprints all files except for public-facing files such as *index.html*. Since the index.html is updated frequently, it reflects the changed filenames causing a cache refresh. See the Manage expiration of web content in Azure CDN for more information.

Back-end deployment

To deploy the function app, we recommend using package files ("Run from package"). Using this approach, you upload a zip file to a Blob Storage container and the Functions runtime mounts the zip file as a read-only file system. This is an atomic operation, which reduces the chance that a failed deployment will leave the application in an inconsistent state. It can also improve cold start times, especially for Node.js apps, because all of the files are swapped at once.

API versioning

An API is a contract between a service and clients. In this architecture, the API contract is defined at the API Management layer. API Management supports two distinct but complementary versioning concepts:

- *Versions* allow API consumers to choose an API version based on their needs, such as v1 versus v2.
- *Revisions* allow API administrators to make non-breaking changes in an API and deploy those changes, along with a change log to inform API consumers about the changes.

If you make a breaking change in an API, publish a new version in API Management. Deploy the new version side-by-side with the original version, in a separate Function App. This lets you migrate existing clients to the new API without breaking client applications. Eventually, you can deprecate the previous version. API Management supports several versioning schemes: URL path, HTTP header, or query string. For more information about API versioning in general, see Versioning a RESTful web API.

For updates that are not breaking API changes, deploy the new version to a staging slot in the same Function App. Verify the deployment succeeded and then swap the staged version with the production version. Publish a revision in API Management.

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. Consider these points to optimize cost of this architecture.

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 VMs are allocated for your code. This plan defines the number of VMs and the VM size.

In this architecture, a function is invoked when a client makes an HTTP request. Because a constant high-volume throughput is not expected in this use case, **consumption plan** is recommended because you pay only for the compute resources you use.

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. The price is based on the capacity in RU/s that you reserve.

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, the function application fetches documents from Azure Cosmos DB in response to HTTP GET requests from the client. Azure Cosmos DB is cost effective in this case because reading operations are significantly cheaper than write operations expressed on RU/s.

Content Delivery Network

Billing rate may differ depending on the billing region based on the location of the source server delivering the content to the end user. The physical location of the client is not the billing region. Any HTTP or HTTPS request that hits the CDN is a billable event, which includes all response types: success, failure, or other. Different responses may generate different traffic amounts.

In this reference architecture the deployment resides in a single Azure region.

To lower costs, consider increasing the cache TTL by caching resource files for a longer duration and setting the longest TTL possible on your content.

For more information, see the Cost section in Microsoft Azure Well-Architected Framework.

Deploy this scenario

To deploy the reference implementation for this architecture, see the GitHub readme.

Next steps

Product documentation:

- What is Azure Blob Storage?
- Azure Content Delivery Network (CDN)
- Introduction to Azure Functions
- About API Management
- Welcome to Azure Cosmos DB
- Microsoft Entra ID
- Azure Monitor overview
- What is Azure Pipelines?

Learn modules:

- Choose the best Azure serverless technology for your business scenario
- Create serverless logic with Azure Functions

Related resources

To learn more about the reference implementation, read Code walkthrough: Serverless application with Azure Functions.

Related guidance:

- Best practices for using CDNs
- Static Content Hosting pattern

Spaceborne data analysis with Azure Synapse Analytics

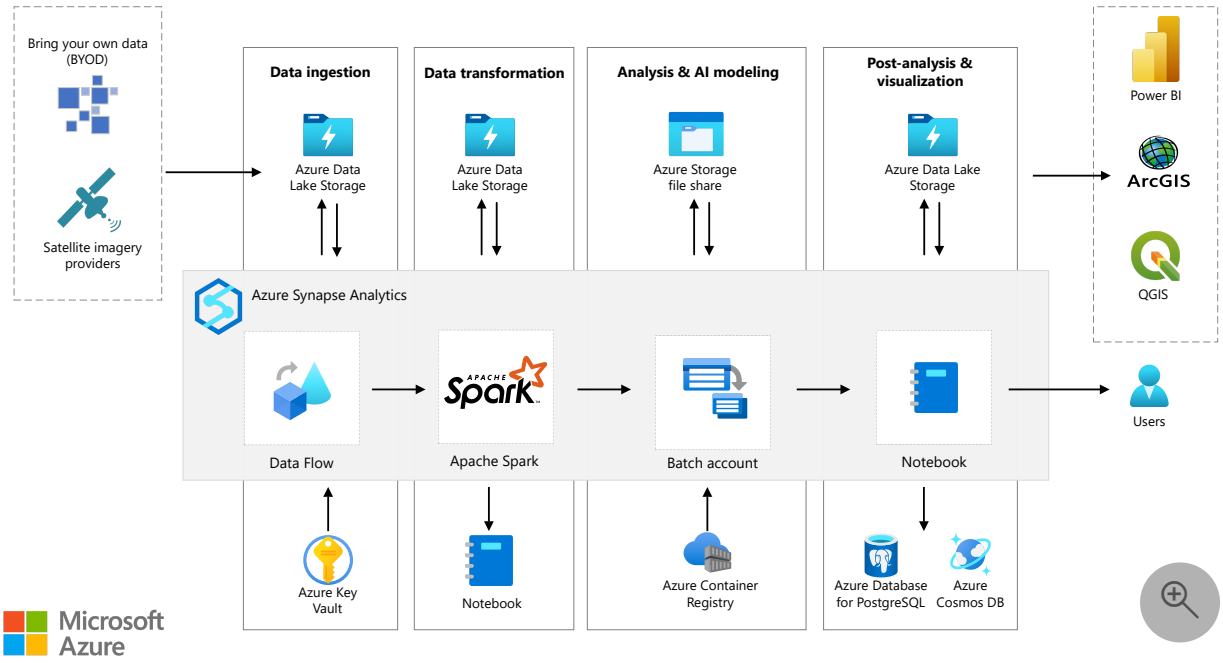
Azure Synapse Analytics | Azure AI services | Azure Computer Vision | Azure Batch | Azure Storage

This architecture is designed to show an end-to-end implementation that involves extracting, loading, transforming, and analyzing spaceborne data by using geospatial libraries and AI models with Azure Synapse Analytics. This article also shows how to integrate geospatial-specific Azure Cognitive Services models, AI models from partners, bring-your-own-data, and AI models that use Azure Synapse Analytics. The intended audience for this document is users with intermediate skill levels in working with geospatial or spaceborne data.

An implementation of this architecture is available on GitHub.

Apache®, Apache Spark, Spark, the Spark logo, Apache Sedona, Apache Incubator, the Apache feather logo and the Apache Incubator project 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.

Architecture



Download a Visio file of this architecture.

Dataflow

The following sections describe the stages in the architecture.

Data ingestion

Spaceborne data is pulled from data sources like Airbus, NAIP/USDA (via the Planetary Computer API), and Maxar. Data is ingested into Azure Data Lake Storage.

Azure Synapse Analytics provides various pipelines and activities, like Web activity, Data Flow activity, and Custom activities, to connect to these sources and copy the data into Data Lake Storage.

Azure Synapse Custom activities run your customized code logic on an Azure Batch pool of virtual machines or in Docker-compatible containers.

Data transformation

The data is processed and transformed into a format that analysts and AI models can consume. Geospatial libraries, including GDAL, OGR, Rasterio, and GeoPandas, are available to perform the transformation.

Azure Synapse Spark pools provide the ability to configure and use these libraries to perform the data transformations. You can also use Azure Synapse Custom activities, which use Azure Batch pools.

An Azure Synapse notebook is a web interface that you can use to create files that contain live code, visualizations, and narrative text. Notebooks are a good place to validate ideas, define transformations, and do quick experiments to get insights from your data and build a pipeline. In the sample code, the GDAL library is used in a Spark pool to perform data transformations. For more information, see the sample code section of this article.

The sample solution implements this pipeline from this data transformation step. The sample is written with the assumption that data is copied in Data Lake Storage by the data ingestion methods described earlier. It demonstrates implementation of this pipeline for raster data processing.

Analysis and execution of AI models

The Azure Synapse notebook environment analyzes and runs AI models.

AI models developed with services like the Cognitive Services Custom Vision model, trained in their own environment, and packaged as Docker containers are available in the Azure Synapse environment.

In the Azure Synapse environment, you can also run AI models that are available from partners for various capabilities like object detection, change detection, and land classification. These models are trained in their own environment and packaged as Docker containers.

Azure Synapse can run such AI models via a Custom activity that runs code in Batch pools as executables or Docker containers. The sample solution demonstrates how to run a Custom Vision AI model as part of an Azure Synapse pipeline for object detection over a specific geospatial area.

Post-analysis and visualization

- For further analysis and visualization, output from analysis and execution of the AI models can be stored in Data Lake Storage, data-aware databases like Azure Database for PostgreSQL, or Azure Cosmos DB. The sample solution shows how to transform AI model output and store it as GeoJSON data in Data Lake Storage and Azure Database for PostgreSQL. You can retrieve and query the output from there.
- For visualization:
  - You can use licensed tools like ArcGIS Desktop or open-source tools like QGIS.
  - You can use Power BI to access GeoJSON from various data sources and visualize the geographic information system (GIS) data.
  - You can use client-side geospatial JavaScript-based libraries to visualize the data in web applications.

Components

Data sources

- **Imagery providers.**
  - Airbus
  - NAIP/USDA (via the Planetary Computer API)
  - Maxar
- **Bring your own data.** Copy your own data to Data Lake Storage.

Data ingestion

- Azure Synapse Analytics is a limitless analytics service that brings together data integration, enterprise data warehousing, and big data analytics. Azure Synapse contains the same Data Integration engine and experiences as Azure Data Factory, so you can create at-scale ETL pipelines without leaving Azure Synapse.
- Azure Data Lake Storage is dedicated to big data analytics, and is built on Azure Blob Storage.
- Azure Batch enables you to run and scale a large number of batch computing jobs on Azure. Batch tasks can run directly on virtual machines (nodes) in a Batch pool, but you can also set up a Batch pool to run tasks in Docker-compatible containers on the nodes.
  - An Azure Synapse Custom activity runs customized code logic on an Azure Batch pool of virtual machines or in Docker containers.
- 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.

Data transformation

The following geospatial libraries and packages are used together for transformations. These libraries and packages are installed in a serverless Spark pool, which is then attached to an Azure Synapse notebook. For information on installing the libraries, see Install geospatial packages in an Azure Synapse Spark pool, later in this article.

- **Geospatial libraries**
  - GDAL is a library of tools for manipulating spaceborne data. GDAL works on raster and vector data types. It's a good tool to know if you're working with spaceborne data.
  - Rasterio is a module for raster processing. You can use it to read and write several different raster formats in Python. Rasterio is based on GDAL. When the module is imported, Python automatically registers all known GDAL drivers for reading supported formats.
  - GeoPandas is an open-source project that can make it easier to work with spaceborne data in Python. GeoPandas extends the data types used by Pandas to allow spatial operations on geometric types.
  - Shapely is a Python package for set-theoretic analysis and manipulation of planar features. It uses (via Python's ctypes module) functions from the widely deployed GEOS library.
  - pyproj performs cartographic transformations. It converts from longitude and latitude to native map projection x, y coordinates, and vice versa, by using PROJ.

- Azure Batch enables you to run and scale a large number of batch computing jobs on Azure.
- Azure Synapse notebooks is a web interface for creating files that contain live code, visualizations, and narrative text. You can add existing Azure Synapse notebooks to an Azure Synapse pipeline by using the Notebook activity.
- Apache Spark pool provides the ability to configure and use libraries to perform data transformations. You can add existing Spark jobs to an Azure Synapse pipeline by using the Spark Job Definition activity.

Analysis and AI modeling

- Azure Synapse provides machine learning capabilities.
- Azure Batch enables you to run and scale a large number of batch computing jobs on Azure. In this solution, the Azure Synapse Custom activity is used to run Docker-based AI models on Azure Batch pools.
- Azure Cognitive Services provides the ability to embed vision into your apps. You can use Custom Vision, a component of Cognitive Services, to customize and embed state-of-the-art computer vision image analysis for specific domains.
- You can also use bring-your-own AI models and Microsoft partner AI models like blackshark.ai.

Post-analysis and visualization links

- Azure Database for PostgreSQL is a fully managed relational database service designed for hyperscale workloads. It supports spaceborne data via the PostGIS extension.
- Azure Cosmos DB supports indexing and querying of geospatial point data that's represented in GeoJSON.
- Power BI is an interactive data visualization tool for building reports and dashboards. You can get insights on spaceborne data from Esri ArcGIS Maps.
- QGIS is a free open-source GIS for creating, editing, visualizing, analyzing, and publishing geospatial information.
- ArcGIS Desktop is licensed product provided by Esri. You can use it to create, analyze, manage, and share geographic information.

Alternatives

If you want to run containerized AI models that you can call from Azure Synapse, you can use Azure Kubernetes Service, Azure Container Instances, or Azure Container Apps.

Azure Databricks provides an alternative for hosting an analytics pipeline.

Spark in Azure HDInsight provides an alternative for using geospatial libraries in the Apache Spark environment.

Here are some alternative libraries and frameworks that you can use for spaceborne data processing:

- Apache Sedona, formerly named GeoSpark, is a cluster computing system for processing large-scale spatial data. Sedona extends Spark and Spark SQL with out-of-the-box Spatial Resilient Distributed Datasets and SpatialSQL that efficiently load, process, and analyze large-scale spatial data across machines.
- Dask for Python is a parallel computing library that scales the existing Python ecosystem.

Scenario details

Spaceborne data collection is increasingly common. For the application of artificial intelligence, stored archives of data are necessary for machine learning. The need to build a cloud-based solution for spaceborne data analysis has become more important to enable enterprises and governments to drive better-informed business and tactical decisions.

Potential use cases

This solution is ideal for the aerospace and aircraft industries. It addresses these scenarios:

- Raster data ingestion and processing
- Object detection via pre-trained AI models
- Classification of land masses via AI models
- Monitoring changes in the environment via AI models
- Derived datasets from preprocessed imagery sets
- Vector visualization / small-area consumption
- Vector data filtering and cross-data joins

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.

Operational excellence

If you collaborate by using Git for source control, you can use Synapse Studio to associate your workspace with a Git repository, Azure DevOps, or GitHub. For more information, see Source control in Synapse Studio.

- In an Azure Synapse workspace, CI/CD moves all entities from one environment (development, test, production) to another environment.
- You can use Azure DevOps release pipelines and GitHub Actions to automate the deployment of an Azure Synapse workspace to multiple environments.

Performance

- Azure Synapse supports Apache Spark 3.1.2, which is more performant than its predecessors.
- For information about Spark pool scaling and node sizes, see Spark pools in Azure Synapse Analytics.
- With Azure Batch, you can scale out intrinsically parallel for transformations submitted in an Azure Synapse Custom activity. Azure Batch supports specialized GPU-optimized VM sizes that you can use to run AI models.

Reliability

Reliability ensures your application can meet the commitments you make to your customers. For more information, see Overview of the reliability pillar.

For SLA information, see Azure Synapse SLA.

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.

See these articles for security best practices:

- Azure Synapse Analytics security: Introduction
- Azure Synapse Analytics security: Data protection
- Azure Synapse Analytics security: Access control
- Azure Synapse Analytics security: Authentication
- Azure Synapse Analytics: Network security

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.

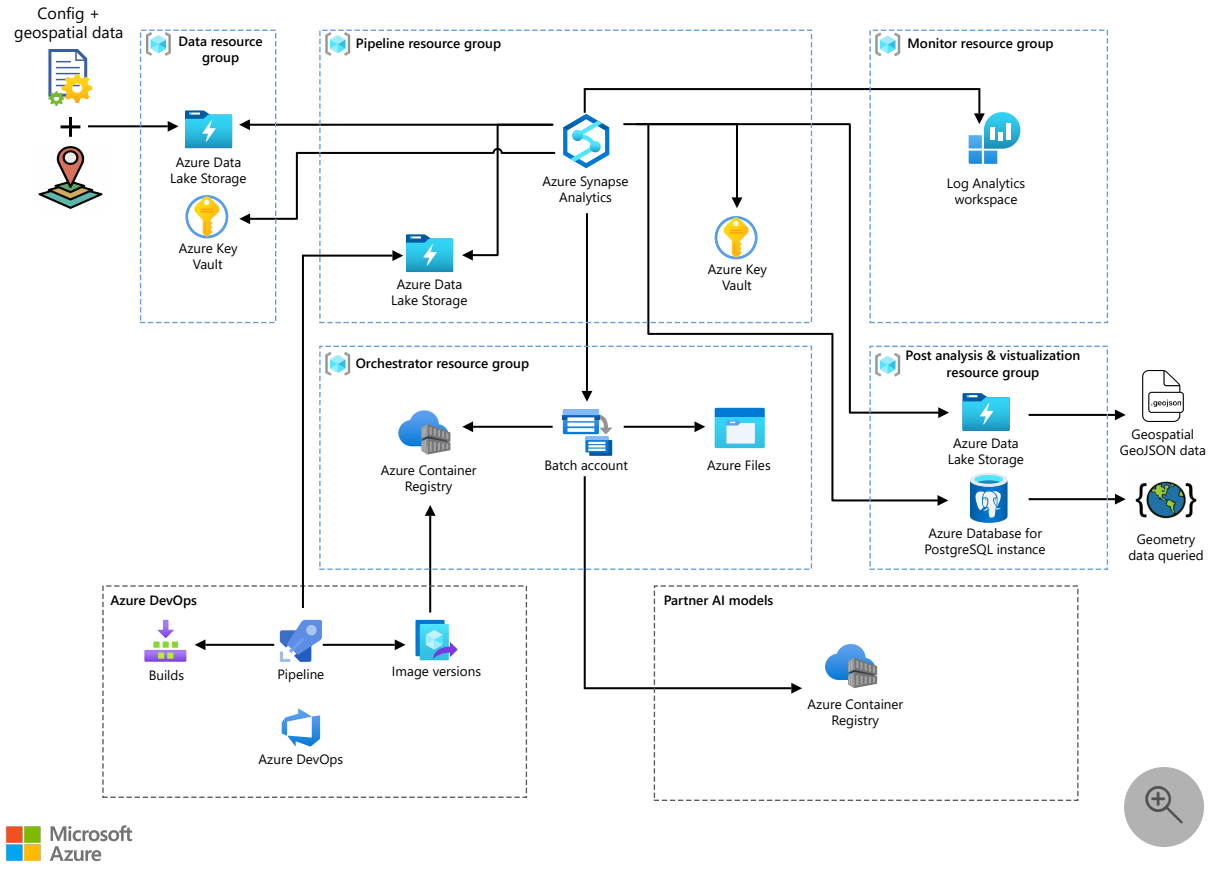
These resources provide information about pricing and cost optimization:

- Plan and manage costs for Azure Synapse
- Azure Synapse in the Azure pricing calculator
- Apache Spark pool in Azure Synapse
- Nodes and pools in Azure Batch
- Azure Batch in the Azure pricing calculator

> **Note**
>
> For pricing and license terms for partner AI models, see the partner's documentation.

Deploy this scenario

A Bicep deployment of the sample solution is available. To get started with this deployment, see these instructions.



Download a Visio file of this architecture.

Limitations

This architecture demonstrates an end-to-end geoprocessing and analytics solution that uses Azure Synapse. This sample implementation is targeted for a small to medium area of interest and limited concurrent geoprocessing of raster data.

Sample code

The following instructions describe how to read, write, and apply transformations to raster data that's stored in Azure Data Lake Storage by using a Synapse notebook. The intention is more to demonstrate the use of libraries in Synapse notebooks than to demonstrate the transformation.

Prerequisites

- Install the geospatial libraries.
- Create an Azure key vault to store secrets. In this scenario, we'll store the access key of the storage account in the key vault. For instructions, see Store credentials in Azure Key Vault.
- Create a linked service for Azure Key Vault by using Azure Synapse.

Instructions

- Print information from the raster data:

Python

```python
from osgeo import gdal
gdal.UseExceptions()
access_key = TokenLibrary.getSecret('<key-vault-name>','<secret-name>')
gdal.SetConfigOption('AZURE_STORAGE_ACCOUNT', '<storage_account_name>')
gdal.SetConfigOption('AZURE_STORAGE_ACCESS_KEY', access_key)
dataset_info = gdal.Info('/vsiadls/aoa/input/sample_image.tiff')
#/vsiadls/<container_name>/path/to/image
print(dataset_info)
```

Note

`/vsiadls/` is a file system handler that enables on-the-fly random reading of primarily non-public files that are available in Azure Data Lake Storage file systems. Prior download of the entire file isn't required. `/vsiadls/` is similar to `/vsiaz/`. It uses the same configuration options for authentication. Unlike `/vsiaz/`, `/vsiadls/` provides real directory management and Unix-style ACL support. For some features, hierarchical support needs to be turned on in Azure storage. For more information, see the **/vsiadls/ documentation**.

Output

```
Driver: GTiff/GeoTIFF
Files: /vsiadls/naip/input/sample_image.tiff
Size is 6634, 7565
Coordinate System is:
PROJCRS["NAD83 / UTM zone 16N",
    BASEGEOGCRS["NAD83",
        DATUM["North American Datum 1983",
            ELLIPSOID["GRS 1980",6378137,298.257222101,
                LENGTHUNIT["metre",1]]],
        PRIMEM["Greenwich",0,
            ANGLEUNIT["degree",0.0174532925199433]],
        ID["EPSG",4269]],
    CONVERSION["UTM zone 16N",
        METHOD["Transverse Mercator",
            ID["EPSG",9807]],
        PARAMETER["Latitude of natural origin",0,
            ANGLEUNIT["degree",0.0174532925199433],
            ID["EPSG",8801]],
        PARAMETER["Longitude of natural origin",-87,
            ANGLEUNIT["degree",0.0174532925199433],
            ID["EPSG",8802]],
```

```
        PARAMETER["Scale factor at natural origin",0.9996,
            SCALEUNIT["unity",1],
            ID["EPSG",8805]],
        PARAMETER["False easting",500000,
            LENGTHUNIT["metre",1],
            ID["EPSG",8806]],
        PARAMETER["False northing",0,
            LENGTHUNIT["metre",1],
            ID["EPSG",8807]]],
    CS[Cartesian,2],
        AXIS["(E)",east,
            ORDER[1],
            LENGTHUNIT["metre",1]],
        AXIS["(N)",north,
            ORDER[2],
            LENGTHUNIT["metre",1]],
    USAGE[
        SCOPE["Engineering survey, topographic mapping."],
        AREA["North America - between 90°W and 84°W - onshore and
offshore. Canada - Manitoba; Nunavut; Ontario. United States (USA) -
Alabama; Arkansas; Florida; Georgia; Indiana; Illinois; Kentucky;
Louisiana; Michigan; Minnesota; Mississippi; Missouri; North Carolina;
Ohio; Tennessee; Wisconsin."],
        BBOX[23.97,-90,84,-84]],
    ID["EPSG",26916]]
Data axis to CRS axis mapping: 1,2
Origin = (427820.000000000000000,3395510.000000000000000)
Pixel Size = (1.000000000000000,-1.000000000000000)
Metadata:
  AREA_OR_POINT=Area
Image Structure Metadata:
  COMPRESSION=DEFLATE
  INTERLEAVE=PIXEL
  LAYOUT=COG
  PREDICTOR=2
Corner Coordinates:
Upper Left  (  427820.000, 3395510.000) ( 87d45'13.12"W, 30d41'24.67"N)
Lower Left  (  427820.000, 3387945.000) ( 87d45'11.21"W, 30d37'18.94"N)
Upper Right (  434454.000, 3395510.000) ( 87d41' 3.77"W, 30d41'26.05"N)
Lower Right (  434454.000, 3387945.000) ( 87d41' 2.04"W, 30d37'20.32"N)
Center      (  431137.000, 3391727.500) ( 87d43' 7.54"W, 30d39'22.51"N)
Band 1 Block=512x512 Type=Byte, ColorInterp=Red
  Overviews: 3317x3782, 1658x1891, 829x945, 414x472
Band 2 Block=512x512 Type=Byte, ColorInterp=Green
  Overviews: 3317x3782, 1658x1891, 829x945, 414x472
Band 3 Block=512x512 Type=Byte, ColorInterp=Blue
  Overviews: 3317x3782, 1658x1891, 829x945, 414x472
Band 4 Block=512x512 Type=Byte, ColorInterp=Undefined
  Overviews: 3317x3782, 1658x1891, 829x945, 414x472
```

- Convert GeoTiff to PNG by using GDAL:

```
Python
```

```
from osgeo import gdal
gdal.UseExceptions()
access_key = TokenLibrary.getSecret('<key-vault-name>','<secret-name>')
gdal.SetConfigOption('AZURE_STORAGE_ACCOUNT', '<storage_account_name>')
gdal.SetConfigOption('AZURE_STORAGE_ACCESS_KEY', access_key)
tiff_in = "/vsiadls/aoa/input/sample_image.tiff"
#/vsiadls/<container_name>/path/to/image
png_out = "/vsiadls/aoa/input/sample_image.png"
#/vsiadls/<container_name>/path/to/image
options = gdal.TranslateOptions(format='PNG')
gdal.Translate(png_out, tiff_in, options=options)
```

- Store GeoTiff images in Azure Data Lake Storage.

  Because of how data is stored in the cloud and the fact that the file handlers `/vsiaz/` and `/vsiadls/` support only sequential writes, we use the file mount feature available in the mssparkutils package. After the output is written to a mount location, copy it to Azure Data Lake Storage as shown in this sample transformation:

  Python

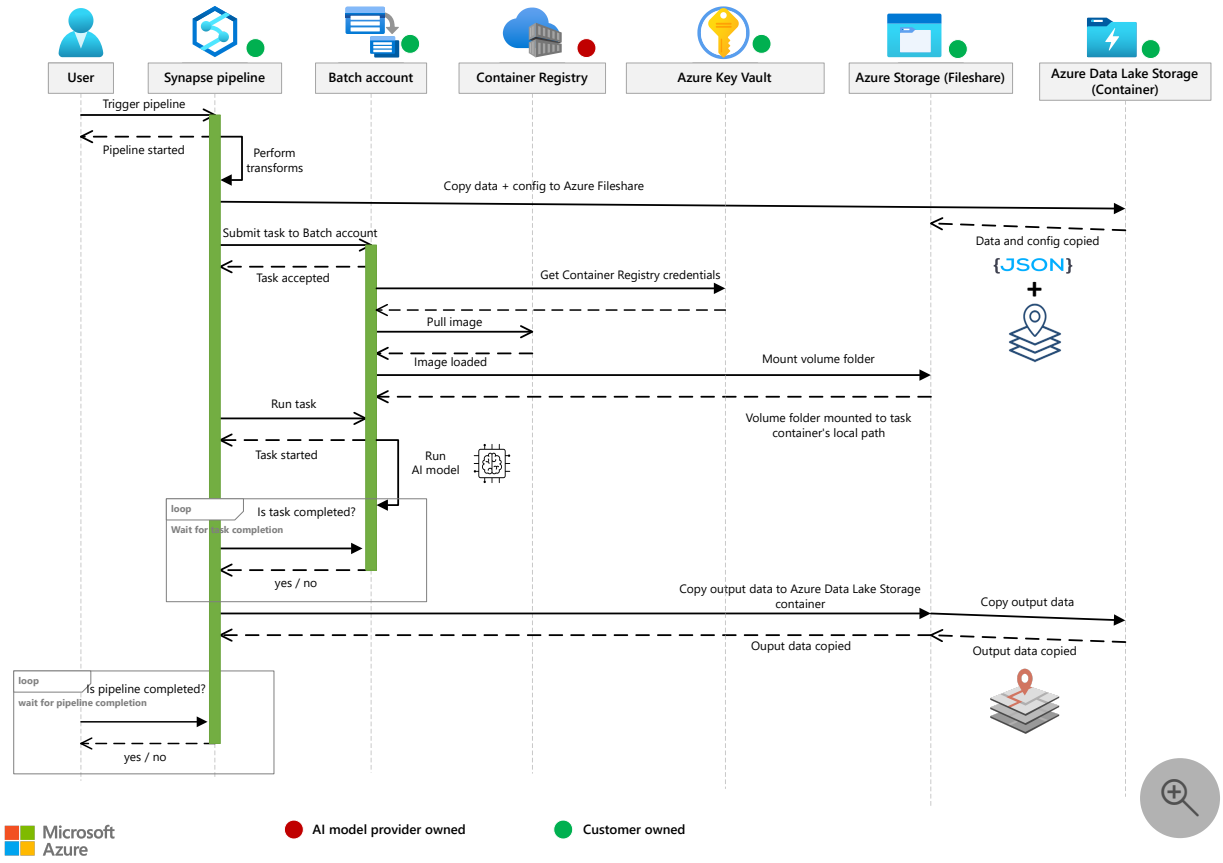
```
import shutil
import sys
from osgeo import gdal
from notebookutils import mssparkutils

mssparkutils.fs.mount(

"abfss://<container_name>@<storage_account_name>.dfs.core.windows.net",
    "/<mount_path>",
    {"linkedService":"<linked_service_name>"}
)

access_key = TokenLibrary.getSecret('<key-vault-name>','<secret-name>')
gdal.SetConfigOption('AZURE_STORAGE_ACCOUNT', '<storage_account_name>')
gdal.SetConfigOption('AZURE_STORAGE_ACCESS_KEY', access_key)

options = gdal.WarpOptions(options=['tr'], xRes=1000, yRes=1000)
gdal.Warp('dst_img.tiff',
'/vsiadls/<container_name>/path/to/src_img.tiff', options=options)

jobId = mssparkutils.env.getJobId()

shutil.copy("dst_img.tiff",
f"/synfs/{jobId}/<mount_path>/path/to/dst_img.tiff")
```

  In Azure Synapse, you can add Azure Data Lake Storage as one of the linked services. For instructions, see Linked services.

Sample solution

An implementation of this architecture is available on GitHub.

This diagram shows the steps in the sample solution:



Download a Visio file of this architecture.

> **Note**
>
> The data is pulled from spaceborne data sources and copied to Azure Data Lake Storage. The data ingestion isn't part of the reference implementation.

1. An Azure Synapse pipeline reads the spaceborne data from Azure Data Lake Storage.
2. The data is processed with the GDAL library in an Azure Synapse notebook.
3. The processed data is stored in Azure Data Lake Storage.
4. The processed data is read from Azure Data Lake Storage and passed to object detection Custom Vision AI models by an Azure Synapse Custom activity. The Custom activity uses Azure Batch pools to run the object detection model.
5. The object detection model outputs a list of detected objects and bounding boxes.

6. The detected objects are converted to GeoJSON and stored in Azure Data Lake Storage.
7. The GeoJSON data is read from Azure Data Lake Storage and stored in a PostgreSQL database.
8. The data is read from the PostgreSQL database. It can be visualized further in tools like ArcGIS Pro, QGIS, and Power BI.

Install geospatial packages in an Azure Synapse Spark pool

You need to install the packages in an Azure Synapse Spark pool by using the package management feature. For more information, see Azure Synapse package management.

To support spaceborne data workloads on Azure Synapse, you need libraries like GDAL, Rasterio, and GeoPandas. You can install these libraries on a serverless Apache Spark pool by using a YAML file. Anaconda libraries are pre-installed on the Spark pool.

Prerequisites

- Create an Azure Synapse workspace.
- Create the Spark pool in Azure Synapse Studio.

Instructions

1. The following libraries and packages are available in the environment.yml file. We recommend using this file to install the libraries in the Spark pools. If you copy the below content, make sure there are no tabs, as YAML only allows spaces as indentation.

YAML

```yaml
name: aoi-env
channels:
  - conda-forge
  - defaults
dependencies:
  - azure-storage-file-datalake
  - gdal=3.3.0
  - libgdal
  - pip>=20.1.1
  - pyproj
  - shapely
  - pip:
```

```
    - rasterio
    - geopandas
```

> **Note**
>
> GDAL uses virtual file system **/vsiadls/** for Azure Data Lake Storage. This functionality is available starting in **GDAL v3.3.0**. Be sure to use version 3.3.0 or later.

2. Go to https://web.azuresynapse.net and sign in to your workspace.

3. Select **Manage** in the navigation pane and then select **Apache Spark pools**.

4. Select **Packages** by selecting the ellipsis button (...) on the Spark pool. Upload the environment.yml file from local and apply the package settings.

5. The notification section of the portal notifies you when the installation is complete. You can also track installation progress by taking these steps:
   a. Go to the Spark applications list on the **Monitor** tab.
   b. Select the **SystemReservedJob-LibraryManagement** link that corresponds to your pool update.
   c. View the driver logs.

6. Run the following code to verify that the correct versions of the libraries installed. The pre-installed libraries that Conda installs will also be listed.

   Python

   ```python
   import pkg_resources
   for d in pkg_resources.working_set:
      print(d)
   ```

For more information, see Manage packages.

Contributors

This article is being updated and maintained by Microsoft. It was originally written by the following contributors.

Principal authors:

- Kungumaraj Nachimuthu | Senior Software Engineer
- Karthick Narendran | Senior Software Engineer

Additional contributors:

- Mick Alberts | Technical Writer
- Taylor Corbett | Senior Data Scientist
- Tushar Dhadiwal | Senior Software Engineer
- Mandar Inamdar | Principal Engineering Manager
- Sushil Kumar | Senior Software Engineer
- Nikhil Manchanda | Principal Engineering Manager
- Safiyah Sadiq | Software Engineer II
- Xiaoyuan Yang | Principal Data Science Manager
- Tai Yee | Senior Program Manager

Next steps

- Getting geospatial insights from big data using SynapseML
- Get started with Azure Synapse Analytics
- Explore Azure Synapse Studio
- Create and consume Cognitive Services

Related resources

- Geospatial data processing and analytics
- Geospatial analysis for the telecommunications industry
- Big data architectures
- End-to-end computer vision at the edge for manufacturing

Vision classifier model with Azure Custom Vision Cognitive Service

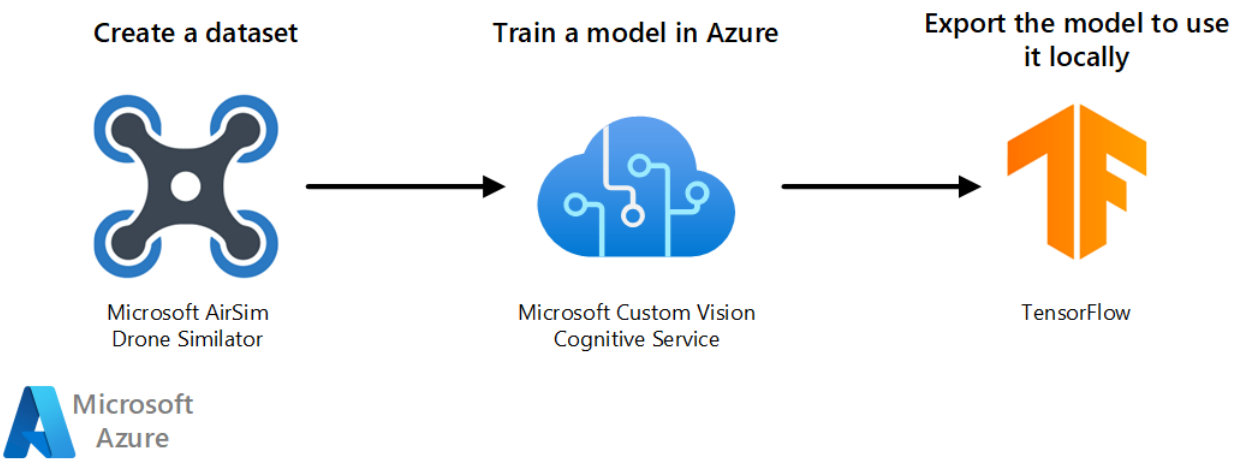
Azure GitHub

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 architecture uses Custom Vision to classify images taken by a simulated drone. It provides a way to combine AI and the Internet of Things (IoT). Azure Custom Vision can also be used for object detection purpose.

Architecture



Download a Visio file of this architecture.

Workflow

1. Use AirSim's 3D-rendered environment to take images taken with the drone. Use the images as the training dataset.
2. Import and tag the dataset in a Custom Vision project. The cognitive service trains and tests the model.
3. Export the model into TensorFlow format so you can use it locally.
4. The model can also be deployed to a container or to mobile devices.

Components

Microsoft AirSim Drone simulator

Microsoft AirSim Drone simulator is built on the Unreal Engine. The simulator is open-source, cross-platform, and developed to help AI research. In this architecture, it creates the dataset of images used to train the model.

Azure Custom Vision

Azure Custom Vision is part of Azure Cognitive Services. In this architecture, it creates an image classifier model.

TensorFlow

TensorFlow is an open-source platform for machine learning (ML). It's a tool that helps you develop and train ML models. When you export your model to TensorFlow format, you'll have a protocol buffer file with the Custom Vision model that you can use locally in your script.

Scenario details

Azure Cognitive Services offers many possibilities for Artificial Intelligence (AI) solutions. One of them is Azure Custom Vision, which allows you to build, deploy, and improve your image classifiers. This architecture uses Custom Vision to classify images taken by a simulated drone. It provides a way to combine AI and the Internet of Things (IoT). Azure Custom Vision can also be used for object detection purpose.

Potential use case

This solution is ideal for the rescue, simulation, robotics, aircraft, aerospace, and aviation industries.

Microsoft Search and Rescue Lab suggests a hypothetical use case for Custom Vision. In the lab, you fly a Microsoft AirSim simulated drone around in a 3D-rendered environment. You use the simulated drone to capture synthetic images of the animals in that environment. After creating a dataset of images, you use the dataset to train a Custom Vision classifier model. To train the model, you tag the images with the names of the animals. When you fly the drone again, take new images of the animals. This solution identifies the name of the animal in each new image.

In a practical application of the lab, an actual drone replaces the Microsoft AirSim simulated drone. If a pet is lost, the owner provides images of the pet to the Custom Vision model trainer. Just like in the simulation, the images are used to train the model to recognize the pet. Then, the drone pilot searches an area where the lost pet might be. As it finds animals along the way, the drone's camera can capture images and determine if the animal is the lost pet.

Deploy this scenario

To deploy this reference architecture, follow the steps described in the GitHub repo of the Search and Rescue Lab.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal authors:

- Jose Contreras | Principal Software Engineer

Next steps

- Learn more about Microsoft AirSim
- Learn more about Azure Custom Vision Cognitive Service
- Learn more about Azure Cognitive Services

Related resources

Read other Azure Architecture Center articles:

- Image classification on Azure
- Geospatial analysis with Azure Synapse Analytics
- AI enrichment with image and natural language processing in Azure Cognitive Search

Overview of Azure FarmBeats (Preview)

Article • 11/29/2023

Azure FarmBeats is a business-to-business offering available in Azure Marketplace. It enables aggregation of agriculture data sets across providers. Azure FarmBeats enables you to build artificial intelligence (AI) or machine learning (ML) models based on fused data sets. By using Azure FarmBeats, agriculture businesses can focus on core value-adds instead of the undifferentiated heavy lifting of data engineering.

> **Important**
>
> Azure FarmBeats is retired. You can see the public announcement **here**.
>
> We have built a new agriculture focused service, it's name is Azure Data Manager for Agriculture and it's now available as a preview service. For more information see public documentation **here** or write to us at madma@microsoft.com.

With the preview of Azure FarmBeats you can:

- Assess farm health using vegetation index and water index based on satellite imagery.
- Get recommendations on how many soil moisture sensors to use and where to place them.
- Track farm conditions by visualizing ground data collected by sensors from various vendors.
- Get soil moisture map based on the fusion of satellite and sensor data.

- Gain actionable insights by building AI/ML models on top of aggregated datasets.
- Build or augment your digital agriculture solution by providing farm health advisories.

Data hub

The Azure FarmBeats Data hub is an API layer, which enables aggregation, normalization, and contextualization of various agriculture datasets across providers. You can use Azure FarmBeats to get:

- **Sensor data** from two sensor providers Davis Instruments, Teralytic, Pessl Instruments
- **Satellite imagery** from European Space Agency's Sentinel-2 satellite mission
- **Drone imagery** from three drone imagery providers senseFly, SlantRange, DJI

Data hub is designed as an extensible API platform. We're working with many more providers to integrate with Azure FarmBeats, so you have more choice while building your solution.

Accelerator

The Azure FarmBeats Accelerator is a sample web application, that is built on top of Data hub. The Accelerator jump-starts your user interface and model development. The Azure FarmBeats accelerator uses Azure FarmBeats' APIs. It visualizes ingested sensor data as charts and model outputs as maps. For example, you can use the accelerator to create a farm quickly and get a vegetation index map or a sensor placement map for that farm easily.

Azure role-based access control (Azure RBAC)

An administrator can define access rules for Azure FarmBeats using one of the predefined roles. Roles determine what areas of the application a user has access to and what actions they can perform. There are two kinds of roles in Azure FarmBeats - for users and for partners.

User Roles

An administrator can add and manage users and define their access levels based on two user roles: Admin and Read-Only.

Partner Roles

An administrator can add multiple partners as data providers to Azure FarmBeats. The following summarize the available partner roles in FarmBeats and their permissions:

Expand table

Partner Type	Actions	Scope
Sensor Partner	Create, Read, Update	DeviceModel, Device, SensorModel, Sensor
	Read, Update	ExtendedType
Imagery Partner	Create, Read, Update	Scene, SceneFile
	Read, Update	ExtendedType
	Read	Farm
Weather Partner*	Create, Read, Update	WeatherDataModel, WeatherDataLocation, JobType
(* Coming soon)	Read, Update	ExtendedType
	Read	Farm

Resources

Azure FarmBeats is offered at no extra charge and you pay only for the Azure resources you use. You can use the below resources to know more about the offering:

- Stay informed about the latest Azure FarmBeats news by visiting our Azure FarmBeats blog.
- Seek help by posting a question on our Azure FarmBeats support forum.
- Provide feedback by posting or voting for a feature idea on our Azure FarmBeats feedback forum.

Next steps

Install Azure FarmBeats

Generate Soil Moisture Heatmap

Article • 11/29/2023

Soil moisture is the water that is held in the spaces between soil particles. Soil Moisture Heatmap helps you understand the moisture data at any depth, and at high resolution within your farms. To generate an accurate and usable soil moisture heatmap, a uniform deployment of sensors from the same provider is required. Different providers will have differences in the way soil moisture is measured along with differences in calibration. The Heatmap is generated for a particular depth using the sensors deployed at that depth.

> **Important**
>
> Azure FarmBeats is retired. You can see the public announcement **here**.
>
> We have built a new agriculture focused service, it's name is Azure Data Manager for Agriculture and it's now available as a preview service. For more information see public documentation **here** or write to us at madma@microsoft.com.

This article describes the process of generating a Soil Moisture Heatmap for your farm, using the Azure FarmBeats Accelerator. In this article, you will learn how to:

- Create Farms
- Assign sensors to Farms
- Generate Soil Moisture Heatmap

Before you begin

Ensure the following:

- An Azure subscription.
- A running instance of Azure FarmBeats.
- A minimum three soil moisture sensors are available for the farm.

Create a farm

A farm is a geographical area of interest for which you want to create a soil moisture heatmap. You can create a farm using the Farms API or in the FarmsBeats Accelerator UI

Deploy sensors

You should physically deploy soil moisture sensors on the farm. You can purchase soil moisture sensors from any of our approved partners - Davis Instruments and Teralytic. You should coordinate with your sensor provider to do the physical setup on your farm.

Get soil moisture sensor data from partner

As sensors start streaming, the data into the partner data dashboard, they enable the data into Azure FarmBeats. This can be done from the partner application.

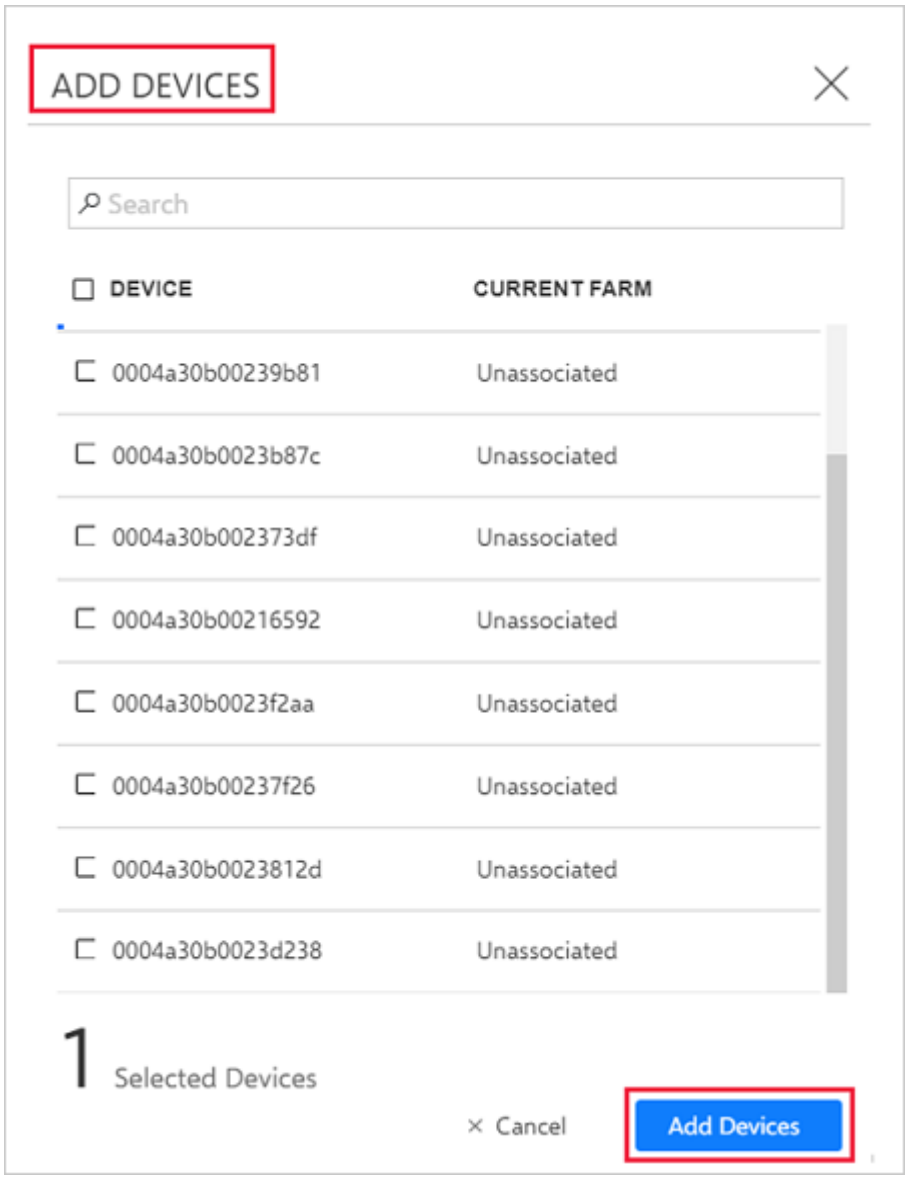
For example, if you have purchased Davis sensors, you will log into your weather link account, and provide the required credentials to enable the data streaming into Azure FarmBeats. To get the required credentials, follow the instructions from Get sensor data.

Once you enter your credentials and select **Submit** on the partner application, you can have the data flowing into Azure FarmBeats.

Assign soil moisture sensors to the farm

Once you have linked your sensor account into Azure FarmBeats, you need to assign the soil moisture sensors to the farm of interest.

1. In the home page, select **Farms** from the menu, the **Farms** list page is displayed.
2. Select **MyFarm** > **Add Devices**.
3. The **Add Devices** window displays. Select any device that is linked to the soil moisture sensors for your farm.



4. Select **Add Devices**.

Generate Soil Moisture Heatmap

This step is to create a job or a long running operation that will generate Soil Moisture Heatmap for your farm.

1. On the home page, go to **Farms** from the left navigation menu to view the farms page.

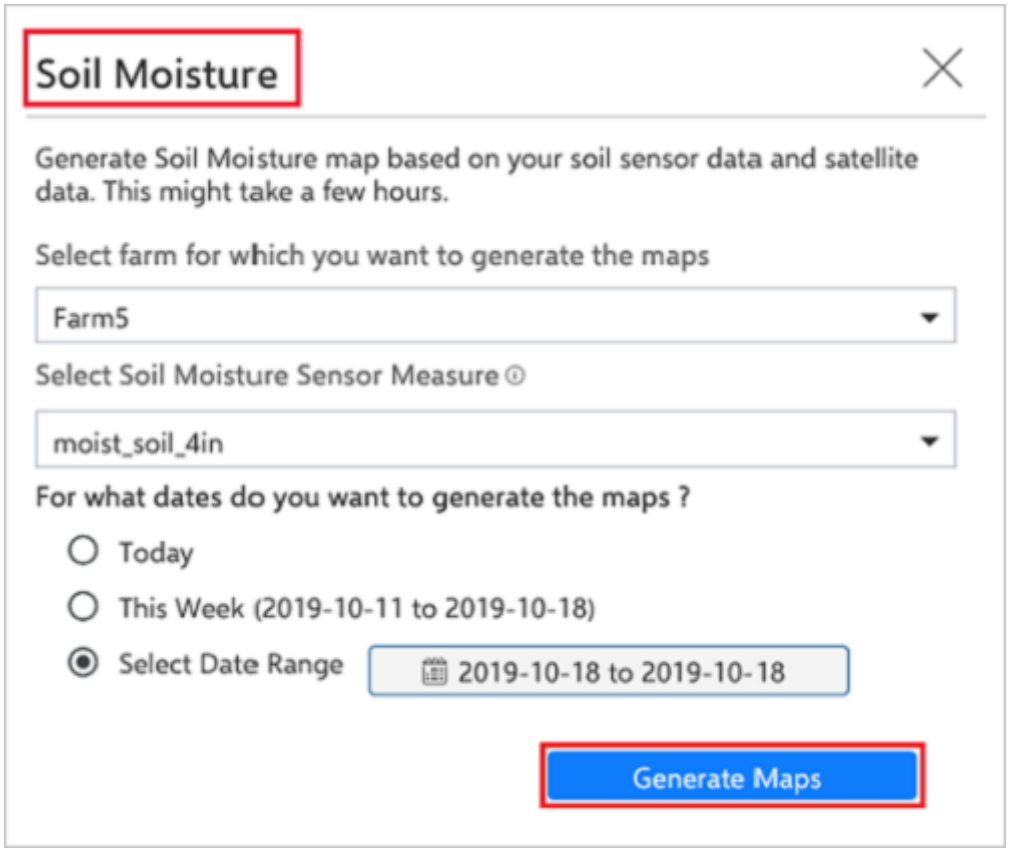
2. Select **MyFarm**.

3. In the **Farm Details** page, select **Generate Precision Map**.

4. From the drop-down menu, select **Soil Moisture**.

5. In the **Soil Moisture** window, select **This Week**.

6. In the **Select Soil Moisture Sensor Measure**, enter the measure you want to use for the map. To find the sensor measure, in **Sensors**, select any soil moisture sensor. In **Sensor Properties**, use the **Measure Name** value.



7. Select **Generate Maps**. A confirmation message with job details is displayed. For more information, see Job Status in Jobs.

> ⓘ **Note**
>
> The job takes around three to four hours to complete.

Download the Soil Moisture Heatmap

Use the following steps:

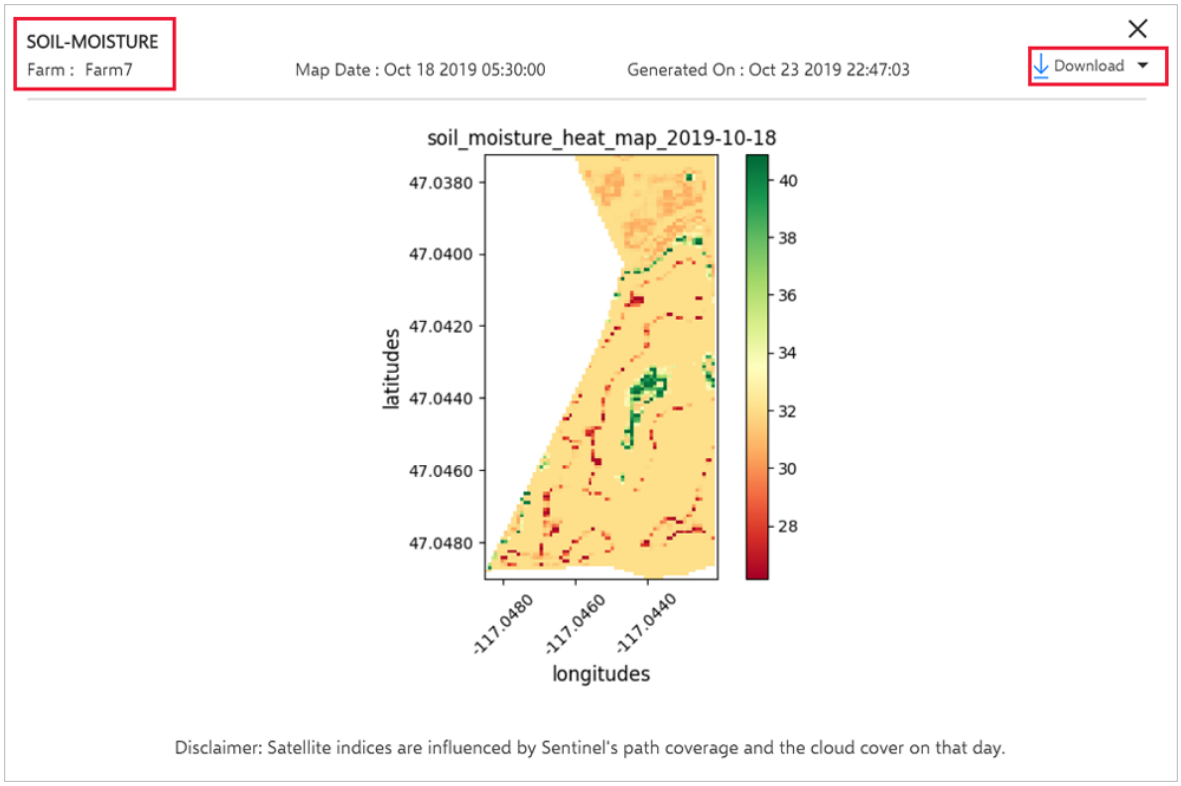
1. In the **Jobs** page, check the **Job Status** for the job you created in the last procedure.

2. When the job status shows **Succeeded**, select **Maps** on the menu.

3. Search for the map by the day it was created in the format <soil-moisture_MyFarm_YYYY-MM-DD>.

4. Select a map in the **Name** column, a pop-up window displays with the preview of the selected map.

5. Select **Download**. The map is downloaded and stored to the local folder of your computer.



Next steps

Now that you've successfully generated a Soil Moisture Heatmap, learn how to generate sensor placement and ingest historical telemetry data.

Sensor partner integration

Article • 11/29/2023

This article provides information about the Azure FarmBeats **Translator** component, which enables sensor partner integration.

> **Important**
>
> Azure FarmBeats is retired. You can see the public announcement **here**.
>
> We have built a new agriculture focused service, it's name is Azure Data Manager for Agriculture and it's now available as a preview service. For more information see public documentation **here** or write to us at madma@microsoft.com.

Using this component, partners can integrate with FarmBeats using FarmBeats Datahub APIs and send customer device data and telemetry to FarmBeats Datahub. Once the data is available in FarmBeats, it is visualized using the FarmBeats Accelerator and can be used for data fusion and for building machine learning/artificial intelligence models.

Before you start

To develop the Translator component, you will need the following credentials that will enable access to the FarmBeats APIs.

- API Endpoint
- Tenant ID
- Client ID
- Client Secret
- EventHub Connection String

See this section for getting the above credentials: Enable Device Integration

Translator development

REST API-based integration

Sensor data integration capabilities of FarmBeats are exposed via the REST API. Capabilities include metadata definition, device and sensor provisioning, and device and sensor management.

Telemetry ingestion

The telemetry data is mapped to a canonical message that's published on Azure Event Hubs for processing. Azure Event Hubs is a service that enables real-time data (telemetry) ingestion from connected devices and applications.

API development

The APIs contain Swagger technical documentation.

Authentication

FarmBeats uses Microsoft Entra authentication. Azure App Service provides built-in authentication and authorization support.

For more information, see Microsoft Entra ID.

FarmBeats Datahub uses bearer authentication, which needs the following credentials:

- Client ID
- Client secret
- Tenant ID

Using these credentials, the caller can request an access token. The token needs to be sent in the subsequent API requests, in the header section, as follows:

```
headers = {"Authorization": "Bearer " + access_token, …}
```

The following sample Python code gives the access token, which can be used for subsequent API calls to FarmBeats.

```python
import requests
import json
import msal

# Your service principal App ID
CLIENT_ID = "<CLIENT_ID>"
# Your service principal password
CLIENT_SECRET = "<CLIENT_SECRET>"
# Tenant ID for your Azure subscription
TENANT_ID = "<TENANT_ID>"

AUTHORITY_HOST = 'https://login.microsoftonline.com'
AUTHORITY = AUTHORITY_HOST + '/' + TENANT_ID

ENDPOINT = "https://<yourfarmbeatswebsitename-api>.azurewebsites.net"
```

```
SCOPE = ENDPOINT + "/.default"

context = msal.ConfidentialClientApplication(CLIENT_ID, authority=AUTHORITY,
client_credential=CLIENT_SECRET)
token_response = context.acquire_token_for_client(SCOPE)
# We should get an access token here
access_token = token_response.get('access_token')
```

HTTP request headers

Here are the most common request headers that need to be specified when you make an API call to FarmBeats Datahub.

Expand table

Header	Description and example
Content-Type	The request format (Content-Type: `application/<format>`). For FarmBeats Datahub APIs, the format is JSON. Content-Type: application/json
Authorization	Specifies the access token required to make an API call. Authorization: Bearer <Access-Token>
Accept	The response format. For FarmBeats Datahub APIs, the format is JSON. Accept: application/json

API requests

To make a REST API request, you combine the HTTP (GET, POST, or PUT) method, the URL to the API service, the Uniform Resource Identifier (URI) to a resource to query, submit data to, update, or delete, and one or more HTTP request headers. The URL to the API service is the API endpoint you provide. Here's a sample: `https://\<yourdatahub-website-name>.azurewebsites.net`

Optionally, you can include query parameters on GET calls to filter, limit the size of, and sort the data in the responses.

The following sample request is to get the list of devices.

Bash

```
curl -X GET "https://microsoft-farmbeats.azurewebsites.net/Device" -H
"Content-Type: application/json" -H "Authorization: Bearer <Access-Token>"
```

Most GET, POST, and PUT calls require a JSON request body.

The following sample request is to create a device. (This sample has an input JSON with the request body.)

Bash

```
curl -X POST "https://microsoft-farmbeats.azurewebsites.net/Device" -H
"accept: application/json" -H  "Content-Type: application/json" -H
"Authorization: Bearer <Access-Token>" -d "{  \"deviceModelId\": \"ID123\",
\"hardwareId\": \"MHDN123\",  \"reportingInterval\": 900,  \"name\":
\"Device123\",  \"description\": \"Test Device 123\",}"
```

Data format

JSON is a common language-independent data format that provides a simple text representation of arbitrary data structures. For more information, see json.org.

Metadata specifications

FarmBeats Datahub has the following APIs that enable device partners to create and manage device or sensor metadata.

- **/DeviceModel**: DeviceModel corresponds to the metadata of the device, such as the manufacturer and the type of device, which is either gateway or node.
- **/Device**: Device corresponds to a physical device present on the farm.
- **/SensorModel**: SensorModel corresponds to the metadata of the sensor, such as the manufacturer, the type of sensor, which is either analog or digital, and the sensor measure, such as ambient temperature and pressure.
- **/Sensor**: Sensor corresponds to a physical sensor that records values. A sensor is typically connected to a device with a device ID.

Expand table

DeviceModel	Description
Type (node, gateway)	Type of the device - Node or Gateway
Manufacturer	Name of the manufacturer
ProductCode	Device product code or model name or number. For example, EnviroMonitor#6800.

DeviceModel	Description
Ports	Port name and type, which is digital or analog.
Name	Name to identify resource. For example, model name or product name.
Description	Provide a meaningful description of the model.
Properties	Additional properties from the manufacturer.
Device	**Description**
DeviceModelId	ID of the associated device model.
HardwareId	Unique ID for the device, such as a MAC address.
ReportingInterval	Reporting interval in seconds.
Location	Device latitude (-90 to +90), longitude (-180 to 180), and elevation (in meters).
ParentDeviceId	ID of the parent device to which this device is connected. For example, if a node is connected to a gateway, the node has parentDeviceID as the gateway.
Name	Name to identify the resource. Device partners need to send a name that's consistent with the device name on the device partner side. If the device name is user-defined on the device partner side, the same user-defined name should be propagated to FarmBeats.
Description	Provide a meaningful description.
Properties	Additional properties from the manufacturer.
SensorModel	**Description**
Type (analog, digital)	Mention analog or digital sensor.
Manufacturer	Name of manufacturer.
ProductCode	Product code or model name or number. For example, RS-CO2-N01.
SensorMeasures > Name	Name of the sensor measure. Only lowercase is supported. For measurements from different depths, specify the depth. For example, soil_moisture_15cm. This name has to be consistent with the telemetry data.
SensorMeasures > DataType	Telemetry data type. Currently, double is supported.
SensorMeasures > Type	Measurement type of the sensor telemetry data. Following are the system-defined types: AmbientTemperature, CO2, Depth,

DeviceModel	Description
	ElectricalConductivity, LeafWetness, Length, LiquidLevel, Nitrate, O2, PH, Phosphate, PointInTime, Potassium, Pressure, RainGauge, RelativeHumidity, Salinity, SoilMoisture, SoilTemperature, SolarRadiation, State, TimeDuration, UVRadiation, UVIndex, Volume, WindDirection, WindRun, WindSpeed, Evapotranspiration, PAR. To add more, refer to the /ExtendedType API.
SensorMeasures > Unit	Unit of sensor telemetry data. Following are the system-defined units: NoUnit, Celsius, Fahrenheit, Kelvin, Rankine, Pascal, Mercury, PSI, MilliMeter, CentiMeter, Meter, Inch, Feet, Mile, KiloMeter, MilesPerHour, MilesPerSecond, KMPerHour, KMPerSecond, MetersPerHour, MetersPerSecond, Degree, WattsPerSquareMeter, KiloWattsPerSquareMeter, MilliWattsPerSquareCentiMeter, MilliJoulesPerSquareCentiMeter, VolumetricWaterContent, Percentage, PartsPerMillion, MicroMol, MicroMolesPerLiter, SiemensPerSquareMeterPerMole, MilliSiemensPerCentiMeter, Centibar, DeciSiemensPerMeter, KiloPascal, VolumetricIonContent, Liter, MilliLiter, Seconds, UnixTimestamp, MicroMolPerMeterSquaredPerSecond, and InchesPerHour. To add more, refer to the /ExtendedType API.
SensorMeasures > AggregationType	Either none, average, maximum, minimum, or StandardDeviation.
SensorMeasures > Depth	The depth of the sensor in centimeters. For example, the measurement of moisture 10 cm under the ground.
SensorMeasures > Description	Provide a meaningful description of the measurement.
Name	Name to identify resource. For example, the model name or product name.
Description	Provide a meaningful description of the model.
Properties	Additional properties from the manufacturer.
Sensor	**Description**
HardwareId	Unique ID for the sensor set by the manufacturer.
SensorModelId	ID of the associated sensor model.
Location	Sensor latitude (-90 to +90), longitude (-180 to 180), and elevation (in meters).
Port > Name	Name and type of the port that the sensor is connected to on the device. This must be the same name as defined in the device model.
DeviceId	ID of the device that the sensor is connected to.

DeviceModel	Description
Name	Name to identify the resource. For example, the sensor name or product name and model number or product code.
Description	Provide a meaningful description.
Properties	Additional properties from the manufacturer.

> **Note**
>
> The APIs return unique IDs for each instance created. This ID needs to be retained by the Translator for device management and metadata sync.

Metadata sync

The Translator should send updates on the metadata. For example, update scenarios are change of device or sensor name and change of device or sensor location.

The Translator should have the ability to add new devices or sensors that were installed by the user post linking of FarmBeats. Similarly, if a device or sensor was updated by the user, the same should be updated in FarmBeats for the corresponding device or sensor. Typical scenarios that require updating a device or sensor are a change in a device location or the addition of sensors in a node.

> **Note**
>
> Delete isn't supported for device or sensor metadata.
>
> To update metadata, it's mandatory to call /Get/{id} on the device or sensor, update the changed properties, and then do a /Put/{id} so that any properties set by the user aren't lost.

Add new types and units

FarmBeats supports adding new sensor measure types and units.

Telemetry specifications

The telemetry data is mapped to a canonical message that's published on Azure Event Hubs for processing. Azure Event Hubs is a service that enables real-time data (telemetry) ingestion from connected devices and applications.

Send telemetry data to FarmBeats

To send telemetry data to FarmBeats, create a client that sends messages to an event hub in FarmBeats. For more information about telemetry data, see Sending telemetry to an event hub.

Here's a sample Python code that sends telemetry as a client to a specified event hub.

Python

```python
import azure
from azure.eventhub import EventHubClient, Sender, EventData, Receiver,
Offset
EVENTHUBCONNECTIONSTRING = "<EventHub Connection String provided by
customer>"
EVENTHUBNAME = "<EventHub Name provided by customer>"

write_client =
EventHubClient.from_connection_string(EVENTHUBCONNECTIONSTRING,
eventhub=EVENTHUBNAME, debug=False)
sender = write_client.add_sender(partition="0")
write_client.run()
for i in range(5):
    telemetry = "<Canonical Telemetry message>"
    print("Sending telemetry: " + telemetry)
    sender.send(EventData(telemetry))
write_client.stop()
```

The canonical message format is as follows:

JSON

```json
{
"deviceid": "<id of the Device created>",
"timestamp": "<timestamp in ISO 8601 format>",
"version" : "1",
"sensors": [
    {
      "id": "<id of the sensor created>",
      "sensordata": [
        {
          "timestamp": "< timestamp in ISO 8601 format >",
          "<sensor measure name (as defined in the Sensor Model)>": <value>
        },
        {
          "timestamp": "<timestamp in ISO 8601 format>",
          "<sensor measure name (as defined in the Sensor Model)>": <value>
        }
      ]
    }
```

```
  ]
}
```

All key names in the telemetry JSON should be lowercase. Examples are deviceid and sensordata.

For example, here's a telemetry message:

JSON

```
{
  "deviceid": "7f9b4b92-ba45-4a1d-a6ae-c6eda3a5bd12",
  "timestamp": "2019-06-22T06:55:02.7279559Z",
  "version" : "1",
  "sensors": [
    {
      "id": "d8e7beb4-72de-4eed-9e00-45e09043a0b3",
      "sensordata": [
        {
          "timestamp": "2019-06-22T06:55:02.7279559Z",
          "hum_max": 15
        },
        {
          "timestamp": "2019-06-22T06:55:02.7279559Z",
          "hum_min": 42
        }
      ]
    },
    {
      "id": "d8e7beb4-72de-4eed-9e00-45e09043a0b3",
      "sensordata": [
        {
          "timestamp": "2019-06-22T06:55:02.7279559Z",
          "hum_max": 20
        },
        {
          "timestamp": "2019-06-22T06:55:02.7279559Z",
          "hum_min": 89
        }
      ]
    }
  ]
}
```

Note

The following sections are related to other changes (eg. UI, error management etc.) that the sensor partner can refer to in developing the Translator component.

Link a FarmBeats account

After customers have purchased and deployed devices or sensors, they can access the device data and telemetry on the device partners' software as a service (SaaS) portal. Device partners can enable customers to link their account to their FarmBeats instance on Azure by providing a way to input the following credentials:

- Display name (an optional field for users to define a name for this integration)
- API endpoint
- Tenant ID
- Client ID
- Client secret
- EventHub connection string
- Start date

> **Note**
>
> The start date enables the historical data feed, that is, the data from the date specified by the user.

Unlink FarmBeats

Device partners can enable customers to unlink an existing FarmBeats integration. Unlinking FarmBeats shouldn't delete any device or sensor metadata that was created in FarmBeats Datahub. Unlinking does the following:

- Stops telemetry flow.
- Deletes and erases the integration credentials on the device partner.

Edit FarmBeats integration

Device partners can enable customers to edit the FarmBeats integration settings if the client secret or connection string changes. In this case, only the following fields are editable:

- Display name (if applicable)
- Client secret (should be displayed in "2x8**********" format or the Show/Hide feature rather than clear text)
- Connection string (should be displayed in "2x8**********" format or Show/Hide feature rather than clear text)

View the last telemetry sent

Device partners can enable customers to view the timestamp of the last telemetry that was sent, which is found under **Telemetry Sent**. This is the time at which the latest telemetry was successfully sent to FarmBeats.

Troubleshooting and error management

Troubleshoot option or support

If customer is unable to receive device data or telemetry in the FarmBeats instance specified, the device partner should provide support and a mechanism for troubleshooting.

Telemetry data retention

The telemetry data should also be retained for a predefined time period so that it can be useful in debugging or resending the telemetry if an error or data loss occurs.

Error management or error notification

If an error affects the device or sensor metadata or the data integration or telemetry data flow in the device partner system, customer should receive a notification. A mechanism to resolve any errors should also be designed and implemented.

Connection checklist

Device manufacturers or partners can use the following checklist to ensure that the credentials provided by the customer are accurate:

- Check to see whether an access token is received with the credentials that were provided.
- Check to see whether an API call succeeds with the access token that was received.
- Check to see whether the EventHub client connection is established.

Next steps

For more information about the REST API, see REST API.

Weather partner integration with FarmBeats

Article • 11/29/2023

This article provides information about the Azure FarmBeats Connector Docker component. As a weather data provider, you can use the Connector Docker to integrate with FarmBeats. Use its APIs to send weather data to FarmBeats. In FarmBeats, the data can be used for data fusion and for building machine learning models or artificial intelligence models.

> **Important**
>
> Azure FarmBeats is retired. You can see the public announcement **here**.
>
> We have built a new agriculture focused service, it's name is Azure Data Manager for Agriculture and it's now available as a preview service. For more information see public documentation **here** or write to us at madma@microsoft.com.

You must provide a suitable Docker image or program and host the docker image in a container registry that customers can access. Provide the following information to your customers:

- Docker image URL
- Docker image tag
- Keys or credentials to access the Docker image
- Customer-specific API keys or credentials to access the data from your system
- VM SKU details (Provide these details if your Docker image has specific VM requirements. Otherwise, customers can choose from supported VM SKUs in Azure.)

Customers use this Docker information to register a weather partner in their FarmBeats instance. For more information about how customers can use the Docker to ingest weather data in FarmBeats, see Get data from weather partners.

Connector Docker development

REST API-based integration

The FarmBeats APIs contain Swagger technical documentation.

If you've already installed FarmBeats, access your FarmBeats Swagger at `https://yourfarmbeatswebsitename-api.azurewebsites.net/swagger`

Notice that *-api* is appended to your FarmBeats website name. The API endpoint is `https://yourfarmbeatswebsitename-api.azurewebsites.net`

Datahub lib

FarmBeats provides a lib that you can use. The lib is currently available as part of the reference implementation. Later, it will be available as an SDK for multiple languages.

Authentication

Authentication with FarmBeats APIs

FarmBeats uses bearer authentication. You can access the APIs by providing an access token in the header section of the request. Here's an example:

```
headers = *{"Authorization": "Bearer " + access_token, …}*
```

You can request the access token from an Azure Functions app that's running on the customer's FarmBeats instance. The Azure Functions URL is provided to the Docker program as an argument. You can get the access token by making a `GET` request on the URL. The response from the URL contains the access token.

Use the helper functions in the Datahub lib to get the access token. For more information, see the GitHub page for the NOAA Docker image.

The access token is valid only for few hours. When it expires, you must request it again.

Authentication with partner-side APIs

To authenticate with the partner-side APIs while the Docker job is running, customers need to provide the credentials during partner registration. Here's an example:

JSON

```
{
 "partnerCredentials": {
   "key1": "value1",
   "key2": "value2"
```

```
    }
}
```

The API service serializes this dict and stores it in a key vault.

Azure Data Factory is used to orchestrate weather jobs. It spins up resources to run the Docker code. Data Factory also provides a mechanism to push data securely to the VM where the Docker job runs. The API credentials are then securely stored in the key vault.

Credentials are read as secure strings from the key vault. They're provided as extended properties in the working directory of the Docker container. Their file path is */mnt/working_dir/activity.json*.

The Docker code can read the credentials from *activity.json* during run time to access partner-side APIs for the customer. In the JSON file, the credentials look like this code example:

JSON

```
{
   "typeProperties" : {
      "extendedProperties" : {
         "partnerCredentials": "" }
   }
}
```

The `partnerCredentials` credential is available in the way that the customer provided it during partner registration.

The FarmBeats lib provides helper functions. Use these functions to read the credentials from the activity properties. For more information, see the GitHub page for the NOAA Docker image.

The file is used only while the Docker code is running. After the code finishes, the file is deleted.

For more information about how Data Factory pipelines and activities work, see Schema and data type mapping.

HTTP request headers

The following table shows the most common request headers that you need to specify when you make an API call to FarmBeats.

Expand table

Header	Description and example
Content-Type	The request format. Example: `Content-Type: application/<format>` For FarmBeats Datahub APIs, the format is JSON. Example: `Content-Type: application/json`
Authorization	The access token that's required to make an API call. Example: `Authorization: Bearer <Access-Token>`
Accept	The response format. For FarmBeats Datahub APIs, the format is JSON. Example: `Accept: application/json`

Data format

JSON is a common language-independent data format that provides a simple text representation of arbitrary data structures. For more information, see JSON.org.

Docker specifications

The Docker program needs two components: the bootstrap and the job. The program can have more than one job.

Bootstrap

The bootstrap component should run when the customer starts the Docker registration on FarmBeats. The following arguments (`arg1` and `arg2`) are passed to the program:

- **FarmBeats API endpoint**: The FarmBeats API endpoint for API requests. This endpoint makes API calls to the FarmBeats deployment.
- **Azure Functions URL**: Your own endpoint. This URL provides your access token for FarmBeats APIs. You can call `GET` on this URL to fetch the access token.

The bootstrap creates the metadata that users need to run your jobs to get weather data. For more information, see the reference implementation.

If you customize the *bootstrap_manifest.json* file, then the reference bootstrap program will create the required metadata for you. The bootstrap program creates the following metadata:

> **Note**

If you update the *bootstrap_manifest.json* file as the **reference implementation** describes, you don't need to create the following metadata. The bootstrap program will use your manifest file to create the necessary metadata.

- **/WeatherDataModel**: The WeatherDataModel metadata represents weather data. It corresponds to data sets that the source provides. For example, a DailyForecastSimpleModel might provide average temperature, humidity, and precipitation information once a day. By contrast, a DailyForecastAdvancedModel might provide much more information at hourly granularity. You can create any number of weather data models.
- **/JobType**: FarmBeats has an extensible job management system. As a weather data provider, you'll have various datasets and APIs (for example, GetDailyForecasts). You can enable these datasets and APIs in FarmBeats by using JobType. After a job type is created, a customer can trigger jobs of that type to get weather data for their location or their farm of interest.

Jobs

The Jobs component is invoked every time a FarmBeats user runs a job of the /JobType that you created as part of the bootstrap process. The Docker run command for the job is defined as part of the /JobType that you created.

The job fetches data from the source and pushes it to FarmBeats. During the bootstrap process, the parameters that are required to get the data should be defined as part of /JobType.

As part of the job, the program must create a /WeatherDataLocation based on the /WeatherDataModel that was created during the bootstrap process. The /WeatherDataLocation corresponds to a location (latitude and longitude coordinates) that the user provided as a parameter for the job.

Object details

Expand table

WeatherDataModel	Description
Name	Name of the weather data model.
Description	A meaningful description of the model.
Properties	Additional properties defined by the data provider.

WeatherDataModel	Description
weatherMeasures > Name	Name of the weather measure. For example, humidity_max.
weatherMeasures > DataType	Either Double or Enum. If Enum, measureEnumDefinition is required.
weatherMeasures > measureEnumDefinition	Required only if DataType is Enum. For example, `{ "NoRain": 0, "Snow": 1, "Drizzle": 2, "Rain": 3 }`
weatherMeasures > Type	Type of weather telemetry data. For example, RelativeHumidity. The system-defined types are AmbientTemperature, NoUnit, CO2, Depth, ElectricalConductivity, LeafWetness, Length, LiquidLevel, Nitrate, O2, PH, Phosphate, PointInTime, Potassium, Pressure, RainGauge, RelativeHumidity, Salinity, SoilMoisture, SoilTemperature, SolarRadiation, State, TimeDuration, UVRadiation, UVIndex, Volume, WindDirection, WindRun, WindSpeed, Evapotranspiration, and PAR. To add more types, see the Add ExtendedType section in this article.
weatherMeasures > Unit	Unit of weather telemetry data. The system-defined units are NoUnit, Celsius, Fahrenheit, Kelvin, Rankine, Pascal, Mercury, PSI, MilliMeter, CentiMeter, Meter, Inch, Feet, Mile, KiloMeter, MilesPerHour, MilesPerSecond, KMPerHour, KMPerSecond, MetersPerHour, MetersPerSecond, Degree, WattsPerSquareMeter, KiloWattsPerSquareMeter, MilliWattsPerSquareCentiMeter, MilliJoulesPerSquareCentiMeter, VolumetricWaterContent, Percentage, PartsPerMillion, MicroMole, MicroMolesPerLiter, SiemensPerSquareMeterPerMole, MilliSiemensPerCentiMeter, Centibar, DeciSiemensPerMeter, KiloPascal, VolumetricIonContent, Liter, MilliLiter, Seconds, UnixTimestamp, MicroMolePerMeterSquaredPerSecond, and InchesPerHour. To add more units, see the Add ExtendedType section in this article.
weatherMeasures > AggregationType	The type of aggregation. Possible values are None, Average, Maximum, Minimum, StandardDeviation, Sum, and Total.
weatherMeasures > Depth	The depth of the sensor in centimeters. For example, the measurement of moisture 10 cm under the ground.
weatherMeasures > Description	A meaningful description of the measurement.

Expand table

JobType	Description
Name	Name of the job. For example, Get_Daily_Forecast. The customer will run this job to get weather data.

JobType	Description
pipelineDetails > parameters > name	Name of the parameter.
pipelineDetails > parameters > type	The parameter type. Possible values include String, Int, Float, Bool, and Array.
pipelineDetails > parameters > isRequired	The parameter's Boolean value. The value is true if the parameter is required. Otherwise, the value is false. The default is true.
pipelineDetails > parameters > defaultValue	Default value of the parameter.
pipelineDetails > parameters > description	Description of the parameter.
Properties	Additional properties from the manufacturer.
Properties > programRunCommand	Docker run command. This command runs when the customer runs the weather job.

Expand table

WeatherDataLocation	Description
weatherDataModelId	ID of the corresponding WeatherDataModel that was created during the bootstrap process.
location	Latitude, longitude, and elevation.
Name	Name of the object.
Description	Description of the weather data location.
farmId	(Optional) ID of the farm. The customer provides this ID as part of the job parameter.
Properties	Additional properties from the manufacturer.

> **Note**
>
> The APIs return unique IDs for each instance that's created. The translator for device management and metadata sync needs to retain this ID.

Metadata sync

The Connector Docker component should be able to send updates on the metadata. For example, it should send updates when the weather provider adds new parameters to a dataset or when new functionality, such as a new 30-day forecast, is added.

> **Note**
>
> Delete isn't supported for metadata in the weather data model.
>
> To update metadata, you have to call `/Get/{ID}` on the weather data model. Update the changed properties, and then do a `/Put/{ID}` to retain any properties that the user sets.

Weather data (telemetry) specifications

Weather data is mapped to a canonical message that's pushed to an Azure event hub for processing. Azure Event Hubs is a service that enables real-time data (telemetry) ingestion from connected devices and applications.

To send weather data to FarmBeats, create a client that sends messages to an event hub in FarmBeats. For more information, see Sending telemetry to an event hub.

The following sample Python code sends telemetry as a client to a specified event hub.

Python

```python
import azure
from azure.eventhub import EventHubClient, Sender, EventData, Receiver,
Offset
EVENTHUBCONNECTIONSTRING = "<EventHub connection string provided by
customer>"
EVENTHUBNAME = "<EventHub name provided by customer>"

write_client =
EventHubClient.from_connection_string(EVENTHUBCONNECTIONSTRING,
eventhub=EVENTHUBNAME, debug=False)
sender = write_client.add_sender(partition="0")
write_client.run()
for i in range(5):
    telemetry = "<Canonical telemetry message>"
    print("Sending telemetry: " + telemetry)
    sender.send(EventData(telemetry))
write_client.stop()
```

Here's the canonical message format:

JSON

```
{
   "weatherstations": [
   {
   "id": "ID of the WeatherDataLocation.",
   "weatherdata": [
   {
     "timestamp": "Timestamp of the data. For historical purposes, this is the time for which the observations are sent. For forecast, this is the time for which data is forecasted. Format is ISO 8601. Default time zone is UTC.",
     "predictiontimestamp": "Timestamp on which the forecast data is predicted. I.e., the time of prediction. Required only for forecast data. Format is ISO 8601. Default time zone is UTC. ",
     "weathermeasurename1": <value>,
     "weathermeasurename2": <value>
     }
     ]
    }
   ]
}
```

Here's an example of a telemetry message:

JSON

```
{
   "weatherstations": [
   {
     "id": "5e4b34e7-bf9e-4f39-xxxx-f3c06d0366ea",
     "weatherdata": [
     {
      "timestamp": "2019-07-10T00:00:00Z",
      "predictiontimestamp": "2019-07-05T00:00:00Z",
      "PrecipitationTotalLiquidEquivalent": 0,
      "AvgPressure": 0,
      "AvgRelativeHumidity": 72
     }
    ]
  }
 ]
}
```

Troubleshooting and error management

Error logging

The partner job runs in the existing job framework. So the errors are logged the same way as errors for other preexisting FarmBeats jobs (like GetFarmData and SensorPlacement). The Data Factory activity that runs within the Data Factory pipeline logs both `STDERR` and `STDOUT`. Both files are available in the `datahublogs-xxx` storage account within the FarmBeats resource group.

The Datahub lib provides helper functions to enable logging as part of overall Datahub logs. For more information, see the GitHub page for the NOAA Docker image.

Troubleshooting and support

If the customer can't receive weather data in the FarmBeats instance, provide support and a mechanism to troubleshoot the problem.

Add ExtendedType

FarmBeats supports adding new sensor measure types and units. You can add new units or types by updating the *bootstrap_manifest.json* file in the reference implementation.

Follow these steps to add a new WeatherMeasure type, for example, PrecipitationDepth.

1. Make a `GET` request on /ExtendedType by using the query `filter - key = WeatherMeasureType`.
2. Note the ID of the returned object.
3. Add the new type to the list in the returned object. Make a `PUT` request on the /ExtendedType{ID} with the following new list. The input payload should be the same as the response that you received earlier. The new unit should be appended at the end of the list of values.

Next steps

Now you have a Connector Docker component that integrates with FarmBeats. Next, find out how to get weather data by using your Docker image in FarmBeats.

Imagery partner integration

Article • 11/29/2023

This article describes how to use the Azure FarmBeats Translator component to send imagery data to FarmBeats. Agricultural imagery data can be generated from various sources, such as multispectral cameras, satellites, and drones. Agricultural imagery partners can integrate with FarmBeats to provide customers with custom-generated maps for their farms.

> **Important**
>
> Azure FarmBeats is retired. You can see the public announcement **here**.
>
> We have built a new agriculture focused service, it's name is Azure Data Manager for Agriculture and it's now available as a preview service. For more information see public documentation **here** or write to us at madma@microsoft.com.

Data, once available, can be visualized through the FarmBeats Accelerator and potentially be used for data fusion and machine learning/artificial intelligence (ML/AI) model building by agricultural businesses or customer system integrators.

FarmBeats provides the ability to:

- Define custom image types, source, and file format by using /ExtendedType APIs.
- Ingest imagery data from various sources via the /Scene and /SceneFile APIs.

The following information focuses on getting any form of imagery into the FarmBeats system.

When you select the **Drone Imagery** section, a pop-up opens to show a high-resolution image of the drone orthomosaic. You can access the partner software, which helps to plan drone flights and get raw data. You'll continue to use the partner's software for path planning and orthomosaic image stitching.

Drone partners need to enable customers to link their customer account with their FarmBeats instance on Azure.

You must use the following credentials in the drone partner software to link FarmBeats:

- API endpoint
- Tenant ID
- Client ID

- Client secret

API development

The APIs contain Swagger technical documentation.

Authentication

FarmBeats uses Microsoft Entra ID. Azure App Service provides built-in authentication and authorization support.

For more information, see Microsoft Entra ID.

FarmBeats Datahub uses bearer authentication, which needs the following credentials:

- Client ID
- Client secret
- Tenant ID

Using the previous credentials, the caller can request an access token, which needs to be sent in the subsequent API requests, in the header section, as follows:

```
headers = {"Authorization": "Bearer " + access_token, …}
```

The following Python code sample retrieves the access token. You can then use the token for subsequent API calls to FarmBeats.

Python

```python
import requests
import json
import msal

# Your service principal App ID
CLIENT_ID = "<CLIENT_ID>"
# Your service principal password
CLIENT_SECRET = "<CLIENT_SECRET>"
# Tenant ID for your Azure subscription
TENANT_ID = "<TENANT_ID>"

AUTHORITY_HOST = 'https://login.microsoftonline.com'
AUTHORITY = AUTHORITY_HOST + '/' + TENANT_ID

ENDPOINT = "https://<yourfarmbeatswebsitename-api>.azurewebsites.net"
```

```
SCOPE = ENDPOINT + "/.default"

context = msal.ConfidentialClientApplication(CLIENT_ID, authority=AUTHORITY,
client_credential=CLIENT_SECRET)
token_response = context.acquire_token_for_client(SCOPE)
# We should get an access token here
access_token = token_response.get('access_token')
```

HTTP request headers

Here are the most common request headers that need to be specified when you make an API call to FarmBeats Datahub.

Expand table

Header	Description and example
Content-Type	The request format (Content-Type: application/<format>). For FarmBeats Datahub APIs, the format is JSON. Content-Type: application/json
Authorization	Specifies the access token required to make an API call. Authorization: Bearer <Access-Token>
Accept	The response format. For FarmBeats Datahub APIs, the format is JSON. Accept: application/json

API requests

To make a REST API request, you combine:

- The HTTP method (GET, POST, and PUT).
- The URL to the API service.
- The resource URI (to query, submit data, update, or delete).
- One or more HTTP request headers.

Optionally, you can include query parameters on GET calls to filter, limit the size of, and sort the data in the responses.

The following sample request is to get the list of devices:

Bash

```
curl -X GET "https://microsoft-farmbeats.azurewebsites.net/Device" -H
"Content-Type: application/json" -H
"Authorization: Bearer <Access-Token>"
```

Most GET, POST, and PUT calls require a JSON request body.

The following sample request is to create a device. This sample has an input JSON with the request body.

```bash
curl -X POST "https://microsoft-farmbeats.azurewebsites.net/Device" -H
"accept: application/json" -H
"Content-Type: application/json" -H "Authorization: Bearer <Access-Token>" -
d
"{  \"deviceModelId\": \"ID123\",  \"hardwareId\": \"MHDN123\",
\"reportingInterval\": 900,
\"name\": \"Device123\",  \"description\": \"Test Device 123\",}"
```

Data format

JSON is a common language-independent data format that provides a simple text representation of arbitrary data structures. For more information, see JSON org.

Ingest imagery into FarmBeats

After the partner has credentials to connect to FarmBeats Datahub, the partner takes the following steps in the Translator component.

1. Create a new extended type for the following fields, in accordance with the type of imagery to be uploaded:

    - **Scene Source**: For example, drone_partner_name
    - **Scene Type**: For example, drone
    - **Scene File Type**: For example, chlorophyll index
    - **Scene File Content Type**: For example, image/tiff

2. Call the /Farms API to get the list of farms from within the Azure FarmBeats system.

3. Provide the customer with an ability to choose a single farm from the list of farms.

    The partner system must show the farm within the partner software to do the path planning and drone flight and image collection.

4. Call the /Scene API and provide required details to create a new scene with a unique scene ID.

5. Receive a blob SAS URL to upload the required images into FarmBeats Datahub, in the context of the chosen farm, in the FarmBeats system.

Here's a detailed flow on the API calls.

Step 1: ExtendedType

Check in the /ExtendedType API to see whether the type and file source are available on FarmBeats. To do so, call a GET on the /ExtendedType API.

Here are the system-defined values:

JSON


```
{
  "items": [
    {
      "id": "ae2aa527-7d27-42e2-8fcf-841937c651d7",
      "createdAt": "2019-06-17T07:32:30.1135716Z",
      "lastModifiedAt": "2019-10-25T09:47:42.5910344Z",
      "key": "SceneFileType",
      "value": [
        "evi",
        "ndvi",
        "ndwi",
        "tci",
        "soil-moisture",
        "sensor-placement",
        "sentinel-b01",
        "sentinel-b02",
        "sentinel-b03",
        "sentinel-b04",
        "sentinel-b05",
        "sentinel-b06",
        "sentinel-b07",
        "sentinel-b08",
        "sentinel-b8a",
        "sentinel-b09",
        "sentinel-b10",
        "sentinel-b11",
        "sentinel-b12",
        "cloud-mask",
        "farm-mask",
        "image/tiff",
        "orthomosaic"
      ],
      "description": "List of scene file types available in system. User can
add more values."
    },
    {
      "id": "b7824d2e-3e43-46d3-8fd4-ec7ba8918d84",
```

```
      "createdAt": "2019-06-17T07:32:30.1135716Z",
      "lastModifiedAt": "2019-10-25T09:45:44.8914691Z",
      "key": "SceneFileContentType",
      "value": [
        "image/tiff",
        "image/png",
        "image/jpeg",
        "text/csv",
        "text/plain",
        "text/tab-separated-values",
        "application/json",
        "application/octet-stream",
        "orthomosaic"
      ],
      "description": "List of scene file content types available in system.
User can add more values."
    },
    {
      "id": "21d2924d-8b3e-4ce4-a7f8-318e85c0f7f2",
      "createdAt": "2019-06-17T07:32:30.1135716Z",
      "lastModifiedAt": "2019-10-25T09:45:37.5096445Z",
      "key": "SceneSource",
      "value": [
        "sentinel-l1c",
        "sentinel-l2a",
        "farmbeats-model",
        "dji",
        "SlantRange-3P",
        "DJI"
      ],
      "description": "List of scene sources available in system. User can
add more values."
    },
    {
      "id": "6e09ca69-cacf-4b53-b63a-53c659aae4a4",
      "createdAt": "2019-06-17T07:32:30.1135716Z",
      "lastModifiedAt": "2019-10-25T09:45:34.7483899Z",
      "key": "SceneType",
      "value": [
        "base-bands",
        "sensor-placement",
        "soil-moisture",
        "evi",
        "ndwi",
        "ndvi",
        "drone",
        "orthomosaic-drone"
      ],
      "description": "List of scene types available in system. User can add
more values."
    },
    {
      "id": "55d8436b-3a86-4bea-87ab-e1b51226525f",
      "createdAt": "2019-06-17T07:32:30.1135716Z",
      "lastModifiedAt": "2019-06-17T07:32:30.1135716Z",
```

```
      "key": "SensorMeasureType",
      "value": [
        "AmbientTemperature",
        "CO2",
        "Depth",
        "ElectricalConductivity",
        "LeafWetness",
        "Length",
        "LiquidLevel",
        "Nitrate",
        "O2",
        "PH",
        "Phosphate",
        "PointInTime",
        "Potassium",
        "Pressure",
        "RainGauge",
        "RelativeHumidity",
        "Salinity",
        "SoilMoisture",
        "SoilTemperature",
        "SolarRadiation",
        "State",
        "TimeDuration",
        "UVRadiation",
        "UVIndex",
        "Volume",
        "WindDirection",
        "WindRun",
        "WindSpeed",
        "Evapotranspiration",
        "PAR"
      ],
      "description": "List of sensor measure types available in system. User
can add more values."
    },
    {
      "id": "0dfd6e6b-df58-428f-9ab8-a0674bfdcbe5",
      "createdAt": "2019-06-17T07:32:30.1135716Z",
      "lastModifiedAt": "2019-06-17T07:32:30.1135716Z",
      "key": "SensorMeasureUnit",
      "value": [
        "NoUnit",
        "Celsius",
        "Fahrenheit",
        "Kelvin",
        "Rankine",
        "Pascal",
        "Mercury",
        "PSI",
        "MilliMeter",
        "CentiMeter",
        "Meter",
        "Inch",
        "Feet",
```

```
        "Mile",
        "KiloMeter",
        "MilesPerHour",
        "MilesPerSecond",
        "KMPerHour",
        "KMPerSecond",
        "MetersPerHour",
        "MetersPerSecond",
        "Degree",
        "WattsPerSquareMeter",
        "KiloWattsPerSquareMeter",
        "MilliWattsPerSquareCentiMeter",
        "MilliJoulesPerSquareCentiMeter",
        "VolumetricWaterContent",
        "Percentage",
        "PartsPerMillion",
        "MicroMol",
        "MicroMolesPerLiter",
        "SiemensPerSquareMeterPerMole",
        "MilliSiemensPerCentiMeter",
        "Centibar",
        "DeciSiemensPerMeter",
        "KiloPascal",
        "VolumetricIonContent",
        "Liter",
        "MilliLiter",
        "Seconds",
        "UnixTimestamp",
        "MicroMolPerMeterSquaredPerSecond",
        "InchesPerHour"
      ],
      "description": "List of sensor measure units available in system. User
can add more values."
    }
  ]
}
```

This step is a one-time setup. The scope of this new scene type is limited to the subscription in which the Azure FarmBeats is installed.

For example, to add SceneSource: "SlantRange," you do a PUT on the ID of the /ExtendedType API with the key "SceneSource" input payload.

JSON

```
{

      "key": "SceneSource",
      "value": [
        "sentinel-l1c",
        "sentinel-l2a",
        "farmbeats-model",
```

```
        "dji",
        "SlantRange-3P"
      ],
      "description": "List of scene sources available in system. User can
add more values."
}
```

The green field is the new addition to the system-defined scene source values.

Step 2: Get farm details

The scenes (.tiff or .csv files) are in the context of a farm. You need to get the farm details by doing a GET on the /Farm API. The API returns the list of farms that are available in FarmBeats. You can select the farm you want to ingest the data for.

GET /Farm response:

JSON

```
{
  "id": "07f3e09c-89aa-4619-8d50-e57fb083d8f9",
  "createdAt": "2019-11-01T13:55:41.8804663Z",
  "lastModifiedAt": "2019-11-01T13:55:41.8804663Z",
  "geometry": {
    "type": "Polygon",
    "coordinates": [
      [
        [
          78.34252251428694,
          17.402556352862675
        ],
        [
          78.34278000635095,
          17.407920852463022
        ],
        [
          78.34883106989963,
          17.40878079576528
        ],
        [
          78.3498181228054,
          17.403539173730607
        ],
        [
          78.34805859369493,
          17.39977166504127
        ],
        [
          78.34252251428694,
          17.402556352862675
```

```
        ]
      ]
    ]
  },
  "name": "Samplefarm",
  "properties": {
    "crops": "Corn",
    "address": "Sample street"
  }
}
```

Step 3: Create a scene ID (POST call)

Create a new scene (.tiff or .csv file) with the given information, which provides the date, sequence, and farm ID with which the scene is associated. The metadata associated with the scene can be defined under properties, which includes the duration and type of measure.

Creating a new scene creates a new scene ID, which is associated with the farm. After the scene ID is created, the user can use the same to create a new file (.tiff or .csv) and store the contents of the file.

Example input payload for the POST call on the /Scene API:

JSON

```
{
  "sceneId": "a0505928-c480-491b-ba31-d38285a28c1d",
  "type": "newtype",
  "contentType": "image/tiff",
  "name": "test scene file",
  "description": "test scene file description",
  "properties": {
    "additionalProp1": {},
    "additionalProp2": {},
    "additionalProp3": {}
  }
}
```

API response:

JSON

```
{
  "id": "a0505928-c480-491b-ba31-d38285a28c1d",
  "createdAt": "2019-10-04T16:19:12.4838584Z",
  "lastModifiedAt": "2019-10-04T16:19:12.4838584Z",
  "type": "new type",
```

```
  "source": "SlantRange-3P",
  "farmId": "d41a33e7-b73e-480e-9279-0fcb3207332b",
  "date": "2019-10-04T16:13:39.064Z",
  "sequence": 5,
  "name": "test scene",
  "description": "test scene description",
  "properties": {}
}
```

Create a scene file

The scene ID returned in step 3 is the input for the scene file. The scene file returns an SAS URL token, which is valid for 24 hours.

If the user requires a programmatic way of uploading a stream of images, the blob storage SDK can be used to define a method by using the scene file ID, location, and URL.

Example input payload for the POST call on the /SceneFile API:

JSON

```
{
  "sceneId": "a0505928-c480-491b-ba31-d38285a28c1d",
  "type": "newtype",
  "contentType": "image/tiff",
  "name": "test scene file",
  "description": "test scene file description",
  "properties": {
    "additionalProp1": {},
    "additionalProp2": {},
    "additionalProp3": {}
  }
}
```

API response:

JSON

```
{
  "uploadSASUrl": "https://storagej2lho.blob.core.windows.net/farm-
scene/2019/a0505928-c480-491b-ba31-d38285a28c1d/e91139a7-4ebd-4e2f-b17c-
c677822dc840?sv=2018-03-
28&sr=b&sig=%2F1426JkDcIFE5g3d%2BjOevCVMIn%2FJo9YKwBn3La5zL8Y%3D&se=2019-10-
05T16%3A23%3A57Z&sp=w",
  "id": "e91139a7-4ebd-4e2f-b17c-c677822dc840",
  "createdAt": "2019-10-04T16:23:57.1192916Z",
  "lastModifiedAt": "2019-10-04T16:23:57.1192916Z",
  "blobUrl": "https://storagej2lho.blob.core.windows.net/farm-
```

```
scene/2019/a0505928-c480-491b-ba31-d38285a28c1d/e91139a7-4ebd-4e2f-b17c-
c677822dc840",
  "sceneId": "a0505928-c480-491b-ba31-d38285a28c1d",
  "type": "newtype",
  "contentType": "image/tiff",
  "name": "test scene file",
  "description": "test scene file description",
  "properties": {}
}
```

The POST call to the /SceneFile API returns an SAS upload URL, which can be used to upload the .csv or .tiff file by using the Azure Blob storage client or library.

Next steps

For more information on REST API-based integration details, see REST API.

Environment monitoring and supply chain optimization with IoT

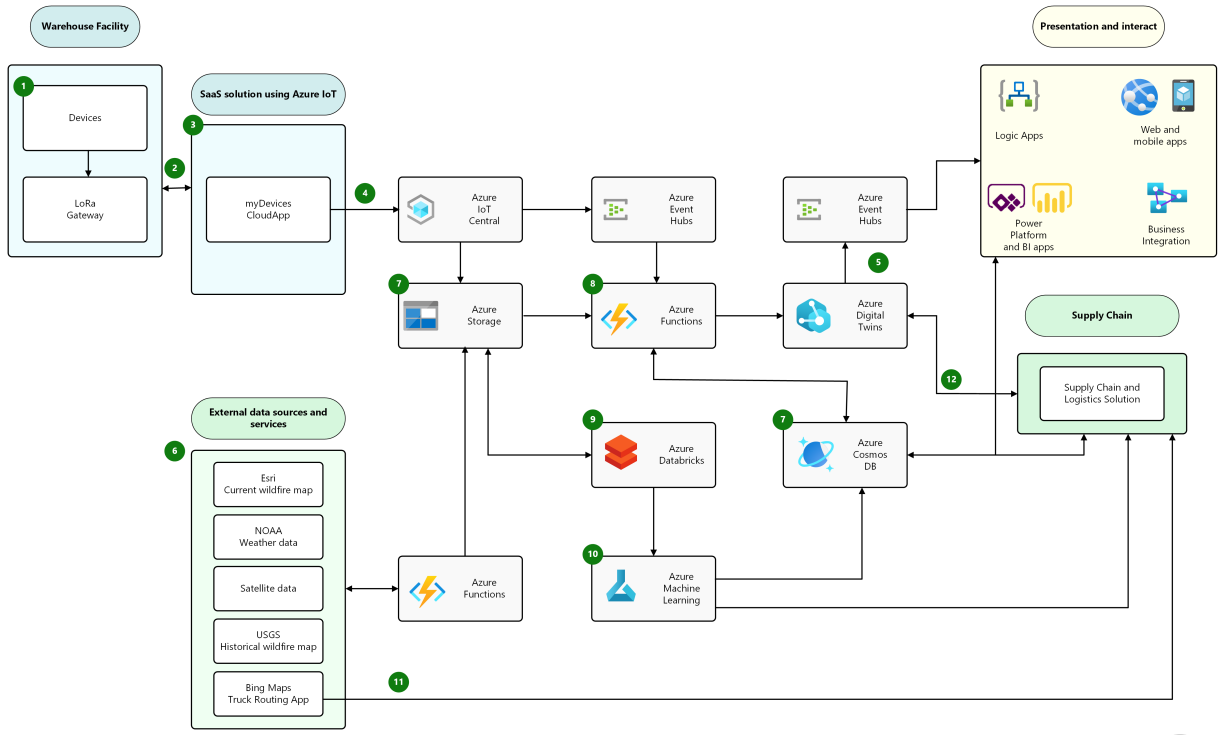
Azure Functions | Azure IoT | Azure Storage | Azure Machine Learning | Azure Databricks | Azure Monitor

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**.

This article describes a warehouse management scenario that monitors environmental conditions through the integration of sensor data and public datasets, processed with ML to generate predictions. The insights are then used to ensure the safety of people and optimize supply chain operations.

Architecture



Microsoft Azure

Download a Visio file of this architecture.

Dataflow

1. Sensors in the warehouse facility are connected and send data to a LoRa (**Lo**ng **Ra**nge) gateway.
2. LoRa gateway pushes data to the cloud using cellular connectivity.
3. myDevices is a software as a service (SaaS)-based plug and play solution. It uses devices and gateways that are automatically provisioned and associated with the corresponding customer.
4. Device data is sent to Azure IoT Central. Customers use the solution for controlling and monitoring the devices.
5. Modeling of the supply chain and warehouse facilities using Azure Digital Twins. This is a live executing environment where applications can ingest data to gain visibility on the status of the supply chain. Digital Twins integrates natively with Azure Event Hub, which other applications interact with to retrieve data from the twin.
6. Temporal and spatial data required by ML models is obtained from the external data sources.
7. Key data is stored in Azure data solutions. Blob storage is used for ML training data. Azure Cosmos DB is used for scored data and key performance indexes.
8. Telemetry data is ingested from IoT Central via Event Hub to ensure decoupling of the data ingestion and consumption. Azure Functions is used to combine external data sources and telemetry data, and then analyze this data set for any anomalies. The data is surfaced through Digital Twins.
9. Azure Databricks performs data transformations required for training the ML models.
10. Wildfire prediction models are trained using Azure Machine Learning by utilizing historical data, real-time data, and micro-weather data.
11. Routing updates are provided by the Bing Maps Truck Routing API.
12. Applications can query Digital Twins directly to obtain relevant data from the model.

Components

- Azure IoT Central is used as the IoT managed platform. It provides security, scalability, and availability as part of the service so that customers can focus efforts on business requirements. Users can integrate with business components such as Power Apps and Power BI, and create notifications through the data export feature in IoT Central.

- Azure storage is used for storing device information in the cloud in a secure and scalable way that is also cost-effective. The data stored is used for training the ML models.

- Azure Cosmos DB is used to store application key performance indicators (KPIs) and model outputs. Azure Cosmos DB is a fully managed NoSQL database service for modern application development. It provides high-speed transactions and can easily enable the service for global distribution.

- Azure Databricks is a data analytics platform optimized for the Microsoft Azure cloud services platform. It's used for transforming, manipulating, and normalizing data so that it can be properly consumed by the machine learning pipeline.

- Azure Machine Learning is used to create wildfire prediction models. The models provide the intelligence required to assess the risk of a wildfire. Input from multiple data sources is required to train the model for accuracy. These sources can include satellite imagery, historical data, local soil conditions, and weather data. Based on the predicated wildfire area from the model, the Supply Chain and Logistics Solution can reroute trucks.

For more detailed discussions, see the Azure IoT reference architecture to understand and explore the various implementation choices available.

Scenario details

Environmental monitoring has become an important activity in the global supply chain. It provides key signals that help drive real-time decisions that can impact suppliers and logistics. Air quality, temperature, wind, humidity, and carbon dioxide (CO2) are some of the indicators that warehouse operators are interested in monitoring during natural disasters. More advanced scenarios can include the fusion of real-time and historical data from weather stations, air quality sensors, and other sources. Machine learning (ML) models can then be used to help predict the effect of these conditions and their possible impact on supply chain operations.

Potential use cases

This solution is ideal for the environment, manufacturing, transportation, and agriculture industries.

- **Fleet management:** This solution can be used where routes need to be optimized for safety based on the evolving conditions of surrounding areas.
- **Agriculture:** Predicting wildfires that will impact the safety of workers and livestock is critical. By providing ample lead time for danger notifications, people in the affected area can evacuate to safety. Farms can also equip livestock areas with automated gates that can unlock and open in dire situations, allowing animals to escape.

Challenges faced

There has been a significant increase in wildfires in recent years, posing a growing danger to humans and the global supply chain. With the number of acres burned increasing by the year, supply chain resilience to climate change is top of mind for many leaders.

In the United States, the annual average of area affected by wildfires is approximately 7,000,000 acres. This area is more than double the average during the 1990s. The situation is even more alarming in other countries/regions. For example, in Australia there's an extra month of summer compared to 50 years ago and long-term drought has further worsened fire conditions. Massive bush fires there have caused economic damages nearly 10 times greater than in the United States. Australian wildfires can impact the global food supply, including goods like beef, milk, wine, and wheat.

Risks to businesses worldwide continue to increase each year, and supply chain resiliency during natural disasters is critical to maintaining the flow of goods globally. Integrating weather-based forecasts and predictions into supply chain capacity planning can help operators adjust production and manage shipping schedules. This system can minimize disruptions and adverse effects.

Business outcomes

Warehouse operators and major distribution centers will benefit from a predictive way to determine if existing logistics infrastructure is in the path of a major fire. Having an early notification system would provide increased lead time to take preventive measures to protect facilities and personnel. Automated notices of changes and pauses in logistics activities would also allow rerouting of shipments with minimal human intervention.

Requirements

- Automation is critical. It can't be assumed that operators and facility managers can gather data across multiple systems to make timely decisions.
- Warehouses, distribution facilities, and operation managers need to be notified by multiple means when there's immediate danger, ensuring the information is received in a timely fashion. Examples include data dashboard, email, and text message.
- Only changes in the data need to be reported.
- The delivery and deployment of the solution needs to be simple. It should be installed without the need of a technician, using plug and play technology.
- The solution needs to be low maintenance and cost effective.

Patterns to address challenges

The table below provides a summary of common use cases and corresponding IoT solutions. Each use case is an example of how an IoT process pattern can be applied to real-world scenarios.

Expand table

Use case	Solutions
Enable supply chain logistics rerouting and production planning by predicting likelihood of interruption due to wildfires near the impacted location.	Ideally, you want to be able to monitor all the key elements of the supply chain so that you can provide a more comprehensive response. myDevices has a catalog of certified plug and play devices that connect to a LoRa network gateway. The gateway sends data to the cloud application using cellular connectivity. LoRa technology is ideal because the signal needs to penetrate deep into the buildings. Sensors for CO2, temperature, humidity, wind direction, and air quality can be installed in relevant building locations including roofs and storage facilities. Sensors can also be installed in trucks for location tracking to facilitate rerouting.
Identify wildfire conditions and understand the degree of danger for a given location.	Wildfire prediction models trained with historical data, micro-weather conditions, and local sensor data can help assess the risk of a wildfire.
Automated alerts for evacuation and facility	Once unsafe conditions are detected, the digital twin of the facility can then be updated to show that it's no longer online.

Use case	Solutions
rerouting	Once updated, other distribution centers within the network can begin to reroute traffic accordingly, allowing on-site facility managers and warehouse operators to focus on employee safety. This scenario uses ML to predict where the wildfire will spread, using public real-time and historical data sets along with micro weather data for more accurate predictions. Sensors track current wildfire conditions, and facility alarms trigger employee evacuation.

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.

Connectivity

The solution's on-site devices and sensors need to send data to applications in the cloud, but reliable internet access may not be available for some locations, such as in rural areas.

This solution uses a LoRa network to provide cellular connectivity. LoRa has good building penetration, making it ideal for warehouse-related applications. This approach is cost-effective and provides flexibility for remote locations that require easy-to-connect IoT devices and sensors.

Plug and play

In a remote setting, it's critical that devices are easy to deploy without requiring specialized expertise. myDevices has an extensive catalog of IoT devices and gateways that can be applied to multiple scenarios. They're Plug and Play certified, so all the user needs to do is place them in the right location and turn them on. With their IoT Central integration, customers can easily customize their dashboard to consume their device data and create alerts.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- Ansley Yeo | Technology Leader

Next steps

- IoT in transportation and logistics: How Azure can be used to bring greater efficiency and reliability to your value chain with world-class IoT and location intelligence services.
- Architecture of IoT Central connected logistics application template: An app template and guidance for developing end to end connected logistics solutions.
- Bing Maps Truck Routing API: A commercial routing tool that calculates safe and efficient routes and considers a vehicle's attributes in relation to any route limitations.
- Azure Digital Twins - Supply Chain demo uses Digital Twins to model a supply chain scenario.
- myDevices provides LoRa connectivity and devices that enable solutions to be deployed quickly in locations where connectivity is a challenge and broad network coverage is needed.
- C.H. Robinson Navisphere works with Microsoft Azure and Azure IoT to provide real-time visibility into the supply chain and for more predictability and proactive decision making.
- EPA AirNow API provides access to real-time air quality and wildfire data from AirNow, a service maintained by the US Environmental Protection Agency (EPA) and other federal, tribal, state, and local agencies.

Related resources

- Azure IoT reference architecture
- Process real-time vehicle data using IoT: A reference architecture for ingesting real-time vehicle data for analysis, including route optimization.
- Condition monitoring for industrial IoT is an example scenario demonstrating how end manufacturers can connect their assets to the cloud using OPC UA (Open Platform Communication Unified Architecture) and the Industrial Components.
- IoT and data analytics is an example scenario for building solutions that integrate data from many IoT devices into a comprehensive data analysis architecture to improve and automate decision making.
- IoT monitor and manage loops is a design pattern describing a supervisory system that continually monitors a physical system controlled by a set of networked IoT devices.

- IoT analyze and optimize loops is a design pattern enabling the generation and application of business optimization insights to physical systems controlled by software by sourcing telemetry, refining it, and combining it with enterprise data sources to generate insights.
- IoT using Azure Cosmos DB describes an example architecture for using Azure Cosmos DB to ingest device telemetry data at high rates and can serve indexed queries back with low latency and high availability.

Low-latency network connections for industry

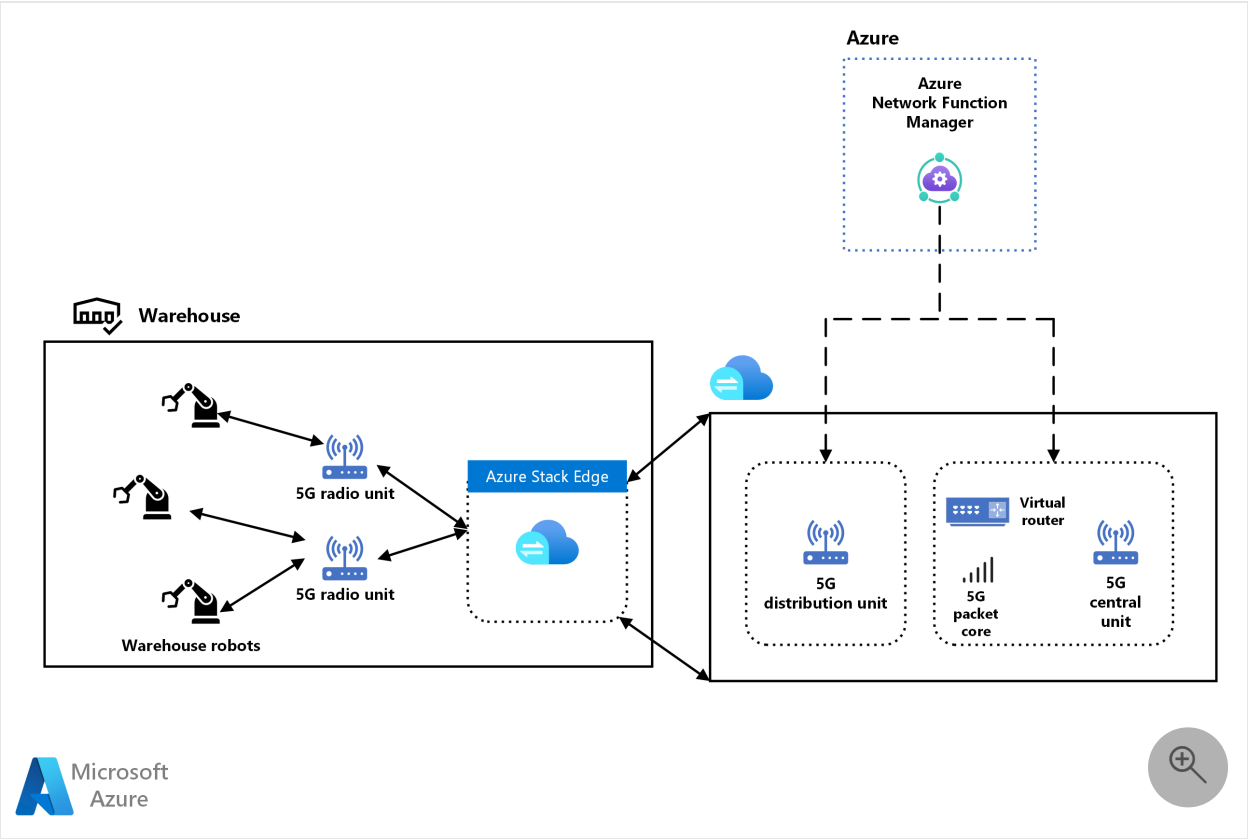
Azure Stack Edge

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 describes how manufacturers can use high-performance, low-latency 5G Standalone networks to scale up industrial automation and productivity.

Architecture



Download a Visio file of this architecture.

Dataflow

1. Embedded 5G-enabled internet protocol (IP) modules connect warehouse robots to 5G Open Radio Access Network (ORAN) radio units. RAN is a common wireless network infrastructure for mobile networks.
2. The 5G radio units connect over a wired switching network to the 5G distribution unit software, which runs on private multi-access edge compute (MEC) on Azure Stack Edge.
3. The 5G distribution unit connects with the virtual router, 5G central unit, and 5G packet core, which run on a separate Azure private MEC instance on Azure Stack Edge.
4. The 5G packet core provides device authentication, an IP address, and connectivity based on a preconfigured profile.
5. Azure Network Function Manager controls both MEC instances.
6. Optimizing the 5G network and keeping traffic confined to Azure private MEC provides the low latency these connection scenarios require.

Components

This solution uses the following Azure components:

- Azure Stack Edge is a portfolio of devices that bring compute, storage, and intelligence to the IoT Edge.
- Azure Network Function Manager enables the deployment of network functions to the IoT Edge using consistent Azure tools and interfaces.

Scenario details

5G Standalone networks reliably connect machines to other machines or controllers. 5G-enabled Internet of Things (IoT) devices like robots can communicate and operate autonomously on the factory or warehouse floors.

The 5G Standalone network is deployed as a Non-Public Network (NPN), with all data remaining on premises. The NPN configuration offers security, privacy, and reliability. Azure deploys and manages the network and devices.

Potential use cases

This solution is ideal for the manufacturing, agriculture, energy, facilities, telecommunications, and robotics industries. Use this approach for scenarios like:

- Picking shipments efficiently in a warehouse.

- Dispersing seeds from an autonomous seed sprayer machines on a farm, based on information from soil sensors.
- Conserving energy in commercial buildings by shutting off lights when no motion is detected in a room.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- Nikhil Ravi | Product Management Leader

Next steps

- Azure Private MEC
- Azure Industrial IoT
- Azure Network Function Manager simplifies 5G deployments (Video)

Related resources

- Connect an on-premises network to Azure
- Condition monitoring for Industrial IoT

Solutions for the sports industry

Article • 08/31/2023

From tracking and analyzing action on the court, to providing real-time streaming of events, to making tickets automatically appear on fans' phones as they approach the stadium gate, the sports industry has long been a leading user of high-tech solutions like Azure.

Watch this video to find out about how the NBA uses SAP together with Azure to facilitate its cloud evolution:

https://www.youtube-nocookie.com/embed/gAa3k3UFFsw

This short video features Utah Jazz guard Donovan Mitchell, who relies on NBA CourtOptix, powered by Azure, to generate metrics about his playing.

Architectures for the sports industry

The following articles provide detailed analysis of architectures created and recommended for the sports industry.

Guide	Summary	Technology focus
Build a sports analytics architecture on Azure	Build an architecture that uses Azure services to process and maintain data that's used by sports analytics solutions.	Analytics
Create smart places by using Azure Digital Twins	Use Azure Digital Twins to create models of smart places, like stadiums, from IoT device data. View and control products, systems, environments, and experiences.	IoT
Intelligent e-commerce product search engine	Use a dedicated search service to dramatically increase the relevance of search results for your e-commerce customers. Promote recent information in search results, which is useful in the sports industry.	Web

Solution ideas for the sports industry

Following are other ideas that you can use as a starting point for your sports industry solution.

- IoT Connected Platform for COVID-19 detection and prevention
- Live streaming digital media
- Video-on-demand digital media

Overview of the Fan Engagement solution

Article • 12/15/2023

The Fan Engagement solution of the media and entertainment accelerator includes the following features:

- An extension to Common Data Model to include concepts for fan engagement, including 60-plus new entity definitions and relationships.
- Three model-driven applications with new forms and dashboards that offer data entry and data management in support of Fan Engagement scenarios:
  - Event and venue management
  - Sports management
  - Guest management
- Two additional model-driven applications that extend the out-of-the-box Dynamics 365 Sales capabilities and customize them for media-specific product selling:
  - Media product sales and ticketing
  - Display and advertising sales
- A sample Power Apps portal, enabling self-registration of media events. Attendees can browse all upcoming events and sign up. This information is automatically captured and stored in your instance of the media-enhanced Common Data Model as new guest and registration entities. Built-in integration with Microsoft Teams enables live virtual events to be accessed by registered attendees directly through the portal.
- A solution that you can deploy and install from GitHub.

Site map extensions

With the Dynamics 365 media and entertainment accelerator, institutions can optimize fan engagement, improve interactions, profile customers and predict their actions, and gain insights from analytics. When the accelerator is installed in Dynamics 365, the experience is transformed into one specifically built for operators of theme parks, sport venues, theaters, arenas, or other places for tourism, hospitality, business conferencing, or the arts. The goal is to allow such institutions to quickly build Power Apps and Power BI visualizations.

Media and entertainment institutions can build their business processes on top of the entities in the accelerator. The ribbon customizations contain entities to record ticketing products, sponsorships, events, venues, sport teams and leagues, and more.

Entities and workflows

This accelerator provides these new Common Data Model entities to support the needs of media and entertainment businesses:

Fan and guest management entities

- Guest
- Guest category
- External guest IDs
- Interests
- Loyalty programs and levels
- Registrations
- Promotion codes

Event and venue management

- Venues
- Facilities
- Attractions, attraction tracks, and attraction schedules
- Concession stands and partners
- Gates
- Hours of operations
- Weather conditions
- Talent
- Media assets and asset categories
- Microsoft Teams integration for supporting attractions that will be live and online for remote attendees
- Portal publishing information

Sports management entities

- League
- Team
- Player
- Player team history

- Round
- Playing field
- Conference
- Division
- Season

Ticket sales

- Tickets
- Admissions
- Seat maps, seat tiers, seats, sections, and rows

Advertising sales

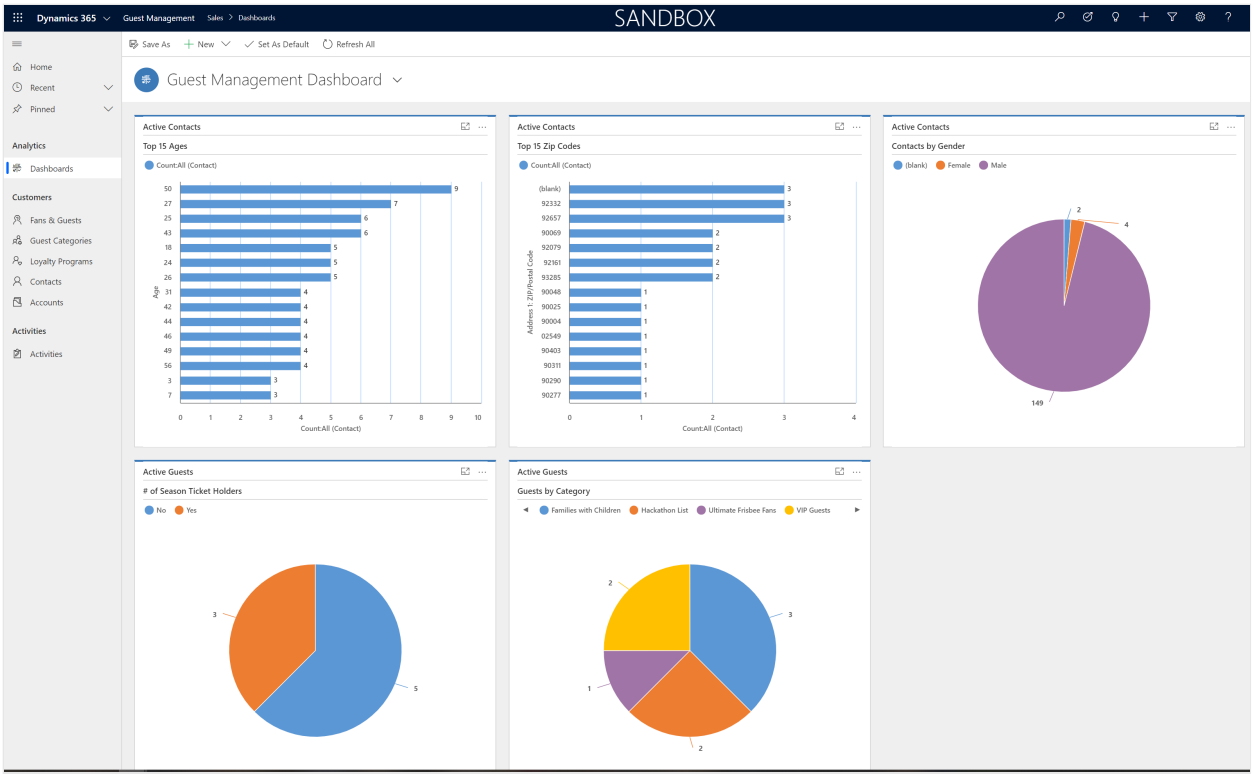
- Sponsorships
- Ad zones
- Ad segments and ad segment groups
- Ad spots
- TMAs (Television market areas)
- Regions
- Zip Codes

Forms and dashboards

The Dynamics 365 media and entertainment accelerator combines standard Dynamics 365 entities with customized entities to make it easier to build solutions. This section describes some of the forms, views, and dashboards that demonstrate the new entities and the data model.

Fan and guest management

The media and entertainment accelerator is intended to help successfully drive the progression of a customer from an anonymous or casual guest toward becoming a deeply engaged fan, regular attendee, subscribing member, or an active member of a frequent buyer or loyalty program. The accelerator solution helps track that journey with the addition of the new "guest" and "loyalty program" entities, which can help eventually with the conversion to leads, and then to customers and contacts.



Maggy Fruend
Guest

General | Favorites | External Guest IDs | Guest History | Leads | Media Registrations | Related

Name *	Maggy Fruend	Category	VIP Guests
Display Name	Maggy F	Contact	---
Telephone	206-555-1212	Company	---
Email Address	maggy@live.com	Is a Season Ticket Holder	Yes
Guardian	---	OK to send Email	Yes
Special Requirements	Gluten-Free Diet	Marketing Opt In	Yes
External Created Date	4/28/2020 9:30 AM		

New | Deactivate | Delete | Refresh | Assign | Share | Email a Link | Flow | Word Templates | Run Report

Microsoft Rewards
Loyalty Program

General | Related

Name *	Microsoft Rewards
Description	Redeem your points for gift cards, sweepstakes entries, nonprofit donations, and more. It's the easiest way to get rewarded for doing what you already love to do.
Loyalty Unit Type *	Points
Total # of Accounts	3
Last updated:	5/5/2020 4:47 PM
Owner *	[illegible]

Loyalty Levels

New Loyalty Level | Add Existing Loyalty L...

Name	Number	Loyalty Program	Fixed Guest Level
Gold	1	Microsoft Rewards	Loyalty Program
Platinum	254112	Microsoft Rewards	Known Guest

All # A B C D E F G H I J K L M N O P Q R S T U V W X

Gold
Loyalty Level

General Loyalty Program Accounts Related

Name	*	**Gold**
Description		**Additional Rewards and Privileges**
Loyalty Program		**Microsoft Rewards**
Number	*	**1**
VIP Flag		**No**
Fixed Guest Level		**Loyalty Program**
Balance Minimum Requirement		**50,000.00**

Events and venue management

The media data entities included in the accelerator enable powerful ways to activate and connect data gathered across your entire event and venue management spectrum. This coverage includes not only scheduling industry events such as conferences, concerts, sporting events, and art festivals, but also planning all of the attractions, rides, speakers, and concession stands within a venue (such as theme parks, museums, or tourist sites).

Version 2 of the accelerator adds direct integration with Microsoft Teams so that the events and attractions can include live Teams meetings or broadcasts. These Teams meetings are automatically scheduled through the app. Once the meeting is saved, a Teams link is available that allow attendees to join in real time. This event can be optionally pushed to the event registration portal.

Microsoft Build 2019

Media Event

General | Event Details | Ticketing | Venues | Talent | Concession Partners | Child Events | Weather Conditions | Related

Name *	Microsoft Build 2019	Is Sold Out	Yes
Event Type	Conference	Sold Out On	2/1/2019
Description	Experience the latest technologies and learn new ways to solve your development challenges. At Microsoft Build, hear from the engineers behind the products you use every day and connect with your peers in a digital	Total Cost	$1,900,000
Season	Microsoft FY19 Shows	Total Profit	$2,000,000
Start Date	5/6/2019 8:00 AM	Event Number	ME-1001
End Date	5/8/2019 8:00 PM	Event URL	https://www.microsoft.com/en-us/build
Duration	3	Number Registered	1
		Last updated:	5/5/2020 4:51 PM
Duration Unit	Days	Number Attended	149
Capacity	5,000	Last updated:	5/5/2020 4:51 PM

New Media Event

General | Event Details | Ticketing | Venues | Talent | Concession Partners | Child Events | Weather Conditions

Name * ---

Event Type --Select--

Description

Season

Start Date

End Date

Duration

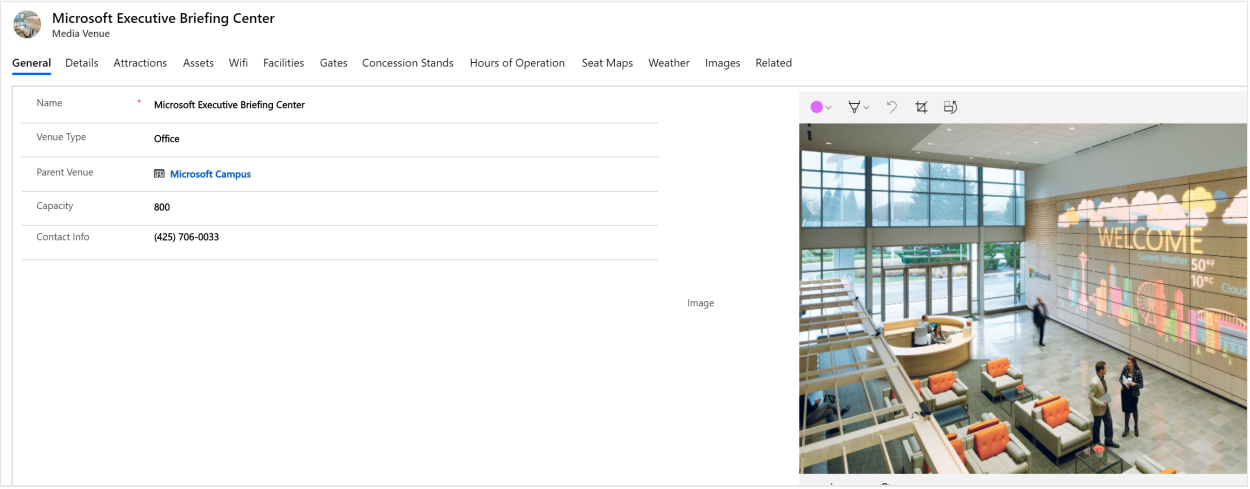
Duration Unit

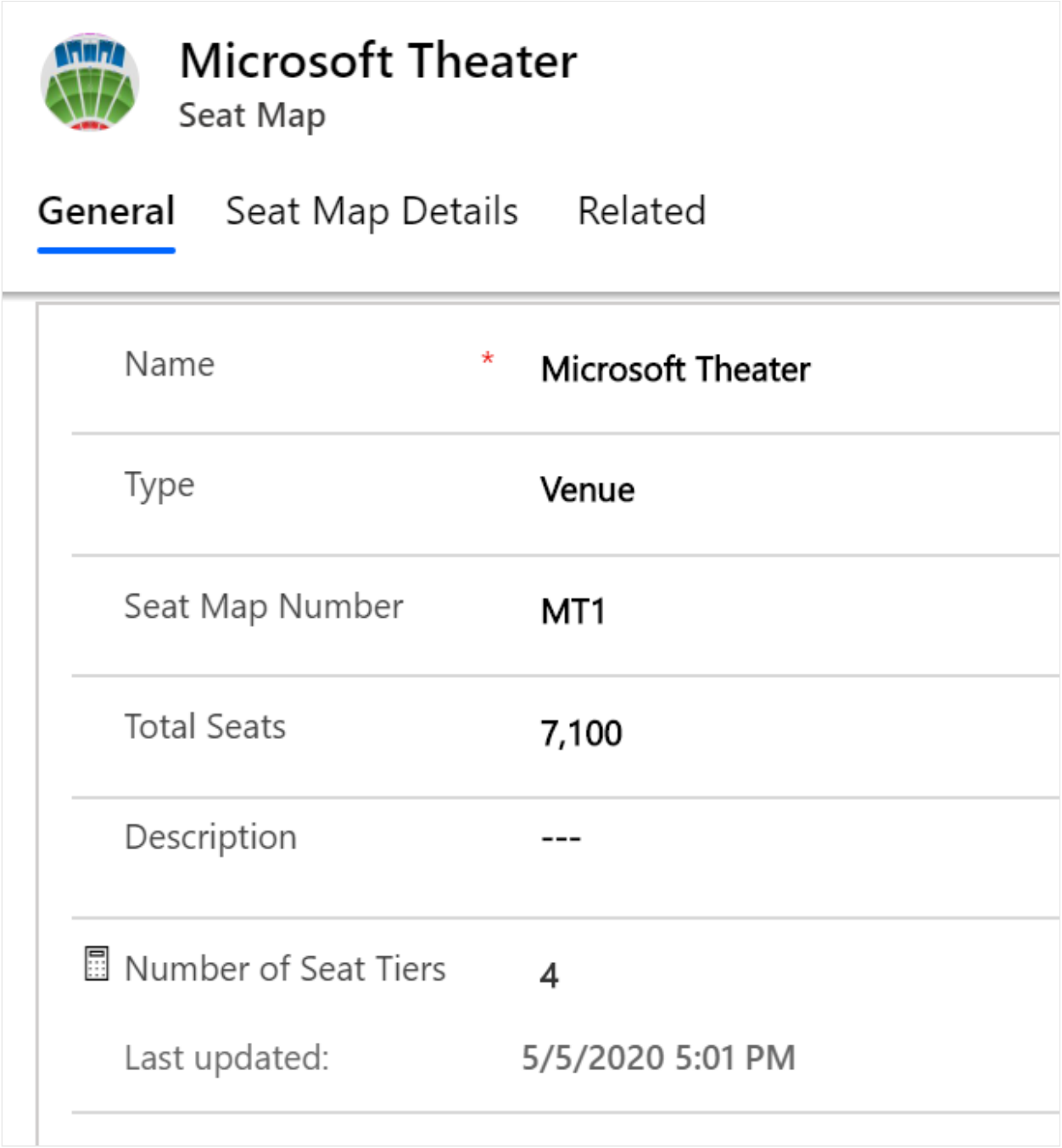
Capacity

- --Select--
- Live Broadcast
- Music Concert
- Arts Festival
- Fair/Carnival
- Theatrical Event
- Sporting Game or Match
- Conference
- Hackathon
- Show
- Dance
- Electronic Gaming Event
- Lecture/Seminar/Speaker Event
- Food/Beverage Event

Event Schedules

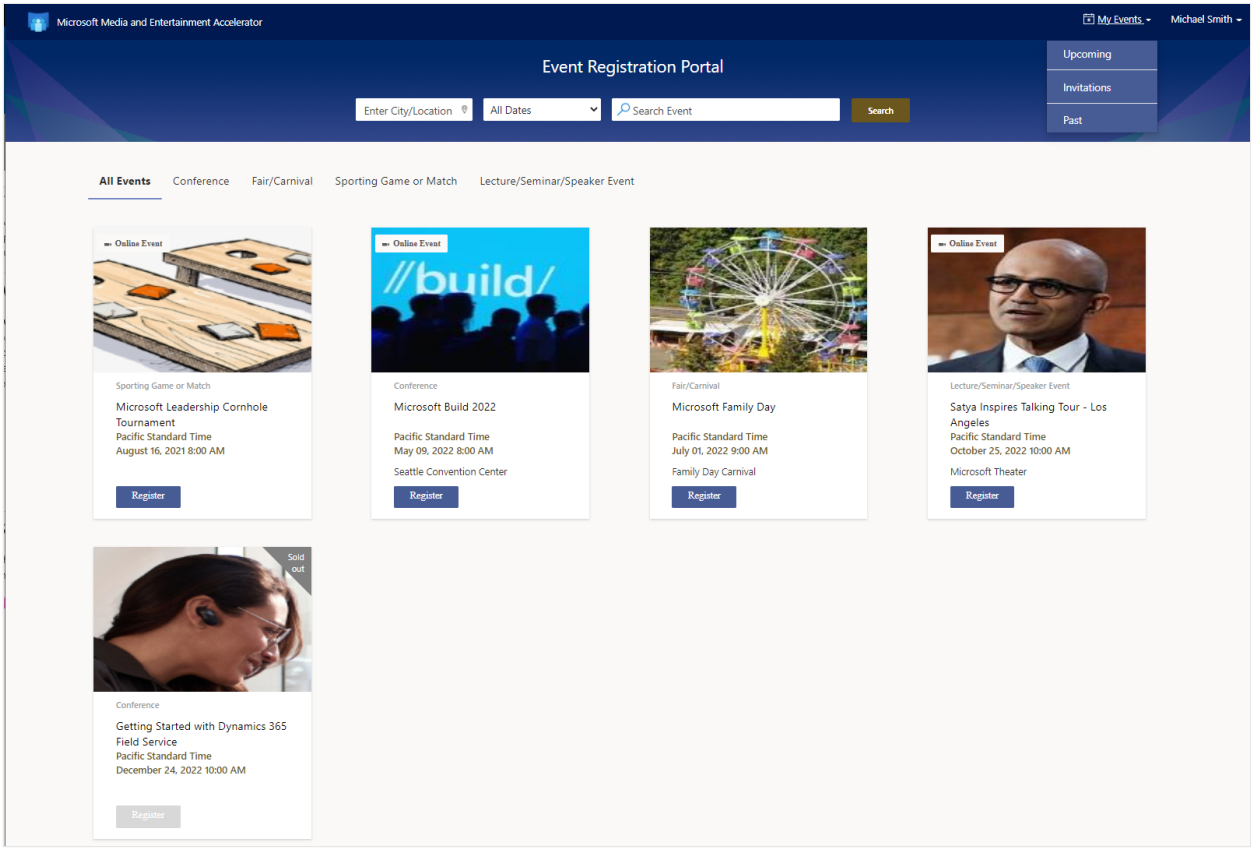
Event Attractions

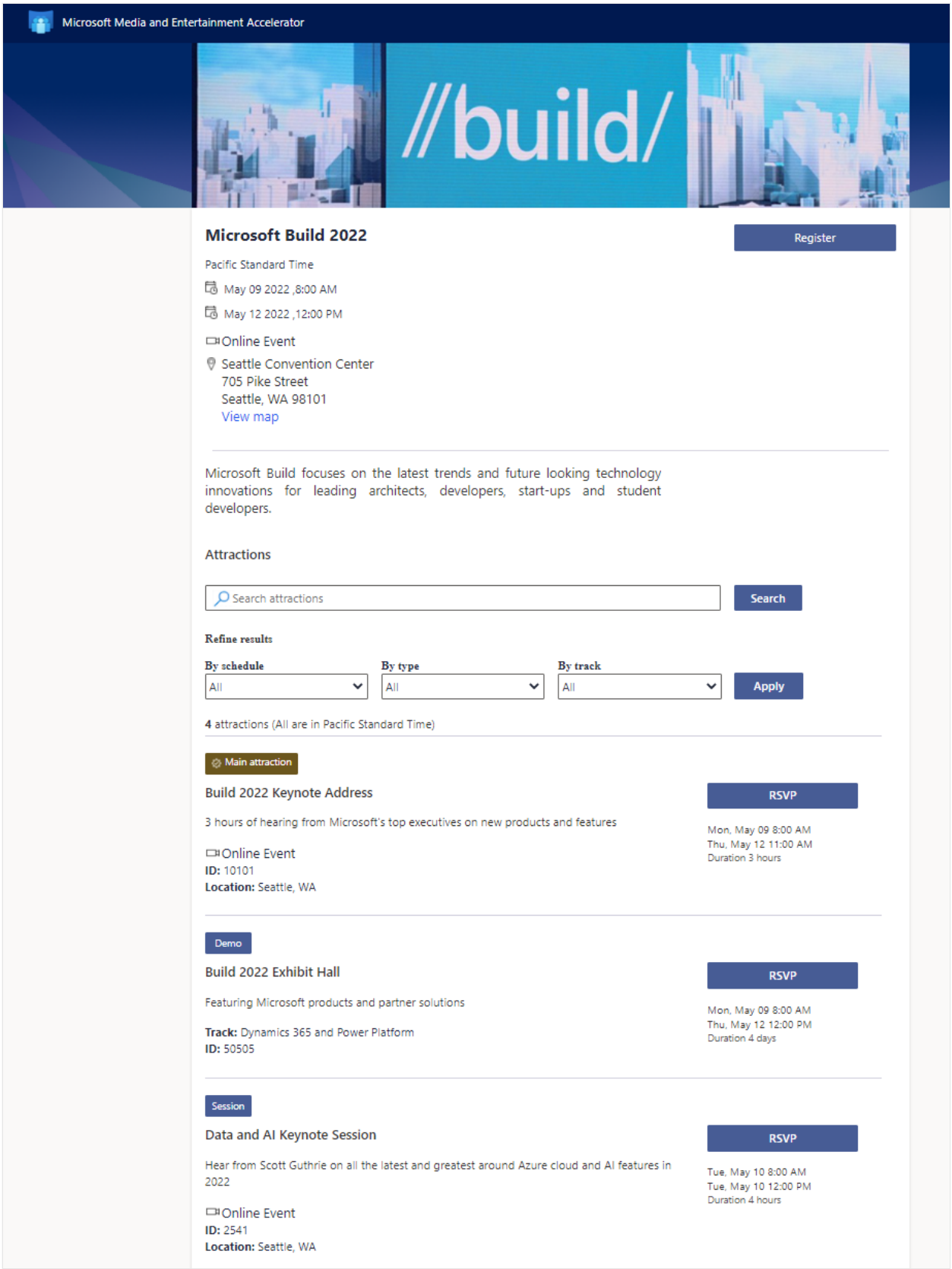




Event registration portal

The event and registration portal allows consumers to search for upcoming media events, view their details (dates, locations, descriptions), and register to attend the event. Consumers search for events and refine their search based on event type, location, and date. In version 2.0 of the accelerator, the Power Apps portal has been integrated with Microsoft Teams to allow attendees to join online events, concerts, or sports matches and interact with scheduled group chats, shared broadcasts, and virtual group sessions without the need for a separate Teams client.





Sports management

Key customer targets of the fan engagement scenarios within the media and entertainment accelerator are sports leagues, teams, and organizations that can use the accelerator to manage and schedule their seasons, games, players, referees, scorers, and other key officials. The accelerator adds many sports-specific data entities and relationships that enable specific use cases for the sports industry.

Sports Administration

- Leagues
- Teams
- Players
- Conferences
- Divisons
- Seasons
- Rounds

CS

Commons Soccer Field

Playing Field

General | Playing Field Points | Images | Related

Name	*	Commons Soccer Field
Parent Media Venue		Microsoft Commons Field
Sport Type		Soccer
Shape		Rectangle
Field Material		Grass
Has a Roof?		No
Has Lights?		Yes
Latitude		---
Longitude		---
Length		---
Width		---

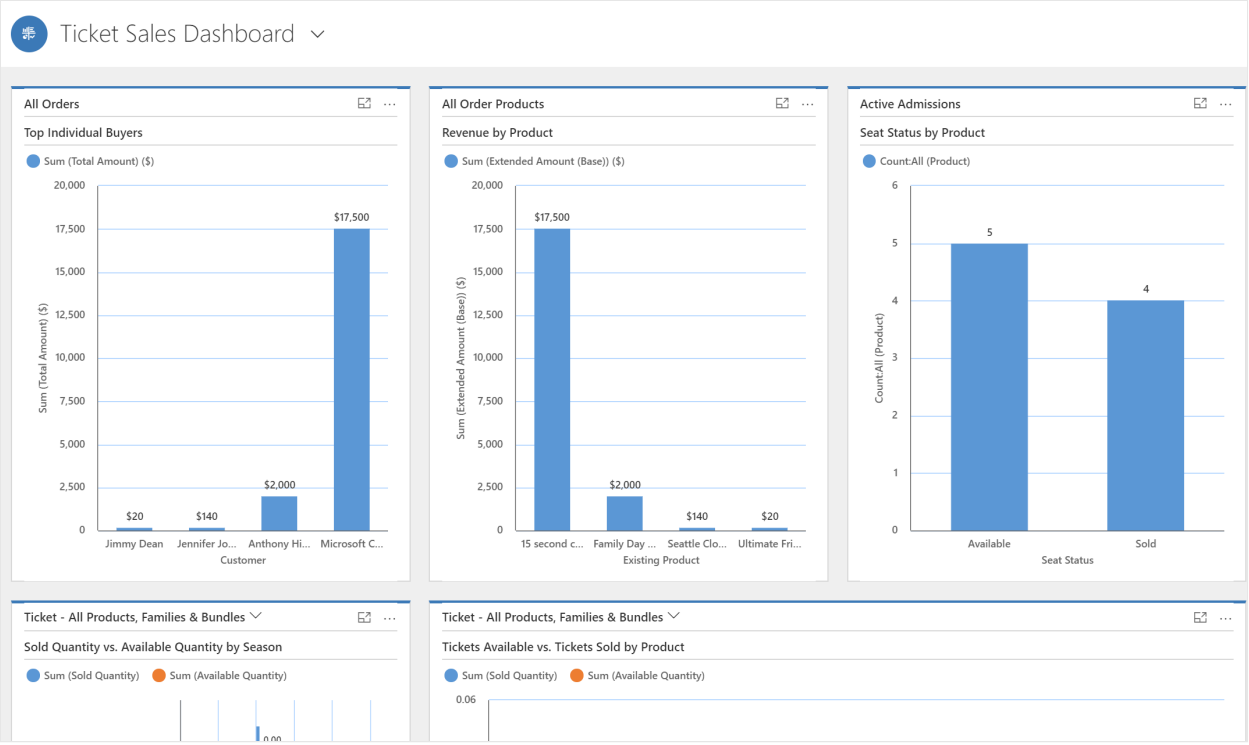
Media product sales

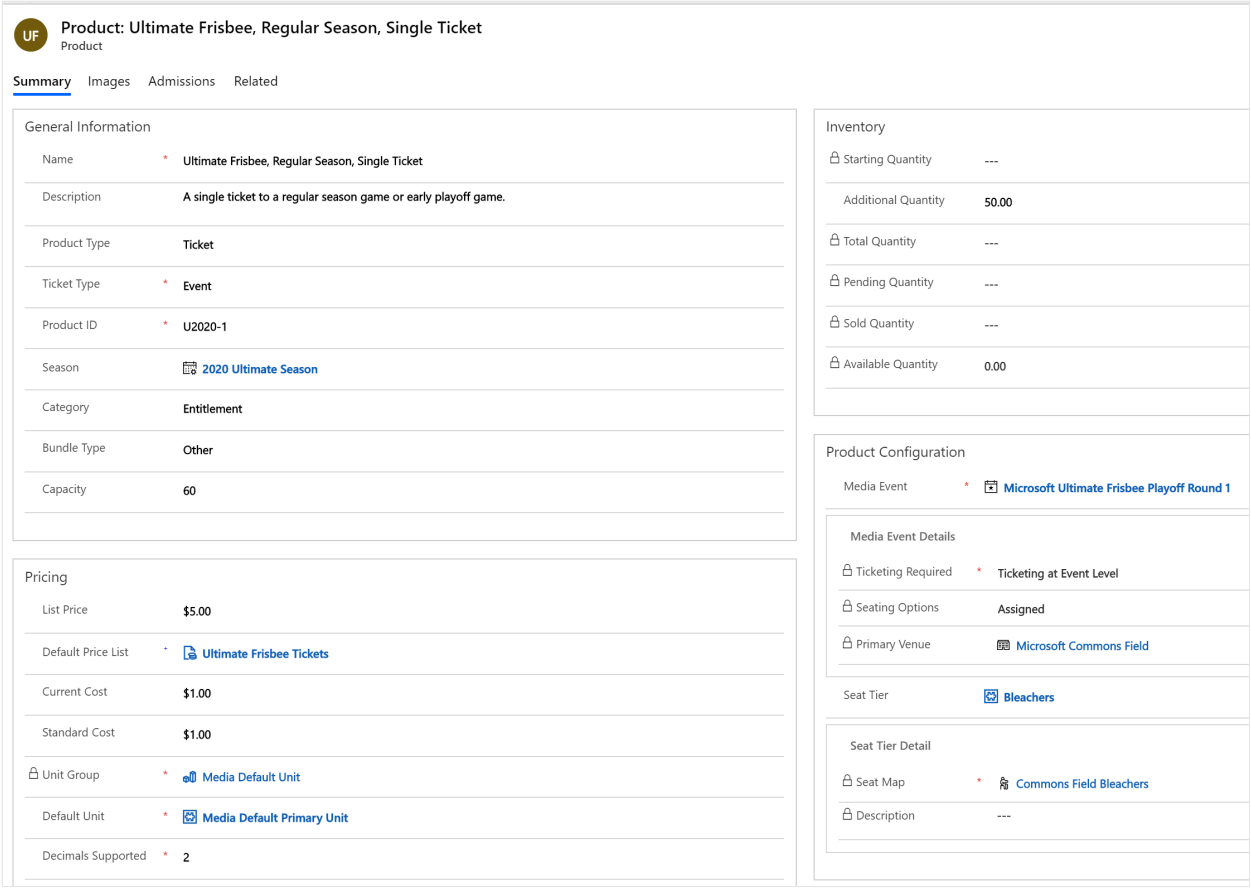
The media and entertainment accelerator can be used to easily build new product sales applications that help to promote, sell, and manage media-specific products like subscription services, ticketing, and advertising.

Ticketing and advertising sales

New ticketing data entities include the ability to build reservations and seat management solutions through seat maps. They also enable the assignment and tracking of available and blocked seat inventory used within a venue, event, or individual attraction or conference session. That in turn allows for the monetization of individual, seasonal, or group tickets to those events.

Similarly, you can build an application that enables the creation and sales of advertising display products. There's a key focus on display messaging sales at a physical venue (digital billboards, scoreboards, and other signs spread across an arena or theme park). Such an application lets you specify "what, where, and when" for display advertising, as well as to track and manage the sale and fulfillment of those advertisements and sponsorships.





Additional resources

- The media data model, solutions, data samples, Microsoft Power BI examples, SDK extensions, and more are provided as part of the open-source creative license and available on GitHub.

Create smart places by using Azure Digital Twins

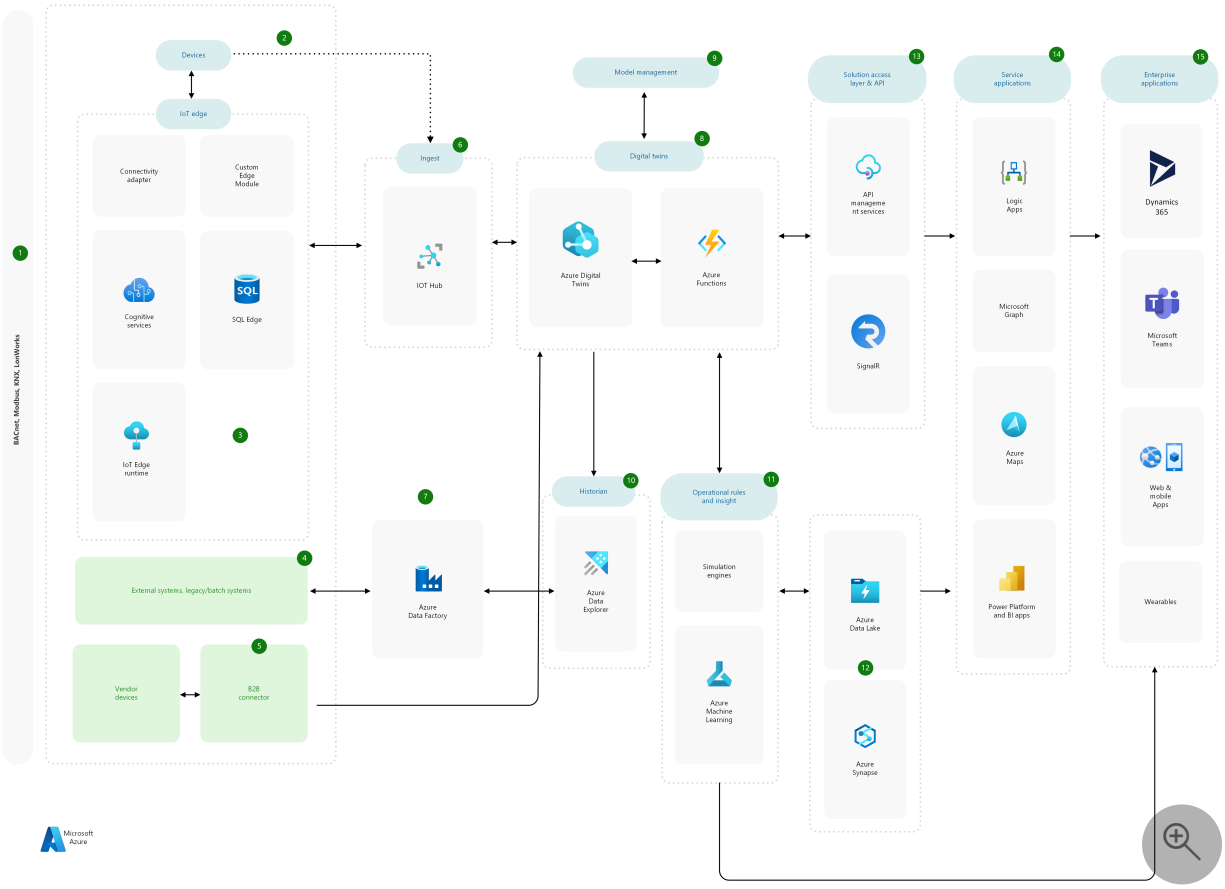
Azure Data Explorer | Azure Data Factory | Azure Digital Twins | Azure Functions | Azure IoT Hub

This article outlines a solution for smart spaces. Azure Digital Twins forms the core of the architecture by modeling the environment. Azure IoT Hub, which is a managed IoT service, also plays a significant role, as does the analytics service Azure Data Explorer.

Architecture

The following diagram shows the flow of data in this solution:

- The boxes that contain multiple icons represent categories of services. Within each category, services work independently or together to provide functionality.
- Arrows between boxes represent communication between the corresponding areas.



Download a Visio file of this architecture.

Dataflow

1. The environment can use these and other communication protocols:

    - Building Automation Controls network (BACnet)
    - Modbus
    - KNX
    - LonWorks

2. On-premises devices and systems send telemetry and other data to the cloud. Data sources include:

    - Brownfield devices
    - Direct-connect sensors
    - Sensors that independent software vendors (ISVs) provide
    - Existing business systems

3. Devices, sensors, and actuators generate telemetry. Some devices interact directly with IoT Hub. Other devices send data to IoT Hub through Azure IoT Edge.

4. External, batch, or legacy systems send data to Azure Data Factory. This static data typically originates in files and databases.

5. Business-to-business connectors translate vendor data and stream it to Azure Digital Twins.

6. IoT Hub ingests device telemetry. IoT Hub also provides these services:

    - Device-level security
    - Device provisioning services
    - Device twins
    - Command and control services
    - Scale-out capabilities

7. Data Factory transforms semi-static data and transfers it to Azure Data Explorer or to long-term storage.

8. Azure Functions receives the IoT Hub data and uses Azure Digital Twins APIs to update Azure Digital Twins. Azure Digital Twins holds the spatial graph of the buildings and environment. Azure Digital Twins models the environment with Digital Twins Definition Language (DTDL). Azure Functions processes the data, performing fault detection and graph updates.

9. Various components create, store, and load DTDL models.

10. Azure Digital Twins sends the data through Azure Event Grid to Azure Data Explorer. This analytics service functions as a historian by storing the solution's time series data.

11. Simulation engines and AI tools process the data. Examples include Azure Cognitive Services, AI models, and partner simulation services.

12. Azure Data Lake provides long-term storage for the data. Azure Synapse Analytics analyzes and reports on the data.

13. For visualization tools and enterprise apps, the solution access layer provides secure access to core system services:

    - Azure API Management offers functionality for normalizing, securing, and customizing APIs. This platform also enforces usage quotas and rate limits.

    - Azure SignalR Service sends notifications to UIs when telemetry and data changes.

    - For applications that exchange data asynchronously or at volume, various components provide publishing and subscribing mechanisms:
      - IoT Hub
      - Azure Service Bus queues
      - Azure Event Hubs
      - Web hooks

14. Service applications collect data from the access control API layer. These applications then analyze and prepare the data for end-user applications. Microsoft tools like Power Apps, Power BI, and Azure Maps create reports and insights on data in the Azure data stores.

15. Enterprise applications use the prepared data. Examples include:

    - Dynamics 365 modules.

    - ISV solutions.

    - Microsoft Teams apps.

    - Field-optimized solutions such as mobile apps and wearables:
      - HoloLens
      - RealWear HMT

Components

The solution uses these components:

Core components

- IoT Hub connects devices to Azure cloud resources. This managed service provides:
  - Device-level security.
  - Device provisioning services.
  - Device twins.
  - Command and control services.
  - Scale-out capabilities.

- Azure IoT SDKs provide the recommended way for devices to connect to IoT Hub. Devices that can use these kits include:
  - Azure Sphere devices.
  - Devices that run Azure RTOS.

- IoT Edge runs cloud workloads on IoT Edge devices. Specifically, this central message hub can run real-time analytics through Azure Machine Learning and Azure Stream Analytics. IoT Edge also functions as a gateway to IoT Hub for:
  - Devices with low-power requirements.
  - Legacy devices.
  - Constrained devices.

- Data Factory is an integration service that works with potentially large blocks of data from disparate data stores. You can use this platform to orchestrate and automate data transformation workflows. For instance, Data Factory can bridge the gap between semi-static stores and historian components like Azure Data Explorer.

- Business-to-business connectors translate and stream data bidirectionally between vendor components and Azure Digital Twins. A growing number of vendors use DTDL to create industry-standard models. RealEstateCore provides an example. As a result, these integrations are expected to become simpler over time.

- Azure Digital Twins stores digital representations of IoT devices and environments. You can use this data for data propagation or real-time analysis. Internally, Azure Digital Twins:
  - Models environments with DTDL.
  - Offers a REST API for entering data.
  - Provides SDKs that support control and data plane operations for various languages.

You can build ontologies, or pre-existing model sets, by using DTDL. You can also start with an industry-supported model:
  - RealEstateCore ontology
  - Smart Cities ontology
  - Energy Grid ontology

- Azure Digital Twins Explorer is a developer tool that you can use to visualize and interact with Azure Digital Twins data, models, and graphs. This tool is currently in public preview.

- Model management components maintain the DTDL model:

  - For model creation, these options are available:
    - Azure Digital Twins Explorer
    - ISV solutions
    - Custom-built tools
    - Text or code editors

  - Repositories store ontologies:
    - GitHub stores RealEstateCore, the Smart Cities ontology, and the Energy Grid ontology.
    - For custom ontologies, customized repositories and solution-specific repos in GitHub are available.

  - For loading models into Azure Digital Twins, these options exist:
    - UploadModels, a tool for uploading DTDL ontologies
    - Samples in the Azure Digital Twins tools repository

- Azure Functions is an event-driven serverless compute platform. With Functions, you can use triggers and bindings to integrate services at scale.

- Azure Data Explorer is a fast, fully managed data analytics service. You can use this service for real-time analysis on large volumes of data. Azure Data Explorer can handle diverse data streams from applications, websites, IoT devices, and other sources.

- Azure Cognitive Services provides AI functionality. These services offer a set of pre-trained, neural network models for the cloud. The REST APIs and client library SDKs can help you build cognitive intelligence into apps. You can use Cognitive Services functionality:
  - In near real time.
  - At certain data thresholds.
  - On demand.

    - For complex jobs with long processing times.

- Azure Machine Learning is a cloud-based environment that helps you build, deploy, and manage predictive analytics solutions. With these models, you can forecast behavior, outcomes, and trends.

- Azure Data Lake stores a large amount of data in its native, raw format. The data typically comes from multiple, heterogeneous sources and may be structured, semi-structured, or unstructured.

- Azure Synapse Analytics is an analytics service for data warehouses and big data systems. This service integrates with Power BI, Machine Learning, and other Azure services.

- 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 logs call metadata and enforces usage quotas and rate limits.

- Azure Service Bus is a fully managed enterprise message broker. Service Bus supports message queues and publish-subscribe topics.

- Azure Event Hubs is a fully managed streaming platform for big data.

- Azure SignalR Service is an open-source software library that provides a way to send notifications to web apps in real time.

Service applications

- Azure Logic Apps automates workflows by connecting apps and data across clouds.

- Azure Maps offers geospatial APIs for adding maps, spatial analytics, and mobility solutions to apps.

- Microsoft Graph provides tools for accessing data in Microsoft 365, Windows 10, and Enterprise Mobility + Security.

- Power Platform is a collection of products and services that provide low-code tools for creating efficient and flexible solutions:
    - Power Apps is a suite of apps, services, connectors, and a data platform. You can use Power Apps to transform manual business operations into digital, automated processes.

- Power BI is a collection of software services and apps that display analytics information.
- Power Automate streamlines repetitive tasks and paperless processes.
- Power Virtual Agents provides no-code chatbots to meet customer and employee needs at scale.

Enterprise applications

- Dynamics 365 is a portfolio of applications for managing business operations.
- Microsoft Teams provides services for meeting, messaging, calling, and collaborating.
- Azure App Service and its Web Apps feature provide a framework for building, deploying, and scaling web apps.

Shared support components

These services provide support for components in all areas of the solution:

- Azure Monitor collects and analyzes app telemetry, such as performance metrics and activity logs. This service notifies apps and personnel about irregular conditions.
- Microsoft Defender for IoT is a unified security service that protects IoT systems by identifying vulnerabilities and threats.
- Azure DevOps Services provides services, tools, and environments for managing coding projects and deployments.
- Microsoft Entra ID is a cloud-based identity service that controls access to Azure and other cloud apps, including ISV solutions and on-premises solutions.
- Azure Key Vault securely stores and controls access to a system's secrets, such as API keys, passwords, certificates, and cryptographic keys.

Alternatives

- Azure Cosmos DB is another option for data storage. This fully managed NoSQL database service scales easily. Azure Cosmos DB offers various ways to access data, including:
  - Document databases.
  - Graph databases.

    - SQL-style queries.
    - An Azure Cosmos DB for Apache Cassandra.

    Azure Synapse Link for Azure Cosmos DB provides a way to run analytics on Azure Cosmos DB data by using Azure Synapse Analytics. As a result, you can combine various data services in solutions that use Azure Cosmos DB.

- Event Hubs can also provide an ingestion service that's scalable and secure. Unlike IoT Hub, which supports bidirectional communication with devices, Event Hubs supports one-way traffic. As a result, you can't use Event Hubs to send commands and policies back to devices. Event Hubs also doesn't offer device-level security. But Event Hubs is appropriate for environments with a high volume of messages from a low number of input devices.

Solution details

Smart places are physical environments that bring together connected devices and data sources. By using these environments, you can see and control:

- Products and systems.
- Interior and exterior spaces.
- Personal experiences with surroundings.

Smart places can include buildings, college campuses, corporate campuses, stadiums, and cities. These environments provide value by helping property owners, facility managers, and occupants operate and maintain sites. Smart places also make spaces more efficient, cost effective, comfortable, and productive.

Smart spaces digitally model spaces and compile relevant data. From that data, you can derive insights on how people, places, and devices are connected.

Potential use cases

This solution applies to many areas:

- Smart campuses (education industry)
- Facilities management (real estate)
- Smart stadiums (sports industry)
- Smart offices
- Energy optimization

Business outcomes

In this example solution, a large commercial real estate owner is digitally transforming an office property. This improvement combines legacy facilities-management data with new features and technologies including:

- Occupancy sensing.
- Cafe queue optimization.
- Parking.
- Shuttle services.

This effort requires integrating brownfield devices and modern Internet of Things (IoT) devices that monitor the physical space. The brownfield devices communicate through common building transports such as BACnet and Modbus.

The company's goals include:

- Optimizing energy usage by diagnosing faults and streamlining field service management. This optimization integrates the existing building management system with devices.
- Deriving new spatial insights and offering innovative occupant experiences by connecting modern devices.
- Developing a cohesive digital model of the environment by bringing together multiple sources of data. The model should expand data analysis opportunities.
- Creating a scalable solution that can collect and archive millions of data points.
- Building a solution that can easily add partner solutions. The solution should also incorporate partner data into the environment's digital twin.

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 solution.

Reliability

Reliability ensures your application can meet the commitments you make to your customers. For more information, see Overview of the reliability pillar.

Scalability

Solutions for smart places solutions can be relatively simple, low-volume implementations. They can also be sophisticated implementations that handle a high volume of data. A solution that aggregates heating, ventilation, and air conditioning (HVAC) telemetry across a large campus is an example of a high-volume implementation.

The core Azure services in this solution are scalable by design and well suited for complex solutions. But when you combine these services, ensure that they don't create choke points. Early in the development cycle, run performance tests at scheduled intervals to identify potential problems.

Flexibility

Design your smart space to be well integrated but also flexible. Smart places use cases are rapidly evolving. At some point after you deploy your solution, you'll need to add new sensors, data types, AI functionality, and visualization techniques. To increase flexibility:

- Choose a loosely coupled solution like the proposed architecture.
- Use industry standards for data ontology. This approach helps reduce the time needed to add new functionality and integrate new software.
- Use API Management. This platform provides a way to create multiple API styles and signatures for a single underlying API.

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.

Legacy building solutions often rely on a lack of external connectivity as their primary source of security. But even data that doesn't identify people can provide information about a business or the people in a building. For instance, organizations use cameras to count people, track assets, and provide security data.

Be careful where you process and save images. Ensure that you address all customer requirements, including privacy issues. Make security a priority throughout the data life cycle of your smart space solution. Specifically, be aware of what data you collect, where you process and store it, and what conclusions you draw from it.

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 an IoT solution.

- For other cost considerations, see Principles of cost optimization in the Microsoft Azure Well-Architected Framework documentation.
- For a discussion about optimizing the cost of services that IoT solutions commonly use, see Azure IoT Reference Architecture.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal author:

- Mark Kottke | Senior Architect

Other contributor:

- Matthew Cosner | Principal Software Engineering Manager

Next steps

- Powering Microsoft smart buildings with Microsoft Azure Digital Twins
- Develop with Azure Digital Twins
- Brookfield sets a new standard for innovation in real estate with WillowTwin and Azure Digital Twins
- Global sustainability leader targets new heights of carbon neutrality with Azure Digital Twins

Related resources

- Get started with Azure IoT solutions
- IoT solutions conceptual overview
- Vision with Azure IoT Edge
- Azure Industrial IoT analytics guidance
- Choose an Internet of Things (IoT) solution in Azure
- End-to-end manufacturing using computer vision on the edge
- COVID-19 safe environments with IoT Edge monitoring and alerting

- IoT analytics with Azure Data Explorer
- Cognizant Safe Buildings with IoT and Azure

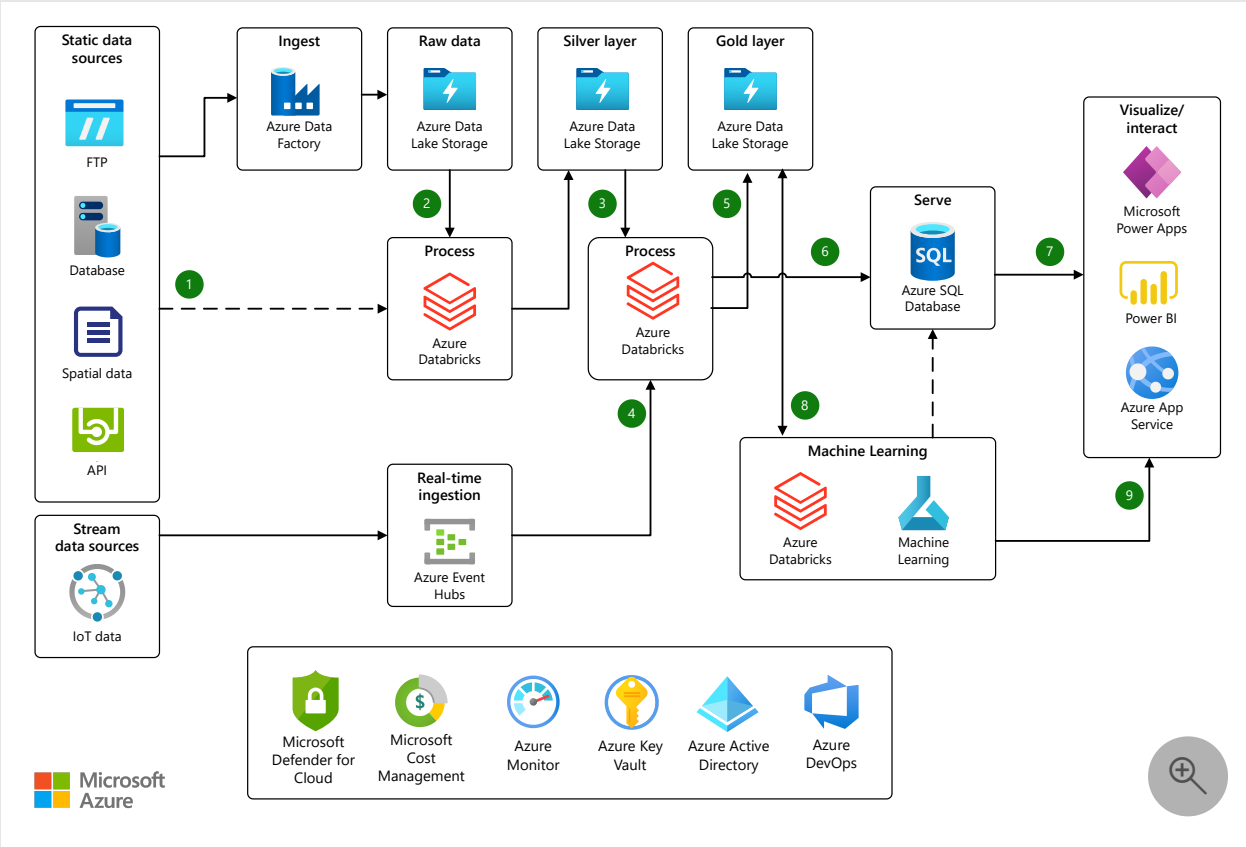
Build a sports analytics architecture on Azure

Azure Data Factory | Azure Data Lake Storage | Azure Databricks | Azure Event Hubs | Power BI

The focus of this article is to show a practical architecture that uses Azure services to process and maintain data used by sports analytics solutions. It provides a framework for sports organizations to build highly scalable solutions with, while giving them the flexibility to add more services that meet the nuanced requirements of their use cases.

Apache®, 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.

Architecture



Download a Visio file of this architecture.

Dataflow

1. Data is ingested from source systems by using one of the following methods:

    - Azure Data Factory ingests raw data from several data sources and stores it in Azure Data Lake Storage for downstream processing.
    - Some raw data sources might be large and might not need the raw data to be stored in Data Lake Storage initially, like the spatial on-court/on-field data. In these cases, you can use Azure Databricks to ingest source data and immediately transform data so that it's cleansed, normalized, and saved to Data Lake Storage in an easy-to-digest format.
    - Data that's generated by sensors in real-time is ingested as messages by Azure Event Hubs.

2. Azure Databricks transforms raw data so that it's cleansed of any errors and normalized. With the cloudFiles feature of Azure Databricks Auto Loader, raw files are automatically processed as they land in Data Lake Storage. The transformed data moves back into Data Lake Storage for further curating.

3. Azure Databricks applies business logic to the transformed data. Stream data is also combined with the transformed data during this process.

4. Azure Databricks processes stream data from Azure Event Hubs and combines it with static data.

5. The final processed data is written to Data Lake Storage in Delta format.

6. Transformed data that's used in the visualization layer, like Power BI, is written to an Azure SQL Database. This database becomes the data source for any reporting needs.

7. Curated data is visualized and manipulated through Power BI, Power Apps, or a custom web application that's hosted by an Azure App Service.

8. Azure Machine Learning builds and trains machine learning models by using data imported into Azure Machine Learning Datasets and external sources. The datasets and sources are directly linked to the Azure Machine Learning Workspace. You can control access and authentication for data and the Machine Learning workspace with Microsoft Entra ID and Azure Key Vault. Models can also be retrained as necessary in Machine Learning.

9. As an alternative to storing model results in Data Lake Storage or SQL Database, you can deploy Machine Learning models to containers using Azure Kubernetes Services (AKS) as a web service and called via a REST API endpoint. The web service deploys by using an Azure App Service, and then you can send data to the REST

API endpoint and receive the prediction returned by the model within the web application.

Throughout the process:

- Azure Monitor collects information on events and performance.
- Key Vault secures passwords, connection strings, and secrets.
- Azure DevOps manages code repositories and deployment pipelines.

Components

- Azure Data Lake Storage is a scalable and secure data lake for high-performance analytics workloads. You can use Data Lake Storage to manage petabytes of data with high throughput. It can accommodate multiple, heterogeneous sources and data that's in structured, semi-structured, or unstructured formats.
- Azure Databricks is a data analytics platform that uses Spark clusters. The clusters are optimized for the Azure platform.
- Azure Data Factory is a fully managed, scalable, and serverless data integration service. It provides a data integration and transformation layer that works with various data stores.
- Azure Machine Learning is a cloud service for accelerating and managing the machine learning project lifecycle. Machine learning professionals, data scientists, and engineers can use it in their day-to-day workflows to train, deploy models, and manage MLOps. You can create a model in Machine Learning or use a model built from an open-source platform like PyTorch, TensorFlow, or scikit-learn. MLOps tools help you monitor, retrain, and redeploy models.
- Azure Event Hubs is a big-data streaming platform and event ingestion service. It can receive and process millions of events per second. Data sent to an event hub can be transformed and stored by using any real-time analytics provider or batching/storage adapters.
- Azure SQL Database is a fully managed platform as a service (PaaS) database engine that handles most of the database management functions like upgrading, patching, backups, and monitoring without user involvement. SQL Database is always running on the latest stable version of the SQL Server database engine and patched OS with high availability.
- Power BI is a collection of software services, apps, and connectors that work together to turn your unrelated sources of data into coherent, visually immersive, and interactive insights.
- Power Apps is a suite of apps, services, connectors, and data platform that provides a rapid development environment to build custom apps for your business needs. By using Power Apps, you can quickly build custom business apps that

connect to your data stored either in the underlying data platform or in various online and on-premises data sources like SharePoint, Microsoft 365, and Dynamics 365.

- Azure App Service is an HTTP-based service for hosting web applications, REST APIs, and mobile back ends. You can use it with your favorite languages, like .NET, .NET Core, and Java.
- Microsoft Defender for Cloud is a tool for security posture management and threat protection.
- Azure Cost Management and Billing helps you understand your Azure invoice, manage your billing account and subscriptions, monitor, control Azure spending, and optimize resource use.
- Azure Monitor delivers a comprehensive solution for collecting, analyzing, and acting on telemetry from your cloud and on-premises environments.
- Azure Key Vault is a cloud service for securely storing and accessing secrets.
- Microsoft Entra ID is an identity service that provides single sign-on, multifactor authentication, and conditional access to guard against most cybersecurity attacks.
- Azure DevOps provides developer services so that teams can plan work, collaborate on code development, and build and deploy applications. Azure DevOps supports a collaborative culture and set of processes that bring together developers, project managers, and contributors to develop software.

Alternatives

- You can use Synapse Spark Pools instead of Azure Databricks for sports analytics by using the same open-source frameworks.
- Instead of Azure SQL Database, you can use Azure SQL Managed Instance to store data that's served to the visualize/interact layer.
- You can use an Azure Synapse Analytics dedicated SQL pool instead of an Azure SQL Database if the reporting requirements require several terabytes of data stored in the serving layer.
- If you don't want to use a database as the serving layer for reporting, you can choose to use a *semantic lakehouse* approach. In this scenario, reporting applications connect to logical tables that are defined by a service like Databricks SQL. These logical tables are used to structure data that's stored in the gold layer (data that's formatted using the Delta format) of Azure Data Lake Storage Gen2, so that the data can be easily read.
- Instead of Azure Databricks, you can use SQL Database or SQL Managed Instance to query and process data. These databases provide the familiar T-SQL language, which you can use for analysis.

- You can use Azure Stream Analytics instead of Azure Databricks to process stream data.
- You can use Azure Machine Learning instead of Azure Databricks to train your machine learning models.
- You can use GitHub instead of Azure DevOps to manage your code repositories and continuous integration and continuous delivery (CI/CD) pipelines.

Scenario details

Sports analytics is a field that applies data analytics techniques to team or individual performance data. Then you can use the data to create a competitive advantage over an opponent. In addition to analyzing traditional box score statistics, there has been an explosion of data in recent years that sports teams can use to improve the performance of an individual athlete or an entire team. Examples of such data include player data collected from sensors and spatial data that captures player movement during a game. Traditional systems struggle to process and maintain these data sources because of the large volumes of data generated. These data sources also format data in several different ways and allow users to process data at different speeds, providing more challenges for traditional data processing solutions.

Potential use cases

This solution is ideal for the sports industry, and applies to the following scenarios:

- Manage large volumes of data from several source systems in a centralized ecosystem.
- Analyze player tracking and temporal data to gain insights into individual and team performance.
- With consideration for spatial metrics, determine the best possible player positioning and strategies during gameplay.
- Process and evaluate player performance data to optimize athlete training routines.
- Analyze historical data to make well-informed personnel decisions during the draft or free agency.
- Store and analyze real-time telemetry from Internet of Things (IoT) devices that are attached to equipment like bats, shoulder pads, and volleyballs.

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.

Follow MLOps guidelines to standardize and manage end-to-end Machine Learning lifecycles that are scalable across multiple workspaces. Before going into production, ensure the implemented solution supports ongoing inference with retraining cycles and automated redeployments of models.

Use the Azure/mlops-v2 GitHub repository as an MLOps resource.

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.

Consider using the following security resources in this architecture:

- Secure cluster connectivity (No Public IP / NPIP)
- Store credentials in Azure Key Vault
- Deploy dedicated Azure services into virtual networks
- Azure Databricks Premium
- Enterprise security for Azure Databricks

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, use the Azure pricing calculator for the services mentioned above.
- Power BI comes with different licensing offerings. For more information, see Power BI pricing .
- Depending on the volume of data and complexity of your geospatial analysis, you might need to scale your Databricks cluster configurations that affect your cost. Refer to the Databricks cluster sizing examples for best practices on cluster configuration.

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.

If you use Azure Data Factory Mapping Data Flows for extract, transform, and load (ETL), follow the performance and tuning guide for mapping data flows. Mapping data flows this way optimizes your data pipeline and ensures that your data flows meet your performance benchmarks.

Deploy this scenario

To deploy this scenario, follow the steps described in this Azure quickstart, Deploy the Sports Analytics on Azure Architecture. Be sure to read the **Prerequisites** section in the quickstart before deploying the solution.

Contributors

This article is maintained by Microsoft. It was originally written by the following contributors.

Principal authors:

- Giulia Gallo | Senior Cloud Solution Architect
- Jake Switzer | Sports Analytics Solution Architect
- Tash Tahir | Principal Cloud Solution Architect

Other contributor:

- Jason Martinez | Technical Writer

Next steps

- App Service overview
- Azure Event Hubs — A big data streaming platform and event ingestion service
- Azure security baseline for Azure Machine Learning
- Consume an Azure Machine Learning model deployed as a web service
- Data integration at scale with Azure Data Factory or Azure Synapse Pipeline
- Data engineering with Azure Databricks
- Introduction to Azure Data Lake Storage Gen2
- What is Azure Machine Learning?
- What is Azure SQL Database?

- What is Power BI?
- What is Power Apps?

Related resources

- IoT and data analytics
- Monitoring Azure Databricks
- Stream processing with Azure Databricks

Live stream digital media

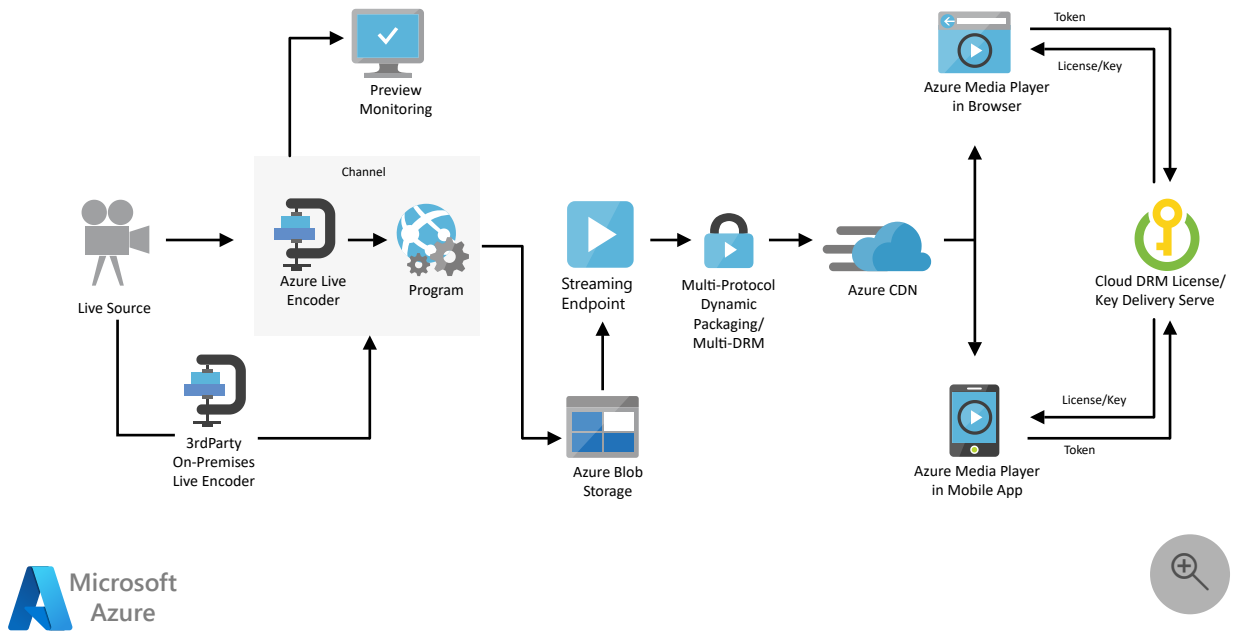
Azure Blob Storage | Azure Content Delivery Network | Azure Media Player | Azure Media Services

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 is built on the Azure managed service: Media Services and Content Delivery Network. These services run in a high-availability environment, patched and supported, allowing you to focus on your solution instead of the environment they run in.

Architecture



Download a Visio file of this architecture.

Components

- Partner on-premises live encoder: Outputs the live source for ingest into the cloud as RTMP(S), or Smooth Streaming.

- Stores large amounts of unstructured data that can be accessed from anywhere in the world via HTTP or HTTPS. You can use Blob storage to expose data publicly to the world, or to store application data privately.
- Media Services: Ingest, encode, preview, store, and deliver your live streaming content. Live Events, Live Outputs, and Streaming Endpoints handle the live streaming functions, including ingestion, formatting, DVR, security, scalability, and redundancy.
- Media Services Streaming Endpoint: Represents a streaming service that can deliver content directly to a client player application, or to a content delivery network (CDN) for further distribution.
- Content Delivery Network: Provides secure, reliable content delivery with broad global reach and a rich feature set.
- Azure Media Player: Uses industry standards such as HTML5 (MSE/EME) to provide an enriched adaptive streaming experience. Regardless of the playback technology used, developers have a unified JavaScript interface to access APIs.
- Preview monitoring: Preview and validate a live stream before further processing and delivery.
- Multi-DRM content protection: Delivers content securely using multi-DRM (PlayReady, Widevine, FairPlay Streaming) or AES clear key encryption.

Solution details

A live streaming solution allows you to capture video in real-time and broadcast it to consumers in real-time. This can include streaming interviews, conferences, and sporting events online. In this solution, video is captured by a video camera and sent to a Live Event input endpoint. The Live Event receives the input stream and makes it available for streaming through a Streaming Endpoint to a web browser or mobile app. The Live Event also provides a preview monitoring endpoint to preview and validate your stream before further processing and delivery. The Live Event can also record and store the ingested content in order to be streamed later (video-on-demand).

Potential use cases

This solution applies to anyone from large corporations to small content creators.

Next steps

- Overview of Media Services live transcoding
- How to use Azure Blob storage
- Overview of Media Services live streaming

- Overview of Content Protection
- Using Azure Content Delivery Network
- Azure Media Services documentation
- Media services content protection

Related resources

- Performance tuning - Event streaming
- HPC media rendering
- Test Media Services V3 encoding
- Content Delivery Network analytics

Video-on-demand digital media

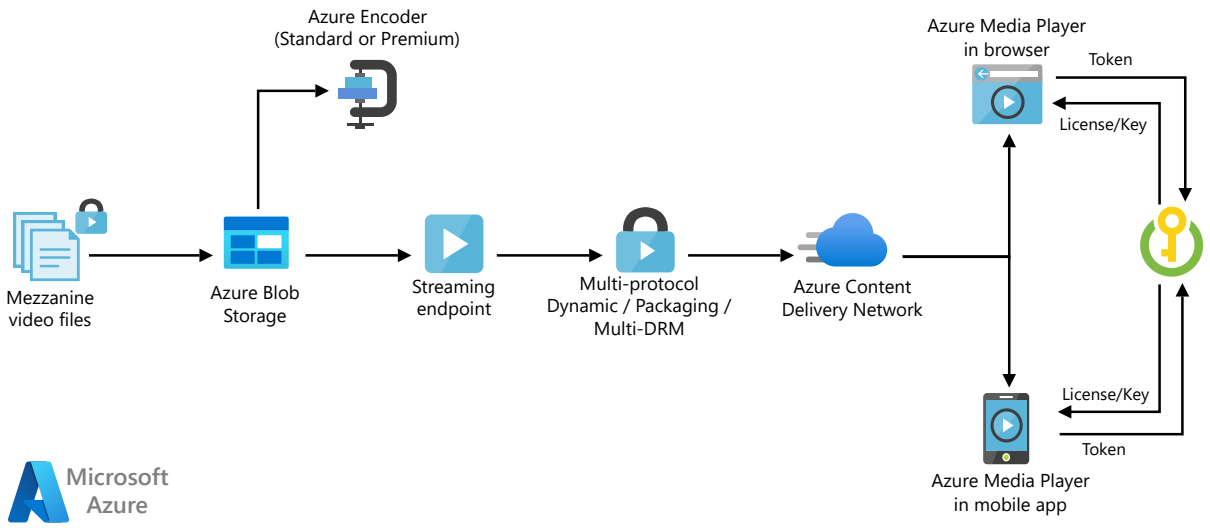
Azure Blob Storage | Azure Content Delivery Network | Azure | Azure Media Player

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 is built on the Azure managed services: Blob Storage, Content Delivery Network, and Azure Media Player from Azure Media Services. These services run in a high-availability environment, patched and supported, allowing you to focus on your solution instead of the environment they run in.

Architecture



Download a Visio file of this architecture.

Components

- Blob Storage: Stores large amounts of unstructured data that can be accessed from anywhere in the world via HTTP or HTTPS. You can use Blob storage to expose data publicly to the world, or to store application data privately. There are multiple options for uploading files to blob storage, including AzCopy, Media

Services Azure portal, .NET SDK, or REST API, Azure CLI, Python, or one of several Azure blob storage tools/SDKs.

- Azure Media Services Encoder: Encoding jobs are one of the most common processing operations in Media Services. You create encoding jobs to convert media files from one encoding to another.
- Azure Media Services Streaming Endpoint: A streaming service that can deliver content directly to a client player application, or to a content delivery network (CDN) for further distribution.
- Content Delivery Network: Provides secure, reliable content delivery with broad global reach and a rich feature set.
- Azure Media Player: Uses industry standards, such as HTML5 (MSE/EME), to provide a rich adaptive streaming experience. Regardless of the playback technology used, developers have a unified JavaScript interface to access APIs. Also, see the Azure Media Player documentation.
- Multi-DRM content protection: Delivers content securely using multi-DRM (PlayReady, Widevine, FairPlay Streaming) or AES Clear Key encryption.

Scenario details

A basic video-on-demand solution that gives you the capability to stream recorded video content to any video-capable endpoint device, mobile application, or desktop browser. This content might include movies, news clips, sports segments, training videos, and customer support tutorials. Video files are uploaded to Azure Blob storage, encoded to a multi-bitrate standard format, and then distributed via all major adaptive bit-rate streaming protocols (HLS, MPEG-DASH, Smooth) to the Azure Media Player client.

You can also use other media players such as:

- Video.js
- Shaka Player
- hls.js
- dash.js
- ExoPlayer
- AVPlayer
- THEOplayer
- NexPlayer

Potential use cases

This solution applies to television, movie, and various online streaming services.

Next steps

- To get started with Azure Media Services, visit the Azure Media Services documentation where you will find quickstarts, tutorials, and samples.
- Azure Media Player overview
- How to use Azure Blob storage
- How to encode an asset using Media Encoder
- How to manage streaming endpoints
- Using Azure Content Delivery Network
- Playing your content with existing players
- Deliver content securely

Related resources

- Live stream digital media
- Gridwich cloud media system
- Instant broadcasting with serverless code

Industry specific Azure IoT reference architectures

Article • 01/11/2023

This article provides an overview of industry-specific internet-of-things (IoT) reference architectures and example solutions.

The following Azure IoT solutions and example scenarios address unique business challenges for specific industries. For overall IoT patterns and guides that apply across industry verticals, see Get started with Azure IoT solutions.

Automotive and transportation

- Process real-time vehicle data using IoT. Vehicle data ingestion, processing, and visualization are key capabilities of connected car solutions. Capturing and analyzing real-time vehicle data provides valuable insights to create new solutions.
- Real-time asset tracking for vehicles. Azure IoT Central and other Azure services track and manage vehicles and other assets in real time.
- Railway health system with IoT Edge. Edge computing enables fast, consistent responses with reduced dependency on cloud connectivity and resources. The intelligent edge brings data processing and storage close to the data source in this train maintenance and safety solution.
- Automated guided vehicles fleet control. Automated guided vehicles (AGVs) are an important part of just-in-time manufacturing and automated shop-floor logistics. In this example, AGVs deliver parts to automotive assembly lines.

Energy and environment

- Environmental monitoring and supply chain optimization with IoT. Environmental monitoring is crucial for global supply chain management. A warehouse management scenario monitors environmental conditions and processes the data with machine learning (ML) to generate predictions.
- Project 15 Open Platform IoT sustainability. Open Platform open-source software connects to the cloud and securely manages devices for scientific and conservation projects. This architecture is a reference for building open-source, end-to-end IoT solutions.

Facilities and real estate

- Create smart places by using Azure Digital Twins. Smart places are physical environments, like buildings, campuses, and cities, that bring together connected devices and data sources. Azure Digital Twins stores digital representations of physical environments to use for monitoring, analysis, and management.
- Safe environments with IoT Edge monitoring and alerting. Combining existing closed-circuit TV infrastructure with the Azure intelligent edge and other Azure and Microsoft services helps organizations monitor, follow, and improve health and safety practices.

Manufacturing

- Connected factory hierarchy service. A hierarchy service lets businesses centrally define how to organize production assets within factories.
- Connected factory signal pipeline. The signal pipeline uses a common configuration interface based on the Open Platform Communications Unified Architecture (OPC UA) to connect heterogenous legacy and modern devices.
- Condition monitoring for industrial IoT. Condition monitoring helps manufacturers discover anomalies before they become critical. In this solution, IoT devices connect to the cloud through OPC UA and Azure industrial IoT components.
- Predictive maintenance for industrial IoT. Predictive maintenance diagnoses and predicts malfunctions and maintenance needs in OPC UA connected equipment. This solution uses mixed reality and digital twins technologies to optimize production in real time.
- End-to-end computer vision at the edge for manufacturing. Industries use computer vision and ML for safety and quality assurance applications. This example shows an end-to-end approach to IoT computer vision that improves processes over time.

Retail

- Video capture and analytics for retail. Retailers like grocery stores can monitor storefront events and take immediate actions to improve customer experience. On-premises IoT Edge devices analyze video data in real time to detect and address issues like empty shelf space or long customer queues.

- Buy online, pickup in store (BOPIS). Azure IoT and cloud components combine to implement a curbside pickup, or buy online and pick up in store (BOPIS), system for retail.

Next steps

- What is Azure IIoT? Cloud platform-as-a-service (PaaS) components use device data to develop business and industrial solutions. For guidance in using Azure PaaS components to create IIoT solutions, see the series on Industrial IoT analytics.

Microsoft Cloud Adoption Framework for Azure

Proven guidance and best practices that help you confidently adopt the cloud and achieve business outcomes.

OVERVIEW
Explore scenarios

DOWNLOAD
Get tools and templates

WHAT'S NEW
Find out what's new

OVERVIEW
Learn about the Framework

Cloud Adoption Framework guidance

Find guidance for each phase of your cloud adoption journey.

Get started

- Get started
- Accelerate migration
- Deliver operational excellence
- Antipatterns to avoid

Strategy

- Motivations
- Business outcomes
- Financial considerations
- Technical considerations

Plan

- Rationalize your digital estate
- Organizational alignment
- Skills readiness plan
- DevOps cloud adoption plan

Ready

Migrate

Innovate

Operating model alignment

Azure landing zone conceptual architecture

Azure landing zone design areas

Implementation options

Azure migration guide

Migration scenarios

Cloud migration best practices

Process improvements

Business value consensus

Build your first MVP

Measure for customer impact

Expand digital inventions

Secure

Overview

Risk insights

Business resilience

Asset protection

Manage

Business commitments

Management baseline

Expand the baseline

Advance operations and design principles

Govern

Methodology

Benchmark assessment

Governance foundation

Improve the foundation

Assessments

Plan your cloud journey with assessments that help you measure the difference between your current state and cloud adoption goals. Assessment results provide curated technical guidance to help you reach your goals.

Cloud Adoption Strategy Evaluator

Build and advance your cloud business case with the cloud adoption strategy assessment.

App and Data Modernization Readiness

Take the first step in modernizing your workloads by taking this application and data modernization assessment.

Strategic Migration Assessment and Readiness

Drive business success by unblocking migration efforts and improving processes.

Governance benchmark

Identify gaps between your current state and business priorities. Get the right resources to help you address the gaps.

Build your skills with Microsoft Learn training

Learn more about Azure and the Cloud Adoption Framework by completing these learning paths.

Microsoft Cloud Adoption Framework for...

Azure Fundamentals

Learn the business value of Azure

Find help

Microsoft has dedicated resources to help unblock your cloud adoption journey.

FastTrack for Azure

Get support from Microsoft engineers

Azure Migrate and Modernize

Get support migrating workloads to the cloud

Partner-led Workshops

Obtain best practices on Cloud Adoption Framework from certified partners

Find a partner

Find a partner to help you get started

Why the Microsoft Cloud Adoption Framework for Azure is right for your business. https://aka.ms/adopt