Classified as Microsoft Confidential 1



**Microsoft Azure Cosmos DB IoT Solution Accelerator**

Version 1.0  
December, 2019

Contents

[About the solution accelerator 4](#_Toc28030814)

[High-level concepts 5](#_Toc28030815)

[Non-relational data and NoSQL 5](#_Toc28030816)

[Event sourcing pattern 7](#_Toc28030817)

[Serverless and no/low code processing 8](#_Toc28030818)

[IoT reference architecture 9](#_Toc28030819)

[High-level architecture for the solution accelerator 15](#_Toc28030820)

[Adapting the sample scenario to your own 15](#_Toc28030821)

[Architecture components 15](#_Toc28030822)

[Detailed walk-through of the solution accelerator components 17](#_Toc28030823)

[List of deployed Azure resources 17](#_Toc28030824)

[Azure Cosmos DB: Data is at the core 21](#_Toc28030825)

[Cloud gateway: Managing and devices and ingesting telemetry with IoT Hub 26](#_Toc28030826)

[Stream processing, event sourcing, and data management 34](#_Toc28030827)

[Machine Learning and advanced analytics 44](#_Toc28030828)

[Additional references 52](#_Toc28030829)

[Appendix A: What is Azure Machine Learning? 52](#_Toc28030830)

[Appendix B: Logic Apps 59](#_Toc28030831)

[Appendix C: IoT device and solution design considerations 66](#_Toc28030832)

The information herein is for informational purposes only and represents the current view of Microsoft Corporation as of the date of this publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information provided after the date of this presentation. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS PRESENTATION.

© 2019 Microsoft. All rights reserved. This document is for informational purposes only. Microsoft makes no warranties, express or implied, with respect to the information presented here

# About the solution accelerator

The primary goal of the Microsoft Azure Cosmos DB IoT Solution Accelerator is to provide guidelines for building end-to-end IoT solutions in Azure. We provide a sample client scenario and starter artifacts to demonstrate these guidelines and to provide a contextual foundation for covering architectural concepts and diving into the technical aspects of the tools and services.

When you start to design an end-to-end solution in the cloud, it is easy to become overwhelmed as to where to start. There are many services involved in the reference architecture and many of them have overlapping capabilities. There are several decision points along the way about core technologies, such as data storage and processing, as well as architecture and development patterns, like event sourcing and microservices. These challenges are why we created the solution accelerator. We want to provide a starting point for learning about creating an IoT-based solution in Azure, as well as a set of tools and starter artifacts you can use to accelerate building your own solutions.

This document is just one part of the solution accelerator, which you can find on GitHub: <https://github.com/solliancenet/cosmos-db-iot-solution-accelerator>. You can print this document or save it to your computer or mobile device to use it as a reference for offline viewing. It is a companion to the online Quickstart guide (<https://github.com/solliancenet/cosmos-db-iot-solution-accelerator/blob/master/Quickstart.md>) that walks you through an easy to follow, step-by-step process to deploy and configure the reference solution. Within the Quickstart guide, we introduce each Azure service and where they fit into the architecture as a whole within the context of the deployment and configuration process. Where appropriate, the Quickstart guide covers important details about service interactions and source code so you can understand the inner workings of the solution’s components, as well as how you can make modifications to adapt the artifacts to your scenario. This document combines those details and expands on them as needed while removing the steps-by-step elements that the Quickstart guide uses to walk you through setting up the solution.

# High-level concepts

In this document, we cover the high-level concepts needed to understand the components of the reference architecture. Although Internet-of-Things (IoT) is the primary theme of this solution accelerator, *the concepts beyond the IoT devices and cloud gateway (IoT Hub) apply to many other, event-oriented scenarios*.

## Non-relational data and NoSQL

A *non-relational database* is a database that does not use the tabular schema of rows and columns found in most traditional database systems. Instead, non-relational databases use a storage model that is optimized for the specific requirements of the type of data being stored. For example, data may be stored as simple key/value pairs, as JSON documents, or as a graph consisting of edges and vertices.

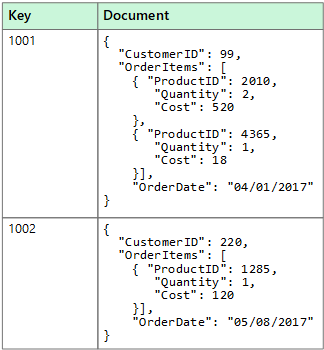
What all of these data stores have in common is that they don't use a relational model. Also, they tend to be more specific in the type of data they support and how data can be queried. For example, time-series data stores are optimized for queries over time-based sequences of data, while graph data stores are optimized for exploring weighted relationships between entities. Neither format would generalize well to the task of managing transactional data.

The term *NoSQL* refers to data stores that do not use SQL for queries, and instead use other programming languages and constructs to query the data. In practice, "NoSQL" means "non-relational database," even though many of these databases do support SQL-compatible queries. However, the underlying query execution strategy is usually very different from the way a traditional RDBMS would execute the same SQL query.

### Document data stores

A document data store manages a set of named string fields and object data values in an entity referred to as a *document*. These data stores typically store data in the form of JSON documents. Each field value could be a scalar item, such as a number, or a compound element, such as a list or a parent-child collection. The data in the fields of a document can be encoded in a variety of ways, including XML, YAML, JSON, BSON, or even stored as plain text. The fields within documents are exposed to the storage management system, enabling an application to query and filter data by using the values in these fields.

Typically, a document contains the entire data for an entity. What items constitute an entity are application-specific. For example, an entity could contain the details of a customer, an order, or a combination of both. A single document might contain information that would be spread across several relational tables in a relational database management system (RDBMS). A document store does not require that all documents have the same structure. This free-form approach provides a great deal of flexibility. For example, applications can store different data in documents in response to a change in business requirements.



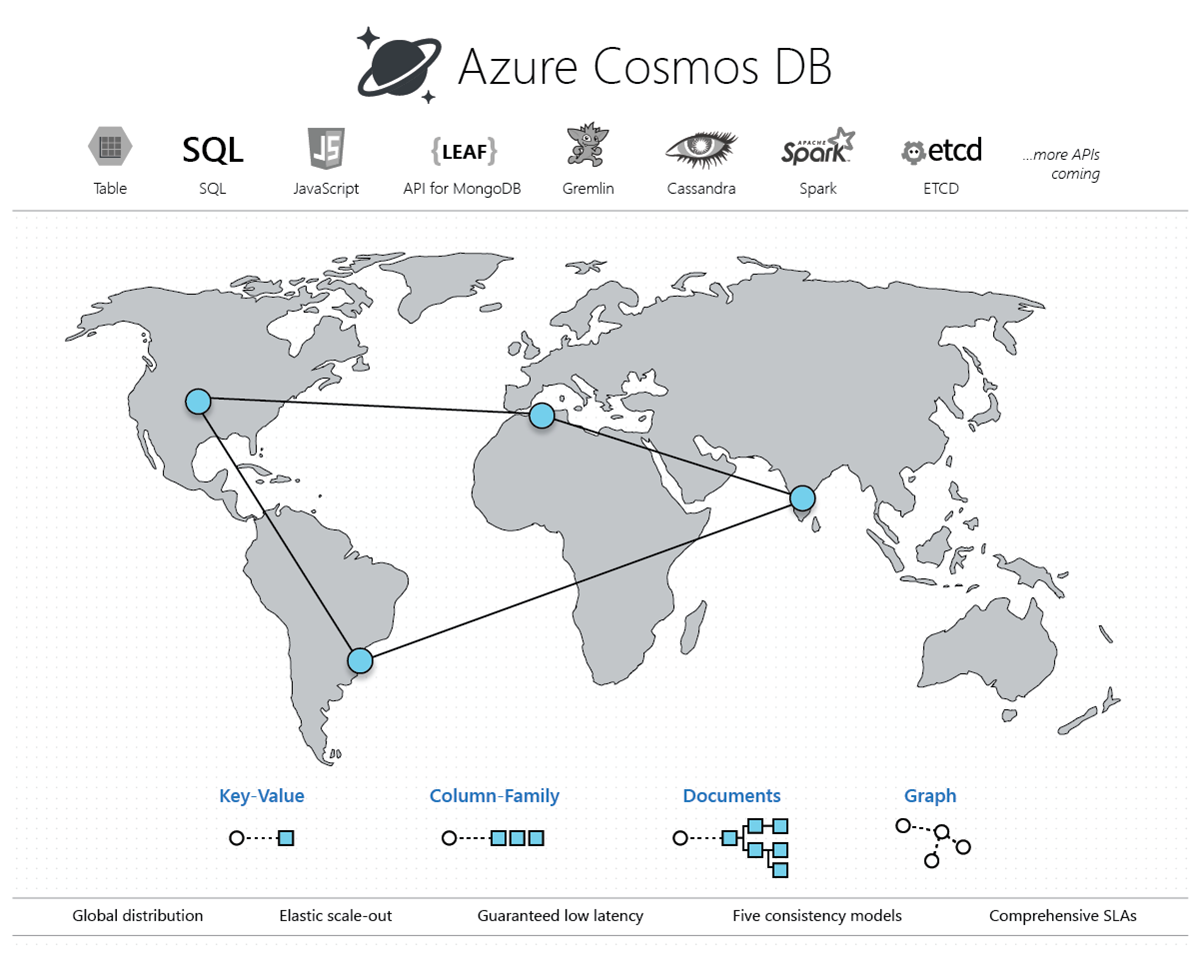
The application can retrieve documents by using the document key. This key is a unique identifier for the document, which is often hashed, to help distribute data evenly. Some document databases create the document key automatically. Others enable you to specify an attribute of the document to use as the key. The application can also query documents based on the value of one or more fields. Some document databases support indexing to facilitate a fast lookup of documents based on one or more indexed fields.

Many document databases support in-place updates, enabling an application to modify the values of specific fields in a document without rewriting the entire document. Read and write operations over multiple fields in a single document are typically atomic.

### Cosmos DB - managed NoSQL database on Azure

[Azure Cosmos DB](https://docs.microsoft.com/azure/cosmos-db/introduction) is Microsoft's globally distributed, multi-model database service. With a click of a button, Cosmos DB enables you to elastically and independently scale throughput and storage across any number of Azure regions worldwide. You can elastically scale throughput and storage and take advantage of fast, single-digit-millisecond data access using your favorite API, including SQL, MongoDB, Cassandra, Tables, or Gremlin. Cosmos DB provides comprehensive [service level agreements](https://aka.ms/acdbsla) (SLAs) for throughput, latency, availability, and consistency guarantees, something no other database service offers.

The solution accelerator uses Cosmos DB as its non-relational data store, along with the SQL API to enable the document data store and SQL query constructs. The high-throughput, high-availability, and low-latency characteristics of Cosmos DB, coupled with its [change feed](https://docs.microsoft.com/azure/cosmos-db/change-feed) feature, makes it an ideal data store for IoT telemetry, operational data, and materialized views while enabling the powerful flexibility and capabilities the event sourcing pattern provides.



## Event sourcing pattern

A vital pattern used in the solution accelerator is the [event sourcing pattern](https://docs.microsoft.com/azure/architecture/patterns/event-sourcing). This pattern defines an approach to handling operations on data that's driven by a sequence of events, each of which is recorded in an append-only store. In our implementation, IoT devices send telemetry as a series of events that imperatively describe the state of each device over time to the event store, where they're persisted. Each event represents a set of changes to the data, which is tied back to the source IoT device.

The event store acts as the system of record (the authoritative data source) about the current state of the data. The event store used in the solution accelerator is an [Azure Cosmos DB](https://docs.microsoft.com/azure/cosmos-db/introduction) container that is tuned for write-heavy workloads through minimal indexing, partitioning on a key with high cardinality, and by setting a throughput adjusted for a high rate of ingesting (more on these concepts later). The Cosmos DB [change feed](https://docs.microsoft.com/azure/cosmos-db/change-feed) is used to publish these events so that consumers are notified so they can handle them if needed.

Typical uses of the events published by the change feed are to maintain [materialized views](https://docs.microsoft.com/azure/architecture/patterns/materialized-view) of entities as telemetry is ingested or actions in the application change them, and for integration with external systems. For example, as device telemetry is saved, materialized views are updated with aggregated information about the IoT device telemetry, which is used to populate parts of the UI such as dashboards and reports. The aggregated data in this example is saved to a different container in Cosmos DB, eliminating the need to query against the event collection and perform expensive aggregates across multiple partitions. Other aggregated data is sent to Power BI to update a real-time dashboard to display the overall state of the IoT devices.

Implementing the event sourcing pattern allows data and software architects to think beyond typical CRUD operations that may be used to for their databases and applications. The components of the event sourcing pattern are loosely coupled and can often operate in parallel for maximum scalability. This pattern helps these architects consider how they can handle the rising velocity, variety, and volume of data in today's Big Data landscape.

## Serverless and no/low code processing

Serverless computing enables developers to build applications faster by eliminating the need for them to manage infrastructure. With serverless applications, the cloud service provider automatically provisions, scales, and manages the infrastructure required to run the code. Frequently, serverless computing is associated with consumption-based pricing, which means that you only pay for compute when needed without paying for resources you do not use.

In understanding the definition of serverless computing, it's important to note that servers are still running the code. The serverless name comes from the fact that the tasks associated with infrastructure provisioning and management are invisible to the developer. This approach enables developers to increase their focus on business logic and deliver more value to the core of the business. Serverless computing helps teams increase their productivity and bring products to market faster, and it allows organizations to optimize resources better and stay focused on innovation.

In this solution accelerator, we use [Azure Functions](https://docs.microsoft.com/azure/azure-functions/functions-overview) as a natural fit for the event-driven processing enabled by the IoT Hub and the Cosmos DB change feed. The consumption pricing plan provides a pay-per-execution model with sub-second billing that charges only for the time and resources it takes to execute the code. The first 1 million requests being free each month. Functions support bindings that make it easy to integrate with services such as IoT Hub, Cosmos DB, and Event Hubs. The functions manage connectivity to these services, including the lifecycle of the client components used to communicate with them. These bindings significantly reduce the amount of code required to create the functions, allowing developers to focus more on application logic and less on the plumbing code.

For low-code processing, we use [Azure Stream Analytics](https://docs.microsoft.com/azure/stream-analytics/stream-analytics-introduction) to analyze and process high volumes of event data from multiple sources simultaneously. Stream Analytics connects to inputs, such as IoT Hub and Event Hubs, and several outputs it can use as data sinks, including Cosmos DB, Power BI, and several other Azure services. It provides a SQL-like query language used to query over the incoming data, where you can easily adjust the event ordering options and duration of time windows when performing aggregation operations through simple language constructs or configurations. We use Stream Analytics in this solution accelerator to aggregate data over time windows of varying sizes. We use these aggregates to populate materialized views in Cosmos DB and to send small aggregates of data directly to Power BI to update a near real-time dashboard.

The service that provides no-code processing in the solution accelerator is [Azure Logic Apps](https://docs.microsoft.com/azure/logic-apps/logic-apps-overview). This service works as a powerful workflow orchestrator that natively [integrates with hundreds](https://docs.microsoft.com/azure/connectors/apis-list) of Azure and 3rd-party services. Users usually build logic apps with the Logic Apps Designer, which provides a simple web-based, drag-and-drop interface. Alternately, logic apps can be built using JavaScript Object Notation (JSON) for scripting, Azure PowerShell commands, and Azure Resource Manager (ARM) templates. We create a logic app in the solution accelerator to send email notifications to recipients when certain event milestones occur, such as when a package delivery is running behind schedule, or when an oil pump encounters an anomaly.

## IoT reference architecture

The IoT reference architecture has been adapted and slightly modified from the [source](https://docs.microsoft.com/azure/architecture/reference-architectures/iot/).

IoT applications can be described as **things** (devices) sending data that generate **insights**. These insights generate **actions** to improve a business or process. An example is an engine (the thing) sending temperature data. This data is used to evaluate whether the engine is performing as expected (the insight). The insight is used to proactively prioritize the maintenance schedule for the engine (the action).

This reference architecture uses Azure PaaS (platform-as-a-service) components. Other options for building IoT solutions on Azure include:

* [Azure IoT Central](https://docs.microsoft.com/azure/iot-central/). IoT Central is a fully managed SaaS (software-as-a-service) solution. It abstracts the technical choices and lets you focus on your solution exclusively. This simplicity comes with a tradeoff in being less customizable than a PaaS-based solution.
* Using OSS components such as the SMACK stack (Spark, Mesos, Akka, Cassandra, Kafka) deployed on Azure VMs. This approach offers a great deal of control but is more complex.

At a high level, there are two ways to process telemetry data, hot path, and cold path. The difference has to do with requirements for latency and data access.

* The **hot path** analyzes data in near-real-time, as it arrives. In the hot path, telemetry must be processed with very low latency. The hot path is typically implemented using a stream processing engine. The output may trigger an alert or be written to a structured format that can be queried using analytical tools.
* The **cold path** performs batch processing at longer intervals (hourly or daily). The cold path typically operates over large volumes of data, but the results don't need to be as timely as the hot path. In the cold path, raw telemetry is captured and then fed into a batch process.

### Reference architecture components

This architecture consists of the following components. Some applications may not require every component listed here.

**IoT devices**. Devices can securely register with the cloud and can connect to the cloud to send and receive data. Some devices may be **edge devices** that perform some data processing on the IoT device itself or in a field gateway. We recommend [Azure IoT Edge](https://docs.microsoft.com/azure/iot-edge/) for edge processing.

**Cloud gateway**. A cloud gateway provides a cloud hub for devices to connect securely to the cloud and send data. It also provides device management, capabilities, including command and control of devices. For the cloud gateway, we recommend [IoT Hub](https://docs.microsoft.com/azure/iot-hub/). IoT Hub is a hosted cloud service that ingests events from devices, acting as a message broker between devices and backend services. IoT Hub provides secure connectivity, event ingestion, bidirectional communication, and device management.

**Device provisioning.** For registering and connecting large sets of devices, we recommend using the [IoT Hub Device Provisioning Service](https://docs.microsoft.com/azure/iot-dps/) (DPS). DPS lets you assign and register devices to specific Azure IoT Hub endpoints at scale.

**Stream processing**. Stream processing analyzes large streams of data records and evaluates rules for those streams. For stream processing, we recommend [Azure Stream Analytics](https://docs.microsoft.com/azure/stream-analytics/). Stream Analytics can execute complex analysis at scale, using time windowing functions, stream aggregations, and external data source joins. Another option is Apache Spark on [Azure Databricks](https://docs.microsoft.com/azure/azure-databricks/).

**Machine learning** allows predictive algorithms to be executed over historical telemetry data, enabling scenarios such as predictive maintenance. For machine learning, we recommend [Azure Machine Learning](https://docs.microsoft.com/azure/machine-learning/service/).

**Warm path storage** holds data that must be available immediately from a device for reporting and visualization. For warm path storage, we recommend [Cosmos DB](https://docs.microsoft.com/azure/cosmos-db/introduction). Cosmos DB is a globally distributed, multi-model database.

**Cold path storage** holds data that is kept longer-term and is used for batch processing. For cold path storage, we recommend [Azure Blob Storage](https://docs.microsoft.com/azure/storage/blobs/storage-blobs-introduction). Data can be archived in Blob storage indefinitely at low cost and is easily accessible for batch processing.

**Data transformation** manipulates or aggregates the telemetry stream. Examples include protocol transformation, such as converting binary data to JSON or combining data points. If the data must be transformed before reaching IoT Hub, we recommend using a [protocol gateway](https://docs.microsoft.com/azure/iot-hub/iot-hub-protocol-gateway) (not shown). Otherwise, data can be transformed after it reaches IoT Hub. In that case, we recommend using [Azure Functions](https://docs.microsoft.com/azure/azure-functions/), which has built-in integration with IoT Hub, Cosmos DB, and Blob Storage.

**Business process integration** performs actions based on insights from the device data. These actions could include storing informational messages, raising alarms, sending email or SMS messages, or integrating with CRM. We recommend using [Azure Logic Apps](https://docs.microsoft.com/azure/logic-apps/logic-apps-overview) for business process integration.

**User management** restricts which users or groups can perform actions on devices, such as upgrading firmware. It also defines capabilities for users in applications. We recommend using [Azure Active Directory](https://docs.microsoft.com/azure/active-directory/) to authenticate and authorize users.

### Scalability considerations

An IoT application should be built as discrete services that can scale independently. Consider the following scalability points:

**IoTHub**. For IoT Hub, consider the following scale factors:

* The maximum [daily quota](https://docs.microsoft.com/azure/iot-hub/iot-hub-devguide-quotas-throttling) of messages into IoT Hub.
* The quota of connected devices in an IoT Hub instance.
* Ingestion throughput — how quickly IoT Hub can ingest messages.
* Processing throughput — how quickly the incoming messages are processed.

Each IoT hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that devices can send to the hub. For more information, see IoT Hub quotas and throttling. You can scale up a hub without interrupting existing operations.

**Stream Analytics**. Stream Analytics jobs scale best if they are parallel at all points in the Stream Analytics pipeline, from input to query to output. A fully parallel job allows Stream Analytics to split the work across multiple compute nodes. Otherwise, Stream Analytics has to combine the stream data into one place. For more information, see [Leverage query parallelization in Azure Stream Analytics](https://docs.microsoft.com/azure/stream-analytics/stream-analytics-parallelization).

IoT Hub automatically partitions device messages based on the device ID. All of the messages from a particular device will always arrive on the same partition, but a single partition will have messages from multiple devices. Therefore, the unit of parallelization is the partition ID.

**Functions**. When reading from the Event Hubs endpoint, there is a maximum of function instance per event hub partition. The maximum processing rate is determined by how fast one function instance can process the events from a single partition. The function should process messages in batches.

**Cosmos DB**. To scale out a Cosmos DB collection, create the collection with a partition key and include the partition key in each document that you write. For more information, see [Best practices when choosing a partition key](https://docs.microsoft.com/azure/cosmos-db/partitioning-overview#choose-partitionkey).

* If you store and update a single document per device, the device ID is a good partition key. Writes are evenly distributed across the keys. The size of each partition is strictly bounded because there is a single document for each key value.
* If you store a separate document for every device message, using the device ID as a partition key would quickly exceed the 10-GB limit per partition. Message-ID is a better partition key in that case. Typically you would still include device ID in the document for indexing and querying.

### Security considerations

Use Azure Key Vault to protect secrets

[Azure Key Vault](https://docs.microsoft.com/azure/key-vault/key-vault-overview) is used to securely store and tightly control access to tokens, passwords, certificates, API keys, and other secrets. Also, secrets stored in Azure Key Vault are centralized, giving the added benefits of only needing to update secrets in one place, such as an application key value after recycling the key for security purposes.

In this solution accelerator, we store application secrets in Azure Key Vault, then configure the Function Apps, Web App, and Azure Databricks to connect to Key Vault securely. These services connect to Key Vault using managed identities and a Key Vault-backed Databricks secret store, respectively.

Trustworthy and secure communication

All information received from and sent to a device must be trustworthy. Unless a device can support the following cryptographic capabilities, it should be constrained to local networks, and all internetwork communication should go through a field gateway:

* Data encryption with a provably secure, publicly analyzed, and broadly implemented symmetric-key encryption algorithm.
* Digital signature with a provably secure, publicly analyzed, and broadly implemented symmetric-key signature algorithm.
* Support for either TLS 1.2 for TCP or other stream-based communication paths or DTLS 1.2 for datagram-based communication paths. Support of X.509 certificate handling is optional and can be replaced by the more compute-efficient and wire-efficient pre-shared key mode for TLS, which can be implemented with support for the AES and SHA-2 algorithms.
* Updateable key-store and per-device keys. Each device must have unique key material or tokens that identify it toward the system. The devices should store the key securely on the device (for example, using a secure key-store). The device should be able to update the keys or tokens periodically, or reactively in emergencies such as a system breach.
* The firmware and application software on the device must allow for updates to enable the repair of discovered security vulnerabilities.

However, many devices are too constrained to support these requirements. In that case, a field gateway should be used. Devices connect securely to the field gateway through a local area network, and the gateway enables secure communication to the cloud.

Physical tamper-proofing

We strongly recommend that device design incorporates features that defend against physical manipulation attempts, to help ensure the security integrity and trustworthiness of the overall system.

For example:

* Choose microcontrollers/microprocessors or auxiliary hardware that provide secure storage and use of cryptographic key material, such as trusted platform module (TPM) integration.
* Secure boot loader and secure software loading, anchored in the TPM.
* Use sensors to detect intrusion attempts and attempts to manipulate the device environment with alerting and potentially "digital self-destruction" of the device.

For additional security considerations, see [Internet of Things (IoT) security architecture](https://docs.microsoft.com/azure/iot-fundamentals/iot-security-architecture).

### Monitoring and logging

Logging and monitoring systems are used to determine whether the solution is functioning and to help troubleshoot problems. Monitoring and logging systems help answer the following operational questions:

* Are devices or systems in an error condition?
* Are devices or systems correctly configured?
* Are devices or systems generating accurate data?
* Are systems meeting the expectations of both the business and end customers?

Logging and monitoring tools are typically comprised of the following four components:

* System performance and timeline visualization tools to monitor the system and for basic troubleshooting.
* Buffered data ingestion, to buffer log data.
* Persistence store to store log data.
* Search and query capabilities to view log data for use in detailed troubleshooting.

Monitoring systems provide insights into the health, security, and stability, and performance of an IoT solution. These systems can also provide a more detailed view, recording component configuration changes, and providing extracted logging data that can surface potential security vulnerabilities, enhance the incident management process, and help the owner of the system troubleshoot problems. Comprehensive monitoring solutions include the ability to query information for specific subsystems or aggregating across multiple subsystems.

Monitoring system development should begin by defining the normal operation, regulatory compliance, and audit requirements. Metrics collected may include:

* Physical devices, edge devices, and infrastructure components that are reporting configuration changes.
* Applications that are reporting configuration changes, security audit logs, request rates, response times, error rates, and garbage collection statistics for managed languages.
* Databases, persistence stores, and caches reporting query and write performance, schema changes, security audit log, locks or deadlocks, index performance, CPU, memory, and disk usage.
* Managed services (IaaS, PaaS, SaaS, and FaaS) reporting health metrics and configuration changes that impact dependent system health and performance.

Visualization of monitoring metrics alert operators to system instabilities and facilitate incident response.

Tracing telemetry

Tracing telemetry allows an operator to follow the journey of a piece of telemetry from creation through the system. Tracing is essential for debugging and troubleshooting. For IoT solutions that use Azure IoT Hub and the [IoT Hub Device SDKs](https://docs.microsoft.com/azure/iot-hub/iot-hub-devguide-sdks), tracing datagrams can be originated as Cloud-to-Device messages and included in the telemetry stream.

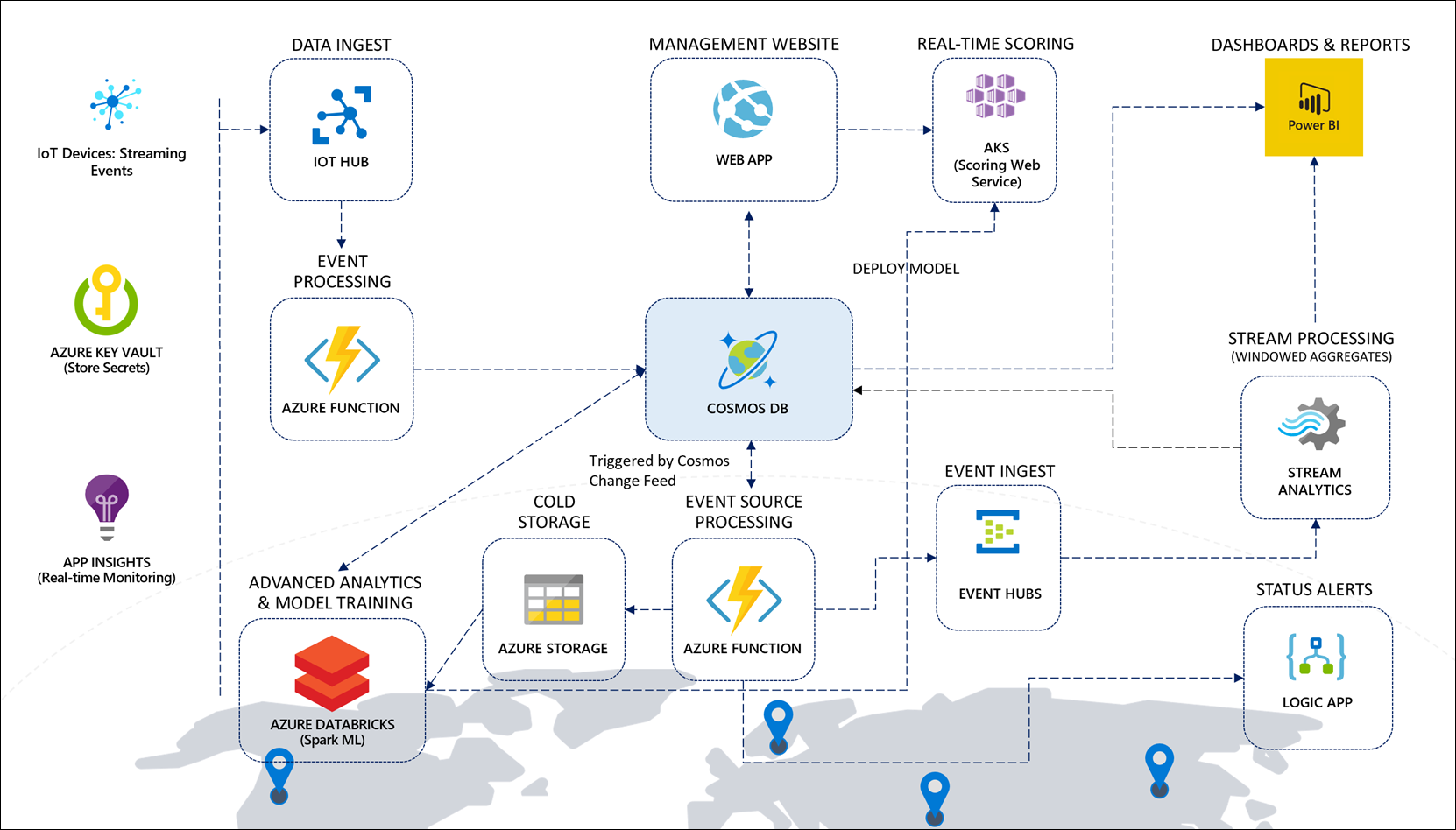
Logging

Logging systems are integral in understanding what actions or activities a solution has performed, failures that have occurred, and can provide help in fixing those failures. Logs can be analyzed to help understand and remedy error conditions, enhance performance characteristics, and ensure compliance with governing rules and regulations.

Though plain-text logging is a lower impact on upfront development costs, it is more challenging for a machine to parse/read. We recommend structured logging be used, as collected information is both machine-parsable and human-readable. Structured logging adds situational context and metadata to the log information. In structured logging, properties are first-class citizens formatted as key/value pairs, or with a fixed schema, to enhance search and query capabilities.

# High-level architecture for the solution accelerator

We have covered a lot of ground with concepts covered by the Cosmos DB IoT solution accelerator. When we apply those concepts to the starter artifacts provided in the accelerator, the outcome is the following high-level architecture:



## Adapting the sample scenario to your own

The sample devices and data provided in this solution accelerator are based on fleets of vehicles containing sensors used to track vehicle and refrigeration unit telemetry. Feel free to adapt this to your own scenario, whether you work with oil field pumps, temperature sensors, or any of the thousands of possibilities in the IoT space.

We refer to vehicles and oil field pumps throughout the guide to help you adapt the sample scenario to your own.

## Architecture components

* Data ingest, event processing, and storage:

The solution for the IoT scenario centers around **Cosmos DB**, which acts as the globally-available, highly scalable data storage for streaming event data, fleet, consignment, package, and trip metadata, and aggregate data for reporting. Vehicle telemetry (you may not have vehicles, but oil field pumps) data flows in from the data generator, through registered IoT devices in **IoT Hub**, where an **Azure function** processes the event data and inserts it into a telemetry container in Cosmos DB.

* Downstream event processing with Azure Functions:

The Cosmos DB change feed triggers three separate Azure functions, with each managing their own checkpoints so they can process the same incoming data without conflicting with one another. One function serializes the event data and stores it into time-sliced folders in **Azure Storage** for long-term cold storage of raw data. Another function processes the vehicle (or pump) telemetry, aggregating the batch data and updating the trip and consignment status in the metadata container, based on odometer readings and whether the trip is running on schedule. This function also triggers a **Logic App** to send email alerts when trip milestones are reached. A third function sends the event data to **Event Hubs**, which in turn triggers **Stream Analytics** to execute time window aggregate queries.

* Stream processing, dashboards, and reports:

The Stream Analytics queries output vehicle-specific aggregates to the Cosmos DB metadata container, and overall vehicle aggregates to **Power BI** to populate its real-time dashboard of vehicle status information. A Power BI Desktop report displays detailed vehicle, trip, and consignment information pulled directly from the Cosmos DB metadata container. It also displays batch battery failure predictions, pulled from the maintenance container.

* Advanced analytics and ML model training:

**Azure Databricks** is used to train a machine learning model to predict vehicle battery failure, based on historical information. It saves a trained model locally for batch predictions, and deploys a model and scoring web service to **Azure Kubernetes Service (AKS)** or **Azure Container Instances (ACI)** for real-time predictions. Azure Databricks also uses the **Spark Cosmos DB connector** to pull down each day's trip information to make batch predictions on battery failure and store the predictions in the maintenance container.

We use vehicle battery data in this sample scenario to provide a concrete example of how Apache Spark, through an Azure Databricks workspace, can directly connect to Cosmos DB and use it as a source for advanced analytics and machine learning. The data, or the machine learning model, are not important. What we highlight is the Spark connector for Cosmos DB and the Azure Key Vault-backed secret store to securely access secrets, like the Cosmos DB connection string. You may choose to adapt the supplied notebooks to your scenario or skip the ML pieces altogether.

* Fleet management web app, security, and monitoring:

A **Web App** allows Contoso Auto to manage vehicles and view consignment, package, and trip information that is stored in Cosmos DB. The Web App is also used to make real-time battery failure predictions while viewing vehicle information. **Azure Key Vault** is used to securely store centralized application secrets, such as connection strings and access keys, and is used by the Function Apps, Web App, and Azure Databricks. Finally, **Application Insights** provides real-time monitoring, metrics, and logging information for the Function Apps and Web App.

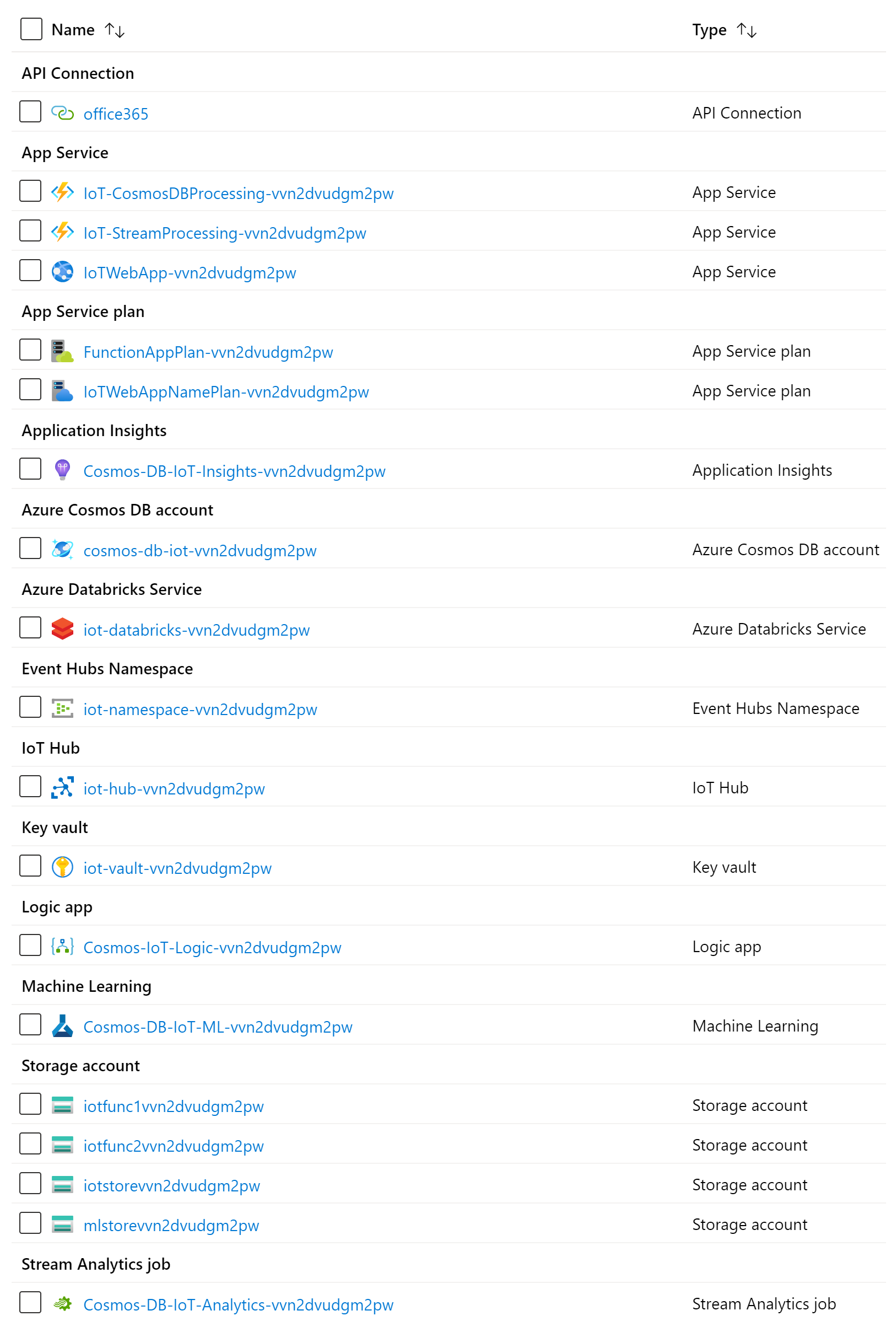
# Detailed walk-through of the solution accelerator components

Now that we have established a baseline for high-level concepts and how they apply to the sample scenario, this section provides a detailed walk-through of each of the components and how information flows from one end of the architecture to the other.

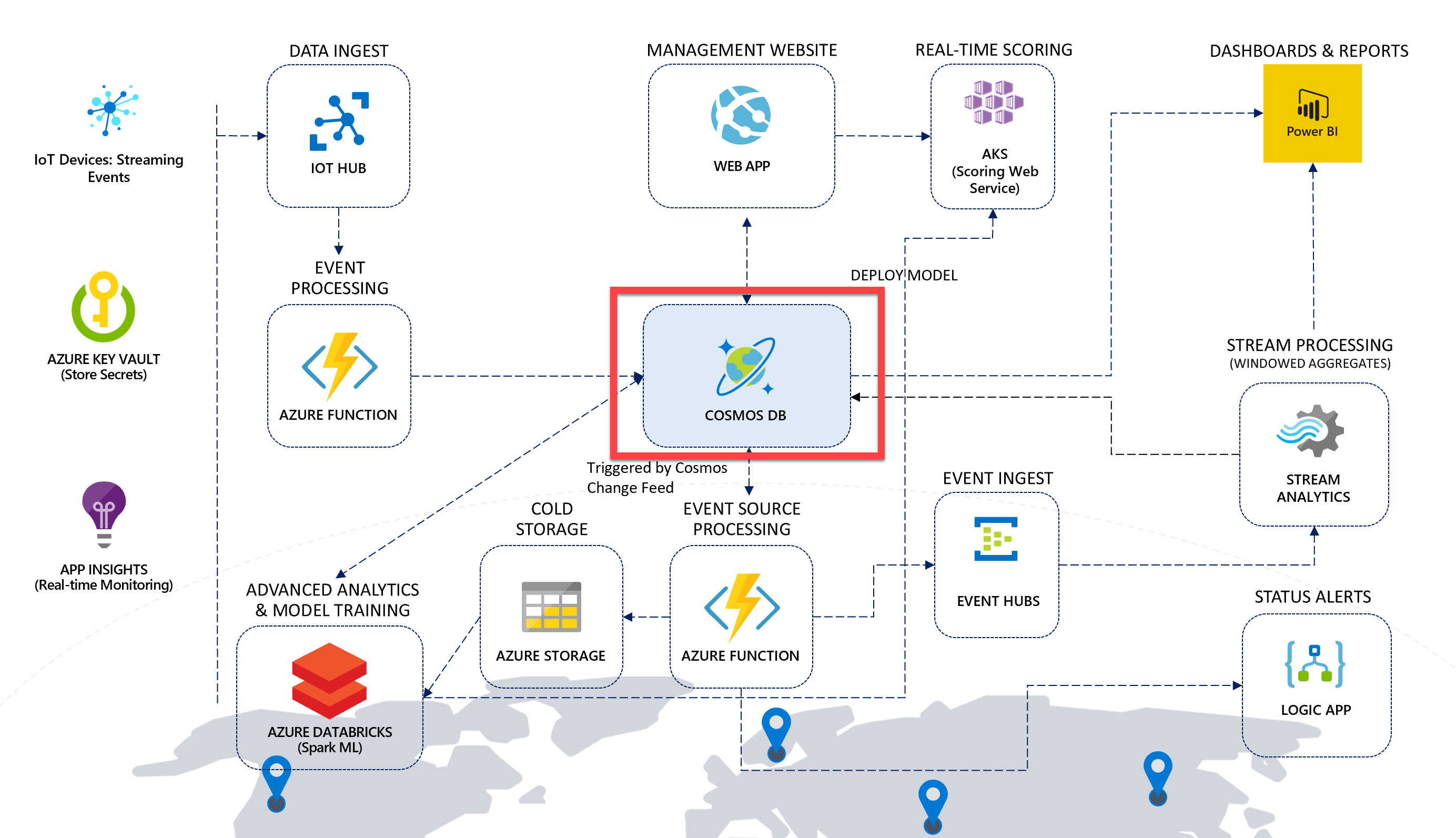
## List of deployed Azure resources

After you execute the deployment template, your Azure resource group should look similar to the following:

|  |  |  |
| --- | --- | --- |
| NAME | TYPE | DESCRIPTION |
| API Connection | | |
| office365 | API Connection | Office 365 connection for the Logic App |
| App Service | | |
| IoT-CosmosDBProcessing-vvn2dvudgm2pw | App Service | Function App that contains functions triggered by the Cosmos DB change feed |
| IoT-StreamProcessing-vvn2dvudgm2pw | App Service | Function App that contains a function triggered by IoT Hub and outputs to the Cosmos DB telemetry container |
| IoTWebApp-vvn2dvudgm2pw | App Service | The example management web app that performs CRUD operations against Cosmos DB |
| App Service Plan | | |
| FunctionAppPlan-vvn2dvudgm2pw | App Service plan | The consumption-based App Service plan for both Function Apps |
| IoTWebAppNamePlan-vvn2dvudgm2pw | App Service plan | The App Service plan for the management web app |
| Application Insights | | |
| Cosmos-DB-IoT-Insights-vvn2dvudgm2pw | Application Insights | Application Insights instance |
| Azure Cosmos DB account | | |
| cosmos-db-iot-vvn2dvudgm2pw | Azure Cosmos DB account | The Azure Cosmos DB account configured with the SQL API |
| Azure Databricks Service | | |
| iot-databricks-vvn2dvudgm2pw | Azure Databricks Service | Azure Databricks workspace used for advanced analytics and Machine Learning (ML) operations |
| Event Hubs Namespace | | |
| iot-namespace-vvn2dvudgm2pw | Event Hubs Namespace | The Event Hubs namespace that contains the “reporting” event hub. A Cosmos DB change feed-triggered function writes to this event hub, which in turn triggers the Stream Analytics job |
| IoT Hub | | |
| iot-hub-vvn2dvudgm2pw | IoT Hub | The IoT Hub instance for managing devices and ingesting telemetry |
| Key Vault | | |
| iot-vault-vvn2dvudgm2pw | Key vault | Azure Key Vault contains secrets used by the Function Apps, web app, and Databricks |
| Logic App | | |
| Cosmos-IoT-Logic-vvn2dvudgm2pw | Logic app | The Logic App that sends notification emails. A Cosmos DB change feed-triggered function triggers the logic app via its HTTP trigger when it needs to send a notification email |
| Machine Learning | | |
| Cosmos-DB-IoT-ML-vvn2dvudgm2pw | Machine Learning | The Azure Machine Learning workspace that manages the custom ML model training, storage, and deployment |
| Storage account | | |
| iotfunc1vvn2dvudgm2pw | Storage account | Azure Storage account for the Cosmos DB Function App |
| iotfunc2vvn2dvudgm2pw | Storage account | Azure Storage account for the stream processing (IoT Hub) Function App |
| iotstorevvn2dvudgm2pw | Storage account | Azure Storage account used for cold storage of all telemetry. A Cosmos DB change feed-triggered function outputs all telemetry to a container (“telemetry”) in time-sliced path format: /yyyy/MM/dd/HH/mm/ss-fffffff.json |
| mlstorevvn2dvudgm2pw | Storage account | Azure Storage account used by the Azure Machine Learning workspace |
| Stream Analytics job | | |
| Cosmos-DB-IoT-Analytics-vvn2dvudgm2pw | Stream Analytics job | Azure Stream Analytics job used for stream processing of events sent to Event Hubs from a Cosmos DB change feed-triggered function. The query uses window functions to create aggregates over time windows of varying length, outputting the results to Cosmos DB and Power BI |



## Azure Cosmos DB: Data is at the core

This high-speed managed NoSQL database is the core component of our solution. 

Cosmos DB was created from the ground up to be a cloud-native, high-speed, globally distributed managed NoSQL database service that fits nicely at the core of many modern solutions today. Its ability to provide very low latency and high throughput that can be tuned for varying workloads makes it an ideal candidate for ingesting and serving telemetry, operational, and analytical data in our reference architecture.

The Cosmos DB database contains three SQL-based containers:

* **telemetry**: Used for ingesting hot vehicle telemetry data with a 90-day lifespan (TTL).
* **metadata**: Stores vehicle, consignment, package, trip, and aggregate event data.
* **maintenance**: The batch battery failure predictions are stored here for reporting purposes.

Each of these containers is configured based on the type of data they hold, as well as the type of workload (read-heavy, write-heavy, occasional access, etc.). Let us evaluate the configuration for each container and the design decisions behind each setting.

### 1. Maintenance

The **Throughput** value for this container is set to **400** RU/s. This is the lowest setting for a container, which is sufficient for the throughput requirements for maintenance data due to low read and write usage.

The **Partition Key** is set to **/vin** (VIN means *vehicle identification number*) so we can group maintenance data by vehicle, and because the vin field is used in most queries.

The **Indexing Policy** is set to the default value, which automatically indexes all fields for each document stored in the container. This is because all paths are included (remember, since we are storing JSON documents, we use paths to identify the property since they can exist within child collections in the document) by setting the value of includedPaths to "path": "/\*", and the only excluded path is the internal \_etag property, which is used for versioning the documents. The default Indexing Policy is:

{

"indexingMode": "consistent",

"automatic": true,

"includedPaths": [

{

"path": "/\*"

}

],

"excludedPaths": [

{

"path": "/\"\_etag\"/?"

}

]

}

### 2. Metadata

The **Throughput** value for this container is set to **50000** RU/s. We are initially setting the throughput on this container to this high number of RU/s because the data generator will perform a bulk insert of metadata the first time it runs. After inserting the data, it will programmatically reduce the throughput to **15000**.

The **Partition Key** is set to **/partitionKey**. This is because we store several different types of documents (records) in this container. As such, the fields vary between document types. Each document has a partitionKey field added, and an entityType field to indicate the type of document, such as "Vehicle", "Package", or "Trip". The partitionKey field is set to a field property value appropriate to the document type, such as vin for Vehicle documents. Trip documents also use vin as the partition key since trip data is retrieved by the related vehicle's VIN and is often retrieved along with vehicle data. The entityType field can be used to filter by type of document within a given partition key.

The **Indexing Policy** is set to the default value.

### 3. Telemetry

The **Throughput** value for this container is set to **15000** RU/s, which is optimal for handling the rate of vehicle telemetry data written to this container.

The **Partition Key** is set to **/partitionKey**. The partitionKey property represents a synthetic composite partition key for the Cosmos DB container, consisting of the VIN + current year/month. Using a composite key instead of simply the VIN provides us with the following benefits:

* 1. Distributing the write workload at any given point in time over a high cardinality of partition keys.
  2. Ensuring efficient routing on queries on a given VIN - you can spread these across time, e.g. SELECT \* FROM c WHERE c.partitionKey IN (“VIN123-2019-01”, “VIN123-2019-02”, …).
  3. Scale beyond the 10GB quota for a single partition key value.

Notice that the **Time to Live** setting is set to **On (no default)**. This was turned off for the other containers. Time to Live (TTL) tells Cosmos DB when to expire, or delete, the document(s) automatically. This setting can help save storage costs by removing what you no longer need. Typically, this is used on hot data or data that must be expired after a period of time due to regulatory requirements. Turning the Time to Live setting on with no default allows us to define the TTL individually for each document, giving us more flexibility in deciding which documents should expire after a set period of time. To do this, we have a ttl field on the document that is saved to this container that specifies the TTL in seconds.

Now view the **Indexing Policy**, which is different from the default policy the other containers use. This custom policy is optimized for write-heavy workloads by excluding all paths and only including the paths used when we query the container (vin, state, and partitionKey):

{

"indexingMode": "consistent",

"automatic": true,

"includedPaths": [

{

"path": "/vin/?"

},

{

"path": "/state/?"

},

{

"path": "/partitionKey/?"

}

],

"excludedPaths": [

{

"path": "/\*"

},

{

"path": "/\"\_etag\"/?"

}

]

}

About Cosmos DB throughput

You will notice that we have intentionally set the **throughput** in RU/s for each container, based on our anticipated event processing and reporting workloads. In Azure Cosmos DB, provisioned throughput is represented as request units/second (RUs). RUs measure the cost of both read and write operations against your Cosmos DB container. Because Cosmos DB is designed with transparent horizontal scaling (e.g., scale-out) and multi-master replication, you can very quickly and easily increase or decrease the number of RUs to handle thousands to hundreds of millions of requests per second around the globe with a single API call.

Cosmos DB allows you to increment/decrement the RUs in small increments of 100 at the database level, or at the container level. It is recommended that you configure throughput at the container granularity for guaranteed performance for the container all the time, backed by SLAs. Other guarantees that Cosmos DB delivers are 99.999% read and write availability all around the world, with those reads and writes being served in less than 10 milliseconds at the 99th percentile.

When you set a number of RUs for a container, Cosmos DB ensures that those RUs are available in all regions associated with your Cosmos DB account. When you scale out the number of regions by adding a new one, Cosmos will automatically provision the same quantity of RUs in the newly added region. You cannot selectively assign different RUs to a specific region. These RUs are provisioned for a container (or database) for all associated regions.

About Cosmos DB partitioning

When you created each container, you were required to define a **partition key**. As you will see later in the lab when you review the solution source code, each document stored within a collection contains a partitionKey property. One of the most important decisions one must make when creating a new container is to select an appropriate partition key for the data. A partition key should provide even distribution of storage and throughput (measured in requests per second) at any given time to avoid storage and performance bottlenecks. For instance, vehicle metadata stores the VIN, which is a unique value for each vehicle, in the partitionKey field. Trip metadata also uses the VIN for the partitionKey field, since trips are most often queried by VIN, and trip documents are stored in the same logical partition as vehicle metadata since they are likely to be queried together, preventing fan-out, or cross-partition queries. Package metadata, on the other hand, uses the Consignment ID value for the partitionKey field for the same purposes. The partition key should be present in the bulk of queries for read-heavy scenarios to avoid excessive fan-out across numerous partitions. This is because each document with a specific partition key value belongs to the same logical partition and is also stored in and served from the same physical partition. Each physical partition is replicated across geographical regions, resulting in global distribution.

Choosing an appropriate partition key for Cosmos DB is a critical step for ensuring balanced reads and writes, scaling, and, in the case of this solution, in-order change feed processing per partition. While there are no limits, per se, on the number of logical partitions, a single logical partition is allowed an upper limit of 10 GB of storage. Logical partitions cannot be split across physical partitions. For the same reason, if the partition key chosen is of bad cardinality, you could potentially have skewed storage distribution. For instance, if one logical partition becomes larger faster than the others and hits the maximum limit of 10 GB, while the others are nearly empty, the physical partition housing the maxed out logical partition cannot split and could cause application downtime.

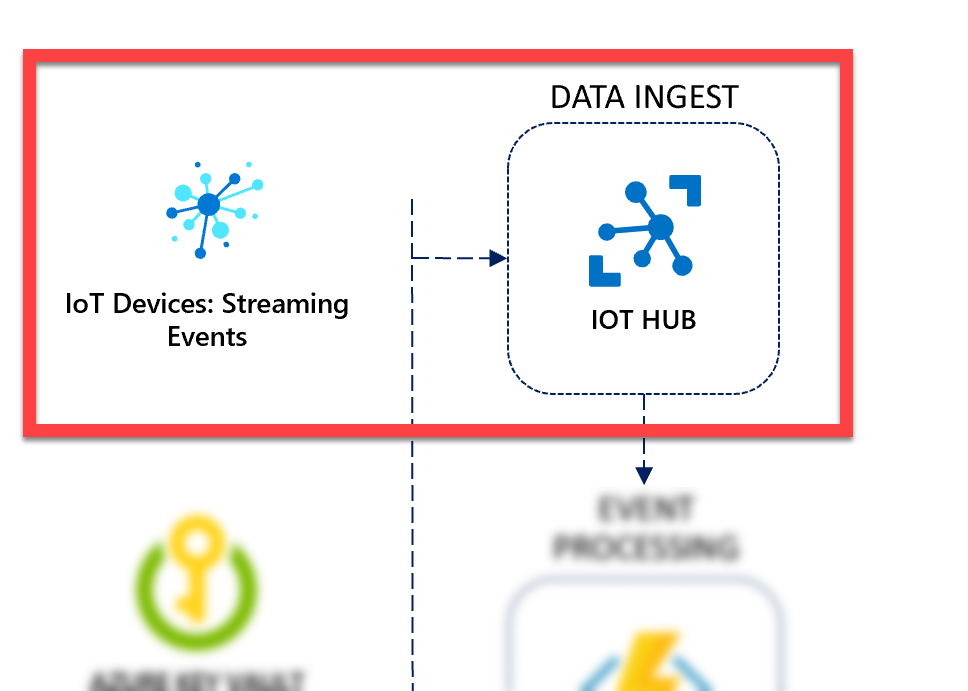
About the Cosmos DB indexing policies

The default indexing policy for newly created containers indexes every property of every item, enforcing range indexes for any string or number, and spatial indexes for any GeoJSON object of type Point. This allows you to get high query performance without having to think about indexing and index management upfront. Since the metadata and maintenance containers have more read-heavy workloads than telemetry, it makes sense to use the default indexing policy where query performance is optimized. Since we need faster writes for telemetry, we exclude unused paths. The use of indexing paths can offer improved write performance and lower index storage for scenarios in which the query patterns are known beforehand, as indexing costs are directly correlated to the number of unique paths indexed.

The indexing mode for all three containers is set to **Consistent**. This means the index is updated synchronously as items are added, updated, or deleted, enforcing the consistency level configured for the account for read-based queries. The other indexing mode one could choose is None, which disables indexing on the container. Usually, this mode is used when your container acts as a pure key-value store, and you do not need indexes for any of the other properties. It is possible to dynamically change the consistency mode prior to executing bulk operations, then changing the mode back to Consistent afterward, if the potential performance increase warrants the temporary change.

## Cloud gateway: Managing and devices and ingesting telemetry with IoT Hub

A cloud gateway provides a cloud hub for devices to connect securely to the cloud and send data. It also provides device management, capabilities, including command and control of devices. For the cloud gateway, we use IoT Hub (<https://docs.microsoft.com/azure/iot-hub/>). IoT Hub is a hosted cloud service that ingests events from devices, acting as a message broker between devices and backend services. IoT Hub provides secure connectivity, event ingestion, bidirectional communication, and device management.



Another benefit of using IoT Hub in our solution, beyond IoT device management and bi-directional communication, is that it can ingest data at an extremely high scale. Under the covers, IoT Hub uses Azure Event Hubs (<https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-about>) to ingest streaming data. In our case, this streaming data is device telemetry. Event Hubs can ingest millions of events per second while guaranteeing at least once delivery with low latency and high throughput. While these data ingestion capabilities carry over to IoT Hub, both services are designed for different purposes.

IoT Hub was developed to address the unique requirements of connecting IoT devices to the Azure cloud while Event Hubs was designed for big data streaming. Microsoft recommends using Azure IoT Hub to connect IoT devices to Azure.

Azure IoT Hub is the cloud gateway that connects IoT devices to gather data and drive business insights and automation. In addition, IoT Hub includes features that enrich the relationship between your devices and your backend systems. Bi-directional communication capabilities mean that while you receive data from devices you can also send commands and policies back to devices. For example, use cloud-to-device messaging to update properties or invoke device management actions. Cloud-to-device communication also enables you to send cloud intelligence to your edge devices with Azure IoT Edge. The unique device-level identity provided by IoT Hub helps better secure your IoT solution from potential attacks.

The following table provides details about how the two tiers of IoT Hub compare to Event Hubs when you're evaluating them for IoT capabilities. For more information about the standard and basic tiers of IoT Hub, see [How to choose the right IoT Hub tier](https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-scaling).

| IoT Capability | IoT Hub standard tier | IoT Hub basic tier | Event Hubs |
| --- | --- | --- | --- |
| Device-to-cloud messaging | Check | Check | Check |
| Protocols: HTTPS, AMQP, AMQP over webSockets | Check | Check | Check |
| Protocols: MQTT, MQTT over webSockets | Check | Check |  |
| Per-device identity | Check | Check |  |
| File upload from devices | Check | Check |  |
| Device Provisioning Service | Check | Check |  |
| Cloud-to-device messaging | Check |  |  |
| Device twin and device management | Check |  |  |
| Device streams (preview) | Check |  |  |
| IoT Edge | Check |  |  |

Even if the only use case is device-to-cloud data ingestion, we highly recommend using IoT Hub as it provides a service that is designed for IoT device connectivity.

### Communicating with IoT Hub and Cosmos DB with the data generator

The data generator project provided in the solution accelerator, FleetDataGenerator, communicates with both the Cosmos DB database and IoT Hub. When you walk through the Quickstart guide, you execute the data generator to seed Cosmos DB with data and simulate vehicles.

There are several tasks that the data generator performs, depending on the state of your environment. The first task is that the generator will create the Cosmos DB database and containers with the optimal configuration for this lab if these elements do not exist in your Cosmos DB account. When you run the generator in a few moments, this step will be skipped because you already created them at the beginning of the lab. The second task the generator performs is to seed your Cosmos DB metadata container with data if no data exists. This includes vehicle, consignment, package, and trip data. Before seeding the container with data, the generator temporarily increases the requested RU/s for the container to 50,000 for optimal data ingestion speed. After the seeding process completes, the RU/s are scaled back down to 15,000.

After the generator ensures the metadata exists, it begins simulating the specified number of vehicles. You are prompted to enter a number between 1 and 5, simulating 1, 10, 50, 100, or the number of vehicles specified in your configuration settings, respectively. For each simulated vehicle, the following tasks take place:

1. An IoT device is registered for the vehicle, using the IoT Hub connection string and setting the device ID to the vehicle's VIN. This returns a generated device key.
2. A new simulated vehicle instance (SimulatedVehicle) is added to a collection of simulated vehicles, each acting as an AMQP device and assigned a Trip record to simulate the delivery of packages for a consignment. These vehicles are randomly selected to have their refrigeration units fail and, out of those, some will randomly fail immediately while the others fail gradually.
3. The simulated vehicle creates its own AMQP device instance, connecting to IoT Hub with its unique device ID (VIN) and generated device key.
4. The simulated vehicle asynchronously sends vehicle telemetry information through its connection to IoT Hub continuously until it either completes the trip by reaching the distance in miles established by the Trip record or receiving a cancellation token.

### How to provision your own devices

Within the data generator, we use the [Azure IoT Hub SDK for .NET](https://www.nuget.org/packages/Microsoft.Azure.Devices.Client/) to directly register and manage devices in Azure IoT Hub. There are also [SDKs available](https://docs.microsoft.com/azure/iot-hub/iot-hub-devguide-sdks#azure-iot-hub-device-sdks) for C, Java, Node.js, Python, and iOS.

The **DeviceManager.cs** file within the data generator project shows how to use the Microsoft.Azure.Devices.RegistryManager to perform create, remove, update, and delete operations on devices. First, we instantiate a new instance of RegistryManager from the IoT Hub connection string:

// Create an instance of the RegistryManager from the IoT Hub connection string.

registryManager = RegistryManager.CreateFromConnectionString(connectionString);

The RegisterDevicesAsync method in the DeviceManager helper class demonstrates how to register a single device with IoT Hub. First, it creates a new device and sets its state to Enabled. Then, it attempts to register the new device. If an exception is returned that a device already exists, we retrieve the registered device, set the status to Enabled, and update the device state in IoT Hub with the status change:

/// <summary>

/// Register a single device with IoT Hub.

/// </summary>

/// <param name="connectionString"></param>

/// <param name="deviceId"></param>

/// <returns></returns>

public static async Task<string> RegisterDevicesAsync(string connectionString, string deviceId)

{

//Make sure we're connected

if (registryManager == null)

IotHubConnect(connectionString);

// Create a new device.

var device = new Device(deviceId) {Status = DeviceStatus.Enabled};

try

{

// Register the new device.

device = await registryManager.AddDeviceAsync(device);

}

catch (Exception ex)

{

if (ex is DeviceAlreadyExistsException ||

ex.Message.Contains("DeviceAlreadyExists"))

{

// Device already exists, get the registered device.

device = await registryManager.GetDeviceAsync(deviceId);

// Ensure the device is activated.

device.Status = DeviceStatus.Enabled;

// Update IoT Hub with the device status change.

await registryManager.UpdateDeviceAsync(device);

}

else

{

Program.WriteLineInColor($"An error occurred while registering IoT device '{deviceId}':\r\n{ex.Message}", ConsoleColor.Red);

}

}

// Return the device key.

return device.Authentication.SymmetricKey.PrimaryKey;

}

The RegisterDevicesAsync method returns a symmetric key that the registered device uses to authenticate when connecting to IoT Hub. The deviceId value is the vehicle's VIN in this case. You can use whatever string value you want for your devices, as long as the values are unique. The RegisterDevicesAsync method is called from the SetupVehicleTelemetryRunTasks method in the data generator:

// Register vehicle IoT device, using its VIN as the device ID, then return the device key.

var deviceKey = await DeviceManager.RegisterDevicesAsync(iotHubConnectionString, trip.vin);

Each vehicle IoT device is simulated by the data generator. The simulated device is represented by the SimulatedVehicle class (**SimulatedVehicle.cs**). When a new simulated device is instantiated, the device ID (the vehicle's VIN) and the device's symmetric key are passed into the constructor and used when creating a new MIcrosoft.Azure.Devices.Client.DeviceClient instance:

public SimulatedVehicle(Trip trip, bool causeRefrigerationUnitFailure,

bool immediateRefrigerationUnitFailure, int vehicleNumber,

string iotHubUri, string deviceId, string deviceKey)

{

\_vehicleNumber = vehicleNumber;

\_trip = trip;

\_tripId = trip.id;

\_distanceRemaining = trip.plannedTripDistance + 3; // Pad a little bit extra distance to ensure all events captured.

\_causeRefrigerationUnitFailure = causeRefrigerationUnitFailure;

\_immediateRefrigerationUnitFailure = immediateRefrigerationUnitFailure;

\_IotHubUri = iotHubUri;

DeviceId = deviceId;

DeviceKey = deviceKey;

\_DeviceClient = DeviceClient.Create(\_IotHubUri, new DeviceAuthenticationWithRegistrySymmetricKey(DeviceId, DeviceKey));

}

You can update this code to simulate your own IoT devices. When you are ready to **use physical devices**, follow the [tutorials found in the IoT Hub documentation](https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-get-started-physical).

The best way to provision multiple IoT devices in a secure and scalable manner is to use the [Azure IoT Hub Device Provisioning Service](https://docs.microsoft.com/azure/iot-dps/about-iot-dps) (DPS). Use the [Microsoft Azure Provisioning SDKs](https://docs.microsoft.com/azure/iot-hub/iot-hub-devguide-sdks#microsoft-azure-provisioning-sdks) for the best experience with using DPS.

### How the data generator configures and uses Cosmos DB

There is a lot of code within the data generator project, so we'll just touch on the highlights. The code we do not cover is commented and should be easy to follow if you so desire.

Within the **Main** method of **Program.cs**, the core workflow of the data generator is executed by the following code block:

// Instantiate Cosmos DB client and start sending messages:

using (\_cosmosDbClient = new CosmosClient(cosmosDbConnectionString.ServiceEndpoint.OriginalString,

cosmosDbConnectionString.AuthKey, connectionPolicy))

{

await InitializeCosmosDb();

// Find and output the container details, including # of RU/s.

var container = \_database.GetContainer(MetadataContainerName);

var offer = await container.ReadThroughputAsync(cancellationToken);

if (offer != null)

{

var currentCollectionThroughput = offer ?? 0;

WriteLineInColor(

$"Found collection `{MetadataContainerName}` with {currentCollectionThroughput} RU/s.",

ConsoleColor.Green);

}

// Initially seed the Cosmos DB database with metadata if empty.

await SeedDatabase(cosmosDbConnectionString, cancellationToken);

trips = await GetTripsFromDatabase(numberSimulatedTrucks, container);

}

try

{

// Start sending telemetry from simulated vehicles to Event Hubs:

\_runningVehicleTasks = await SetupVehicleTelemetryRunTasks(numberSimulatedTrucks,

trips, arguments.IoTHubConnectionString);

var tasks = \_runningVehicleTasks.Select(t => t.Value).ToList();

while (tasks.Count > 0)

{

try

{

Task.WhenAll(tasks).Wait(cancellationToken);

}

catch (TaskCanceledException)

{

//expected

}

tasks = \_runningVehicleTasks.Where(t => !t.Value.IsCompleted).Select(t => t.Value).ToList();

}

}

catch (OperationCanceledException)

{

Console.WriteLine("The vehicle telemetry operation was canceled.");

// No need to throw, as this was expected.

}

The top section of the code instantiates a new CosmosClient, using the connection string defined in either appsettings.json or the environment variables. The first call within the block is to InitializeCosmosDb(). We'll dig into this method in a moment, but it is responsible for creating the Cosmos DB database and containers if they do not exist in the Cosmos DB account. Next, we create a new Container instance, which the v3 version of the .NET Cosmos DB SDK uses for operations against a container, such as CRUD and maintenance information. For example, we call ReadThroughputAsync on the container to retrieve the current throughput (RU/s), and we pass it to GetTripsFromDatabase to read Trip documents from the container, based on the number of vehicles we are simulating. In this method, we also call the SeedDatabase method, which checks whether data currently exists and, if not, calls methods in the DataGenerator class (DataGenerator.cs file) to generate vehicles, consignments, packages, and trips, then writes the data in bulk using the BulkImporter class (BulkImporter.cs file). This SeedDatabase method executes the following on the Container instance to adjust the throughput (RU/s) to 50,000 before the bulk import, and back to 15,000 after the data seeding is complete: await container.ReplaceThroughputAsync(desiredThroughput);.

The try/catch block calls SetupVehicleTelemetryRunTasks to register IoT device instances for each simulated vehicle and load up the tasks from each SimulatedVehicle instance it creates. It uses Task.WhenAll to ensure all pending tasks (simulated vehicle trips) are complete, removing completed tasks from the \_runningvehicleTasks list as they finish. The cancellation token is used to cancel all running tasks if you issue the cancel command (Ctrl+C or Ctrl+Break) in the console.

Scroll down the **Program.cs** file until you find the **InitializeCosmosDb()** method. Here is the code for your reference:

private static async Task InitializeCosmosDb()

{

\_database = await \_cosmosDbClient.CreateDatabaseIfNotExistsAsync(DatabaseName);

#region Telemetry container

// Define a new container.

var telemetryContainerDefinition =

new ContainerProperties(id: TelemetryContainerName, partitionKeyPath: $"/{PartitionKey}")

{

IndexingPolicy = { IndexingMode = IndexingMode.Consistent }

};

// Tune the indexing policy for write-heavy workloads by only including regularly queried paths.

// Be careful when using an opt-in policy as we are below. Excluding all and only including certain paths removes

// Cosmos DB's ability to proactively add new properties to the index.

telemetryContainerDefinition.IndexingPolicy.ExcludedPaths.Clear();

telemetryContainerDefinition.IndexingPolicy.ExcludedPaths.Add(new ExcludedPath { Path = "/\*" }); // Exclude all paths.

telemetryContainerDefinition.IndexingPolicy.IncludedPaths.Clear();

telemetryContainerDefinition.IndexingPolicy.IncludedPaths.Add(new IncludedPath { Path = "/vin/?" });

telemetryContainerDefinition.IndexingPolicy.IncludedPaths.Add(new IncludedPath { Path = "/state/?" });

telemetryContainerDefinition.IndexingPolicy.IncludedPaths.Add(new IncludedPath { Path = "/partitionKey/?" });

// Create the container with a throughput of 15000 RU/s.

await \_database.CreateContainerIfNotExistsAsync(telemetryContainerDefinition, throughput: 15000);

#endregion

#region Metadata container

// Define a new container (collection).

var metadataContainerDefinition =

new ContainerProperties(id: MetadataContainerName, partitionKeyPath: $"/{PartitionKey}")

{

// Set the indexing policy to consistent and use the default settings because we expect read-heavy workloads in this container (includes all paths (/\*) with all range indexes).

// Indexing all paths when you have write-heavy workloads may impact performance and cost more RU/s than desired.

IndexingPolicy = { IndexingMode = IndexingMode.Consistent }

};

// Set initial performance to 50,000 RU/s for bulk import performance.

await \_database.CreateContainerIfNotExistsAsync(metadataContainerDefinition, throughput: 50000);

#endregion

#region Maintenance container

// Define a new container (collection).

var maintenanceContainerDefinition =

new ContainerProperties(id: MaintenanceContainerName, partitionKeyPath: $"/vin")

{

IndexingPolicy = { IndexingMode = IndexingMode.Consistent }

};

// Set initial performance to 400 RU/s due to light workloads.

await \_database.CreateContainerIfNotExistsAsync(maintenanceContainerDefinition, throughput: 400);

#endregion

}

This method creates a Cosmos DB database if it does not already exist; otherwise it retrieves a reference to it (await \_cosmosDbClient.CreateDatabaseIfNotExistsAsync(DatabaseName);). Then it creates ContainerProperties for the telemetry, metadata, and maintenance containers. The ContainerProperties object lets us specify the container's indexing policy. We use the default indexing policy for metadata and maintenance since they are read-heavy and benefit from a greater number of paths, but we exclude all paths in the telemetry index policy and add paths only to those properties we need to query, due to the container's write-heavy workload. The telemetry container is assigned a throughput of 15,000 RU/s, 50,000 for metadata for the initial bulk import, then it is scaled down to 15,000, and 400 for maintenance.

## Stream processing, event sourcing, and data management

So far, we have detailed how we are using Cosmos DB to store data and IoT Hub to manage devices and ingest telemetry at scale. This section is about how we process and store the streaming data, react to new data added to Cosmos DB through its Change Feed for downstream processing ([event sourcing pattern](https://docs.microsoft.com/azure/architecture/patterns/event-sourcing)), and manage the data through CRUD (create, read, update, delete) operations through the web app.

The terms “stream processing” and “event sourcing” may be unfamiliar to you. Let us begin by defining these terms before we dive into how we apply them to our architecture.

When working with IoT devices, we are usually addressing a Big Data problem. Big Data is not limited to the size of the data. In fact, there are four aspects of Big Data that define it; the **4 Vs of Big Data**:

1. **Volume**  
   What most people think of… the sheer amount of data. Worldwide, this grows exponentially. **90%** of today’s data has been created in the past **2 years**!
2. **Velocity**  
   The speed of data coming in, and the speed in which you need to process it. This is where **streaming** and **real-time processing** come to play.
3. **Variety**  
   Data comes from so many sources these days, from **structured** relational data sets and financial transactions to **unstructured** data such as chat and SMS messages, IoT devices, images, logs, MRIs, etc. **90%** generated data is **unstructured**.
4. **Veracity**  
   Data can be **unreliable and flawed**, from bad sensor data to human error.

When you work with IoT data, you are likely dealing with two or more of these four aspects of Big Data. **Stream processing** addresses the Velocity of data. Often it also addresses both Volume and Variety, depending on the amount of data you must process and the number of sources from which that data arrives. Real-time stream processing consumes messages from either queue or file-based storage, process the messages, and forward the result to another message queue, file store, or database. Processing may include querying, filtering, and aggregating messages. Stream processing engines must be able to consume endless streams of data and produce results with minimal latency.

In Azure, there are several technology choices for real-time stream processing:

* [Azure Stream Analytics](https://docs.microsoft.com/en-us/azure/stream-analytics/)
* [HDInsight with Spark Streaming](https://docs.microsoft.com/en-us/azure/hdinsight/spark/apache-spark-streaming-overview)
* [Apache Spark in Azure Databricks](https://docs.microsoft.com/en-us/azure/azure-databricks/)
* [HDInsight with Storm](https://docs.microsoft.com/en-us/azure/hdinsight/storm/apache-storm-overview)
* [Azure Functions](https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview)
* [Azure App Service WebJobs](https://docs.microsoft.com/en-us/azure/app-service/web-sites-create-web-jobs)

Source: Azure Data Architecture Guide ([*https://docs.microsoft.com/en-us/azure/architecture/data-guide/technology-choices/stream-processing*](https://docs.microsoft.com/en-us/azure/architecture/data-guide/technology-choices/stream-processing))

In this solution, we use both **Azure Functions** and **Azure Stream Analytics** for stream processing.

The following tables summarize the key differences in capabilities.

General capabilities

| Capability | Azure Stream Analytics | Apache Spark in Azure Databricks | HDInsight with Storm | Azure Functions | Azure App Service WebJobs |
| --- | --- | --- | --- | --- | --- |
| Programmability | Stream analytics query language, JavaScript | [C#/F#](https://github.com/dotnet/spark), Java, Python, R, Scala | C#, Java | C#, F#, Java, Node.js, Python | C#, Java, Node.js, PHP, Python |
| Programming paradigm | Declarative | Mixture of declarative and imperative | Imperative | Imperative | Imperative |
| Pricing model | [Streaming units](https://azure.microsoft.com/pricing/details/stream-analytics/) | [Databricks units](https://azure.microsoft.com/pricing/details/databricks/) | Per cluster hour | Per function execution and resource consumption | Per-app service plan hour |

Integration capabilities

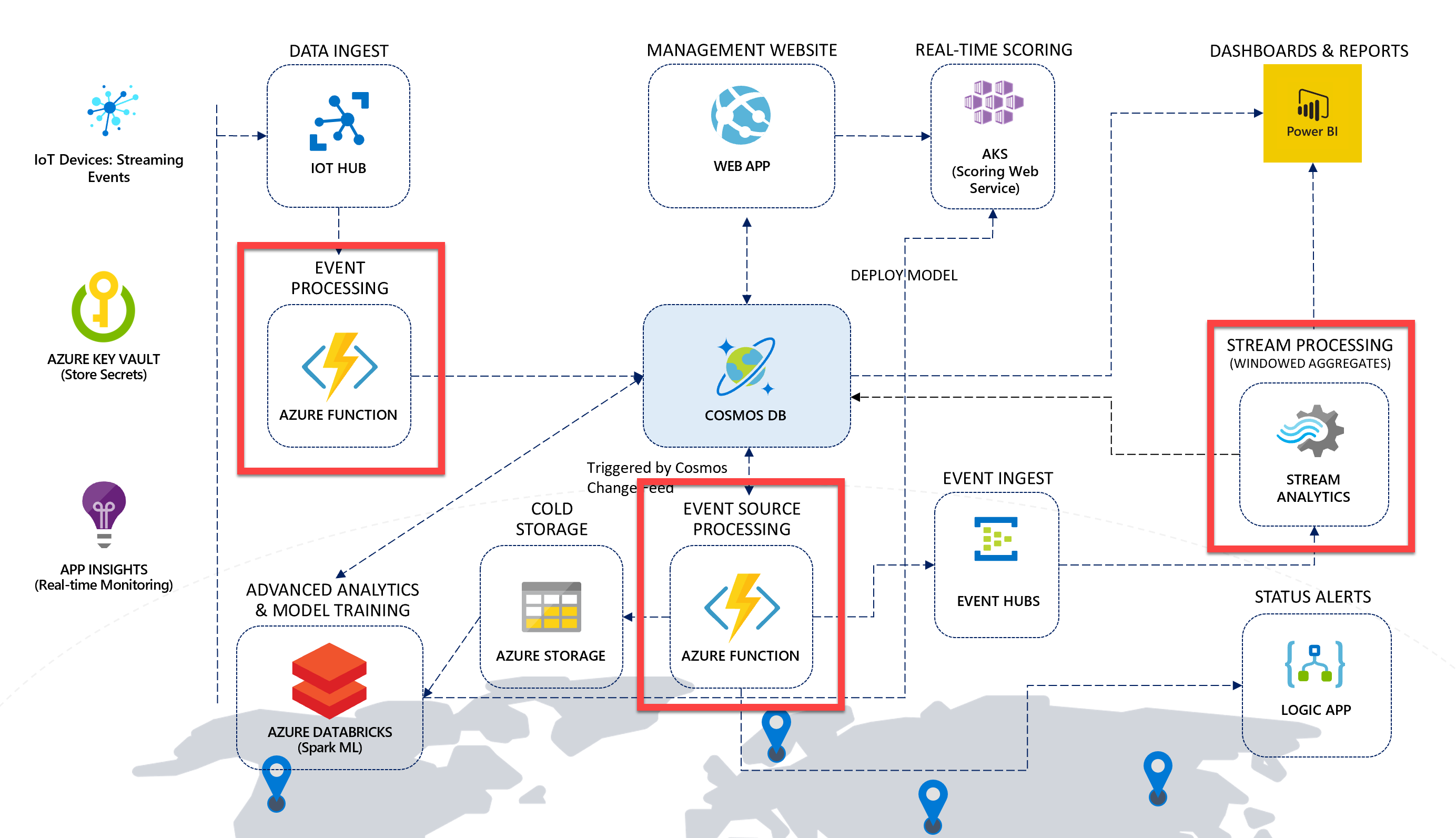
| Capability | Azure Stream Analytics | Apache Spark in Azure Databricks | HDInsight with Storm | Azure Functions | Azure App Service WebJobs |
| --- | --- | --- | --- | --- | --- |
| Inputs | Azure Event Hubs, Azure IoT Hub, Azure Blob storage | Event Hubs, IoT Hub, Kafka, HDFS, Storage Blobs, Azure Data Lake Store | Event Hubs, IoT Hub, Storage Blobs, Azure Data Lake Store | [Supported bindings](https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings#supported-bindings) | Service Bus, Storage Queues, Storage Blobs, Event Hubs, WebHooks, Cosmos DB, Files |
| Sinks | Azure Data Lake Store, Azure SQL Database, Storage Blobs, Event Hubs, Power BI, Table Storage, Service Bus Queues, Service Bus Topics, Cosmos DB, Azure Functions | HDFS, Kafka, Storage Blobs, Azure Data Lake Store, Cosmos DB | Event Hubs, Service Bus, Kafka | [Supported bindings](https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings#supported-bindings) | Service Bus, Storage Queues, Storage Blobs, Event Hubs, WebHooks, Cosmos DB, Files |

Processing capabilities

| Capability | Azure Stream Analytics | Apache Spark in Azure Databricks | HDInsight with Storm | Azure Functions | Azure App Service WebJobs |
| --- | --- | --- | --- | --- | --- |
| Built-in temporal/windowing support | Yes | Yes | Yes | No | No |
| Input data formats | Avro, JSON or CSV, UTF-8 encoded | Any format using custom code | Any format using custom code | Any format using custom code | Any format using custom code |
| Scalability | [Query partitions](https://docs.microsoft.com/en-us/azure/stream-analytics/stream-analytics-parallelization) | Bounded by Databricks cluster scale configuration | Bounded by cluster size | Up to 200 function app instances processing in parallel | Bounded by app service plan capacity |
| Late arrival and out of order event handling support | Yes | Yes | Yes | No | No |

The event sourcing pattern was introduced at the beginning of this document under the high-level concepts topic. As a quick recap, this pattern defines an approach to handling operations on data that's driven by a sequence of events, each of which is recorded in an append-only store. In our implementation, IoT devices send telemetry as a series of events that imperatively describe the state of each device over time to the event store, where they're persisted. Each event represents a set of changes to the data, which is tied back to the source IoT device. The event store, in this case, is Cosmos DB. The Cosmos DB change feed is used to publish these events so that downstream consumers (Azure Functions) are notified so they can handle them.

### Implementing stream processing and the event sourcing pattern with Azure Functions and Stream Analytics

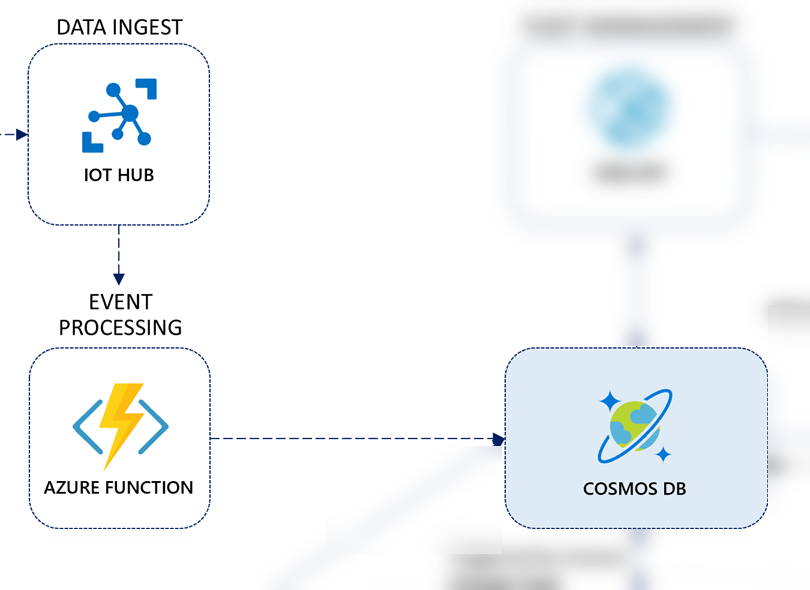


In the architecture for this scenario, Azure Functions play a major role in event processing. These functions execute within an Azure Function App, Microsoft's serverless solution for easily running small pieces of code, or "functions," in the cloud. You can write just the code you need for the problem at hand, without worrying about a whole application or the infrastructure to run it. Functions can make development even more productive, and you can use your development language of choice, such as C#, F#, Node.js, Java, or PHP.

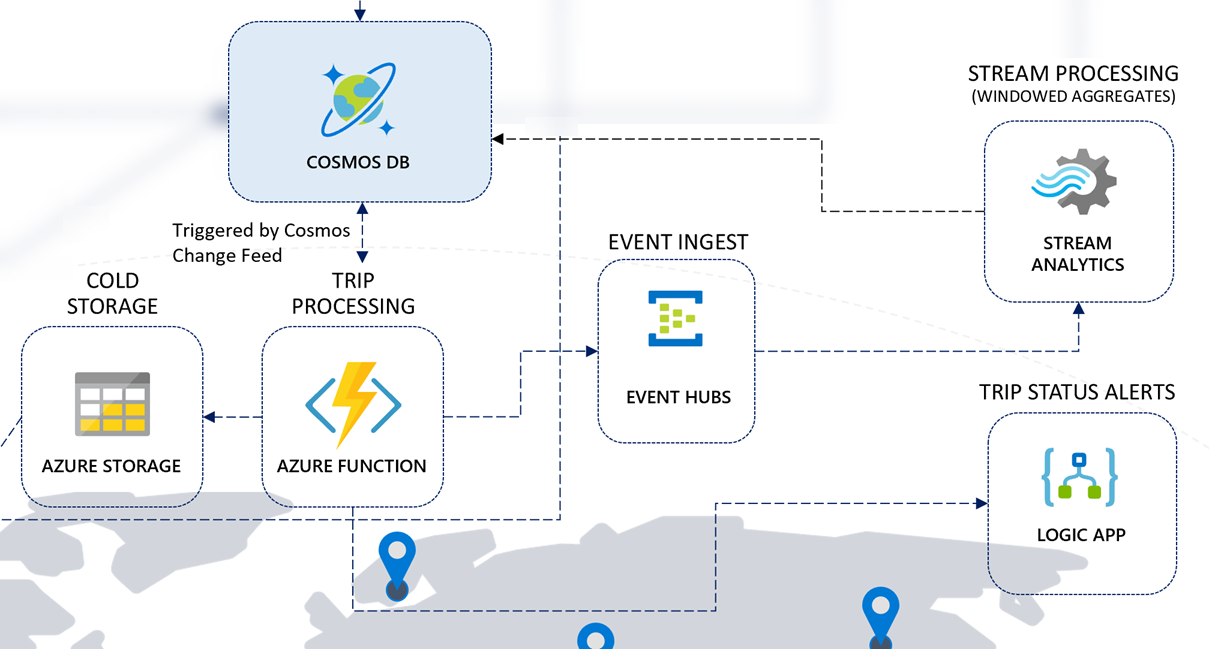
You may wonder, if a Function App contains several functions within, *why do we need two Function Apps instead of one*? The primary reason for using two Function Apps is due to how functions scale to meet demand. When you use the Azure Functions consumption plan, you only pay for the time your code runs. More importantly, Azure automatically handles scaling your functions to meet demand. It scales using an internal scale controller that evaluates the type of trigger the functions are using and applies heuristics to determine when to scale out to multiple instances. The important thing to know is that functions scale at the Function App level. Meaning, if you have one very busy function and the rest are mostly idle, that one busy function causes the entire Function App to scale. Think about this when designing your solution. It is a good idea to **divide extremely high-load functions into separate Function Apps**.

Now let's introduce the Function Apps and Web App and how they contribute to the architecture.

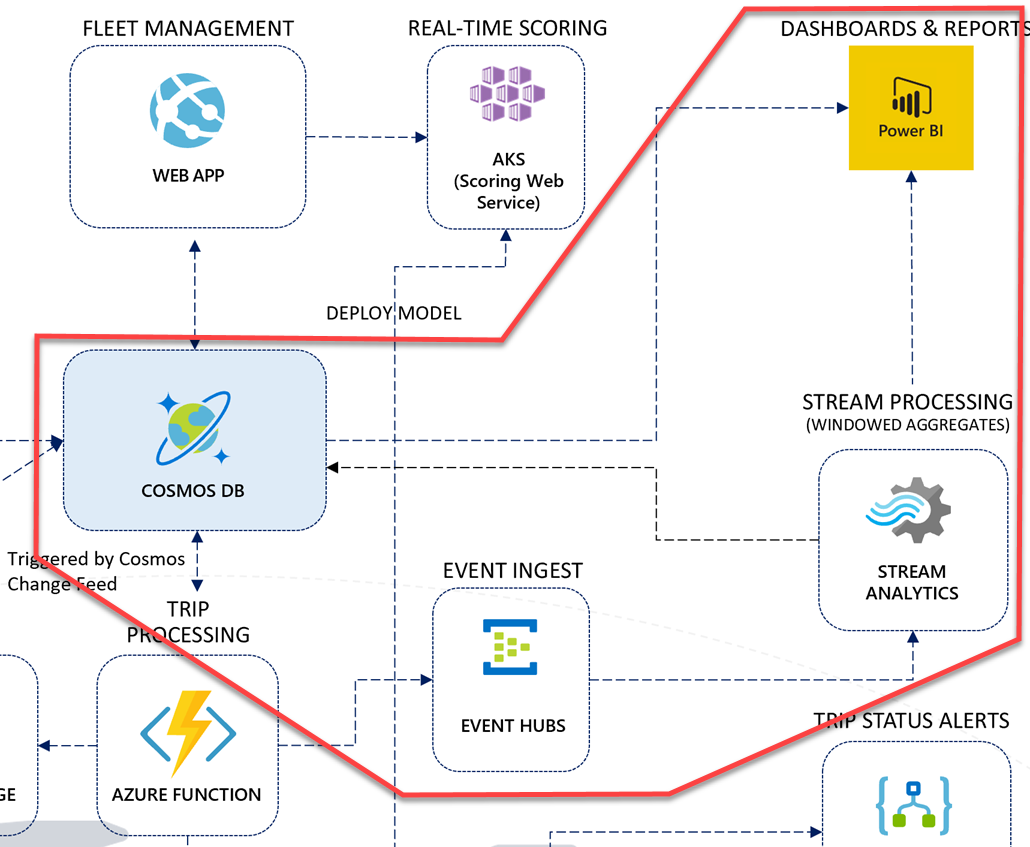
* **IoT-StreamProcessing Function App**: This is the Stream Processing Function App, and it contains two functions:
  + **IoTHubTrigger**: This function is automatically triggered by the IoT Hub's Event Hub endpoint as vehicle telemetry is sent by the data generator. The function performs some light processing to the data by defining the partition key value, the document's TTL, adds a timestamp value, then saves the information to Cosmos DB.
  + **HealthCheck**: This function has an Http trigger that enables users to verify that the Function App is up and running and that each configuration setting exists and has a value. More thorough checks would validate each value against an expected format or by connecting to each service as required. The function will return an HTTP status of 200 (OK) if all values contain non-zero strings. If any are null or empty, the function will return an error (400), indicating which values are missing. The data generator calls this function before running.



* **IoT-CosmosDBProcessing Function App**: This is the Trip Processing Function App. It contains three functions that are triggered by the Cosmos DB Change Feed on the telemetry container. Because the Cosmos DB Change Feed supports multiple consumers, these three functions can run in parallel, processing the same information simultaneously without conflicting with one another. When we define the CosmosDBTrigger for each of these functions, we configure the trigger settings to connect to a Cosmos DB collection named leases to keep track of which change feed events they have processed. We also set the LeaseCollectionPrefix value for each function with a unique prefix so one function does not attempt to retrieve or update the lease information for another. The following functions are in this Function App:
  + **TripProcessor**: This function groups vehicle telemetry data by VIN, retrieves the associated Trip record from the metadata container, updates the Trip record with a trip start timestamp, an end timestamp if completed, and a status showing whether the trip has started, is delayed or has completed. It also updates the associated Consignment record with the status and triggers the Logic App with the trip information if an alert needs to be emailed to the recipient defined in the Function App's app settings (RecipientEmail).
  + **ColdStorage**: This function connects to the Azure Storage account (ColdStorageAccount) and writes the raw vehicle telemetry data for cold storage in the following time-sliced path format: telemetry/custom/scenario1/yyyy/MM/dd/HH/mm/ss-fffffff.json.
  + **SendToEventHubsForReporting**: This function simply sends the vehicle telemetry data straight to Event Hubs, allowing Stream Analytics to apply windowed aggregates and save those aggregates in batches to Power BI and to the Cosmos DB metadata container.
  + **HealthCheck**: As with the function of the same name within the Stream Processing Function App, this function has an Http trigger that enables users to verify that the Function App is up and running and that each configuration setting exists and has a value. The data generator calls this function before running.



The other service we use for stream processing is [Azure Stream Analytics](https://docs.microsoft.com/azure/stream-analytics/stream-analytics-introduction). This service provides real-time analytics through its event-processing engine that can work with high volumes of fast streaming data from multiple sources in parallel. Stream Analytics connects to inputs, such as IoT Hub and Event Hubs, and several outputs it can use as data sinks, including Cosmos DB, Power BI, and several other Azure services. It provides a SQL-like query language used to query over the incoming data, where you can easily adjust the event ordering options and duration of time windows when performing aggregation operations through simple language constructs or configurations. We use Stream Analytics in this solution accelerator to aggregate data over time windows of varying sizes. We use these aggregates to populate materialized views in Cosmos DB and to send small aggregates of data directly to Power BI to update a near real-time dashboard.



If you examine the right-hand side of the solution architecture diagram, you will see a flow of event data that feeds into Event Hubs from a Cosmos DB change feed-triggered function. Stream Analytics uses the event hub as an input source for a set of time window queries that create aggregates for individual vehicle telemetry, and overall vehicle telemetry that flows through the architecture from the vehicle IoT devices. Stream Analytics has two output data sinks:

1. Cosmos DB: Individual vehicle telemetry (grouped by VIN) is aggregated over a 30-second TumblingWindow and saved to the metadata container. This information is used in a Power BI report you will create in Power BI Desktop in a later task to display individual vehicle and multiple vehicle statistics.
2. Power BI: All vehicle telemetry is aggregated over a 10-second TumblingWindow and output to a Power BI data set. This near-real-time data is displayed in a live Power BI dashboard to show in 10-second snapshots how many events were processed, whether there are engine temperature, oil, or refrigeration unit warnings, whether aggressive driving was detected during the period, and the average speed, engine temperature, and refrigeration unit readings.

The **Query** is Stream Analytics' workhorse. This is where we process streaming inputs and write data to our outputs. The Stream Analytics query language is SQL-like, allowing you to use familiar syntax to explore and transform the streaming data, create aggregates, and create materialized views that can be used to help shape your data structure before writing to the output sinks. Stream Analytics jobs can only have one Query, but you can write to multiple outputs in a single Query, as you will do in the steps that follow.

Please take a moment to analyze the query below. We are using the events input name for the Event Hubs input, and the powerbi and cosmosDB outputs, respectively. Also, see where we use the TumblingWindow in durations of 30 seconds for VehicleData, and 10 seconds for VehicleDataAll. The TumblingWindow helps us evaluate events that occurred during the past X seconds and, in our case, create averages over those time periods for reporting.

WITH

VehicleData AS (

select

vin,

AVG(engineTemperature) AS engineTemperature,

AVG(speed) AS speed,

AVG(refrigerationUnitKw) AS refrigerationUnitKw,

AVG(refrigerationUnitTemp) AS refrigerationUnitTemp,

(case when AVG(engineTemperature) >= 400 OR AVG(engineTemperature) <= 15 then 1 else 0 end) as engineTempAnomaly,

(case when AVG(engineoil) <= 18 then 1 else 0 end) as oilAnomaly,

(case when AVG(transmission\_gear\_position) <= 3.5 AND

AVG(accelerator\_pedal\_position) >= 50 AND

AVG(speed) >= 55 then 1 else 0 end) as aggressiveDriving,

(case when AVG(refrigerationUnitTemp) >= 30 then 1 else 0 end) as refrigerationTempAnomaly,

System.TimeStamp() as snapshot

from events TIMESTAMP BY [timestamp]

GROUP BY

vin,

TumblingWindow(Duration(second, 30))

),

VehicleDataAll AS (

select

AVG(engineTemperature) AS engineTemperature,

AVG(speed) AS speed,

AVG(refrigerationUnitKw) AS refrigerationUnitKw,

AVG(refrigerationUnitTemp) AS refrigerationUnitTemp,

COUNT(\*) AS eventCount,

(case when AVG(engineTemperature) >= 318 OR AVG(engineTemperature) <= 15 then 1 else 0 end) as engineTempAnomaly,

(case when AVG(engineoil) <= 20 then 1 else 0 end) as oilAnomaly,

(case when AVG(transmission\_gear\_position) <= 4 AND

AVG(accelerator\_pedal\_position) >= 50 AND

AVG(speed) >= 55 then 1 else 0 end) as aggressiveDriving,

(case when AVG(refrigerationUnitTemp) >= 22.5 then 1 else 0 end) as refrigerationTempAnomaly,

System.TimeStamp() as snapshot

from events t TIMESTAMP BY [timestamp]

GROUP BY

TumblingWindow(Duration(second, 10))

)

-- INSERT INTO POWER BI

SELECT

\*

INTO

powerbi

FROM

VehicleDataAll

-- INSERT INTO COSMOS DB

SELECT

\*,

entityType = 'VehicleAverage',

partitionKey = vin

INTO

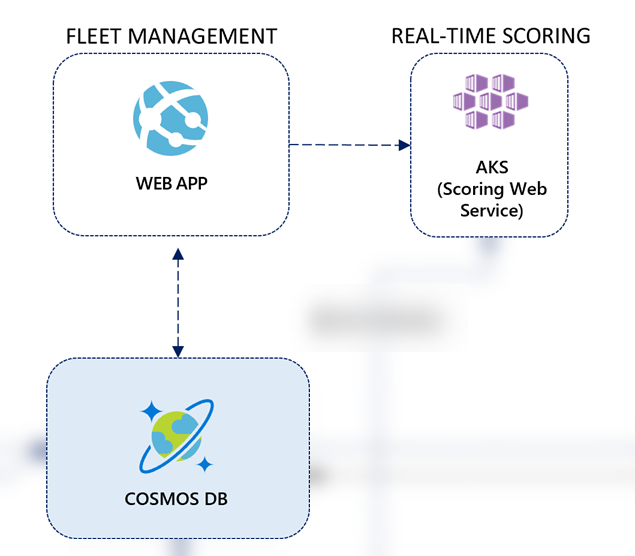
cosmosdb

FROM

VehicleData

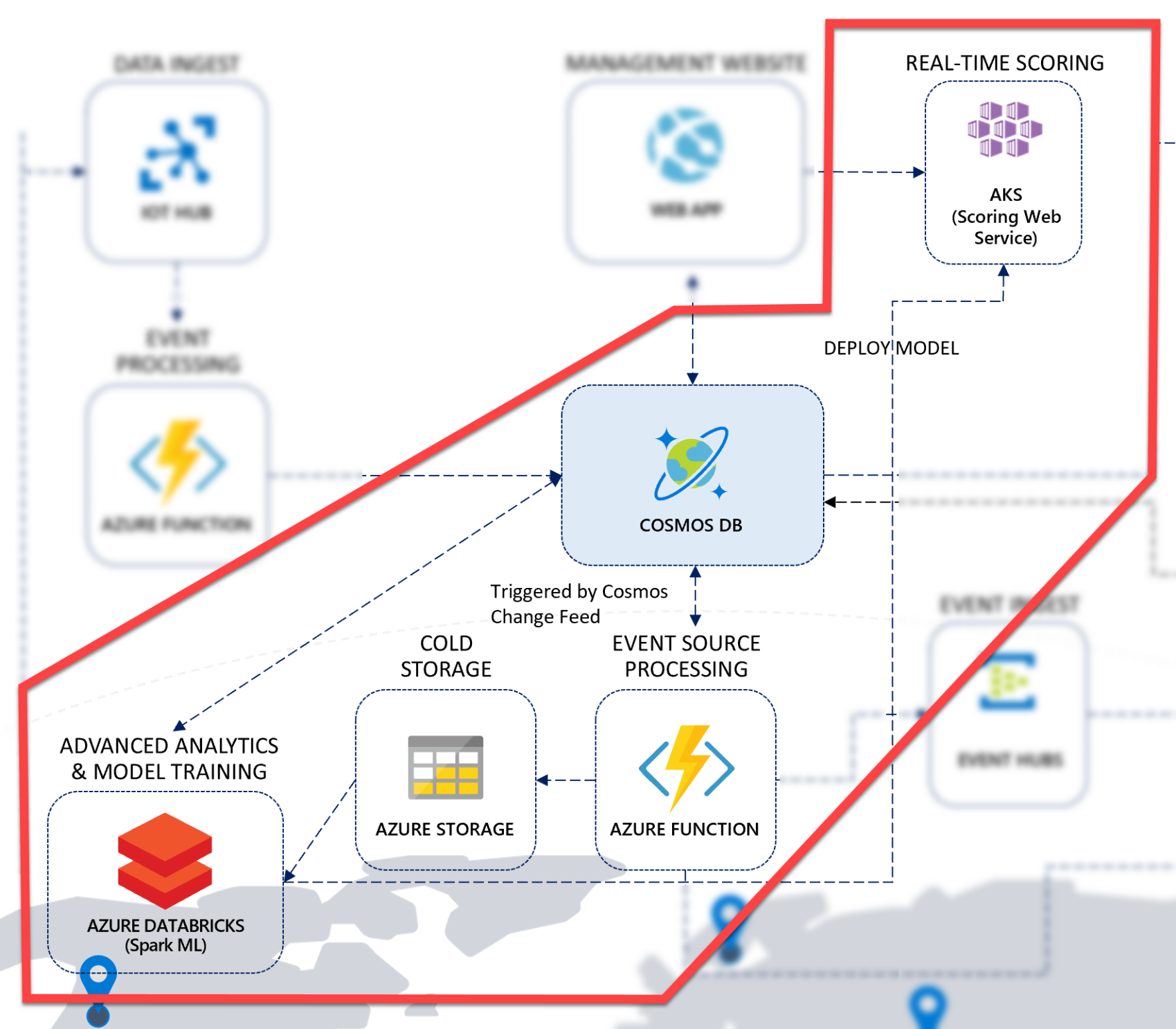
### Data management through the web app

The Web App provides a Fleet Management portal, allowing users to perform CRUD operations on vehicle data, make real-time battery failure predictions for a vehicle against the deployed machine learning model, and view consignments, packages, and trips. It connects to the Cosmos DB metadata container, using the [.NET SDK for Cosmos DB v3](https://github.com/Azure/azure-cosmos-dotnet-v3/).



## Machine Learning and advanced analytics

In this section, we cover the services and technologies used in the solution accelerator to perform advanced analytics and Machine Learning (ML).



[Azure Databricks](https://azure.microsoft.com/services/databricks/) is a fully-managed, cloud-based Big Data and Machine Learning platform, which empowers developers to accelerate AI and innovation by simplifying the process of building enterprise-grade production data applications. Built as a joint effort by the team that started Apache Spark and Microsoft, Azure Databricks provides data science and engineering teams with a single platform for Big Data processing and Machine Learning.

By combining the power of Databricks, an end-to-end, managed Apache Spark platform optimized for the cloud, with the enterprise security and scale of Microsoft's Azure platform, Azure Databricks makes it simple to run large-scale Spark workloads.

In the sample scenario for the solution accelerator, Contoso Auto wants to use the valuable data they are collecting from their vehicles to make predictions about the health of their fleet to reduce downtime due to maintenance-related issues. One of the predictions they would like to make is whether a vehicle's battery is likely to fail within the next 30 days, based on historical data. They would like to run a nightly batch process to identify vehicles that should be serviced, based on these predictions. They also want to have a way to make a prediction in real time when viewing a vehicle on their fleet management website.

To support this requirement, we use Apache Spark on Azure. Spark is a unified, big data and advanced analytics platform that enables data scientists and data engineers to explore and prepare large amounts of structured and unstructured data, then use that data to train, use, and deploy machine learning models at scale. We read and write to Cosmos DB, using the azure-cosmosdb-spark connector (<https://github.com/Azure/azure-cosmosdb-spark>).

Since the problem domain in this scenario focuses on vehicle telemetry, the ML model we created is trained on this type of data and its unique characteristics. The predictions it makes are specific to the problem as outlined by the business requirement. As such, you should evaluate and learn from the trained ML model, the Databricks notebooks, and the data used for this exercise, then apply what you learned to your specific problem domain. Now, let’s dig a little deeper into the problem and how we solved it.

### Problem overview

This scenario focuses on predictions for vehicle batteries reaching their rated number of cycles. The goal here is to avoid sudden failure of the car battery and schedule it to be replaced before it reaches its rated number of cycles. This reduces significantly the risk of collateral failures occurring as well. From a business point of view, the target is optimized car battery replacement schedules to minimize overall maintenance costs.

The connected vehicles report daily, information about the number of trips, duration of those trips and number of battery cycles used. Since reporting is performed on a daily basis, the battery age measured in days is also available as an input. The accumulated lifetime cycles used measure is also part of the dataset. As you can easily guess, the input data comes in the form of time series with the **Lifetime\_Cycles\_Used** field being the subject of prediction. Given the current time series for each battery, we are interested in forecasting the value of this field for the next 30 days to see whether the rated number of cycles will be exceeded or not.

This scenario details the development of a machine learning time series forecasting model. The model is trained on a dataset containing battery telemetry information for 1539 days (between 1/1/2013 and 3/19/2017) for one vehicle battery.

### Solution overview

We use the [Automated Machine Learning](https://docs.microsoft.com/en-us/azure/machine-learning/service/how-to-configure-auto-train) (autoML) capabilities of [Azure Machine Learning](https://docs.microsoft.com/en-us/azure/machine-learning/service/overview-what-is-azure-ml) to quickly train a model that can forecast the evolution of the **Lifetime\_Cycles\_Used** metric. We modeled the problem as a **Forecasting** problem where the goal of the trained model is to forecast the future evolution of a numerical indicator over several cycles (30 days in our case). The automML capabilities enable us to evaluate different algorithms and hyperparameters to get the best-trained model for the problem with minimum effort. The approach used in this example can be extended to various use cases that revolve around the need to predict the time series-based evolution of a numerical value.

We do not provide the notebook that trained the model using autoML. You may find the notebook used within the Azure Machine Learning Notebook Tutorials project on GitHub: <https://github.com/solliancenet/aml-notebook-tutorials>.

The solution accelerator contains two Databricks notebooks: The **Batch Scoring** notebook, and the **Model Deployment** notebook.

#### Batch Scoring notebook

The Batch Scoring notebook uses the pre-trained machine learning (ML) model to determine if the battery needs to be replaced on several vehicles within the next 30 days. The notebook performs the following actions:

1. Installs required Python libraries.
2. Connects to Azure Machine Learning (Azure ML).
3. Downloads a pre-trained ML model, saves it to Azure ML, then uses that model for batch scoring.
4. Uses the Cosmos DB Spark connector to retrieve completed Trips and Vehicle metadata from the metadata Cosmos DB container, prepares the data using SQL queries, then surfaces the data as temporary views.
5. Applies predictions against the data, using the pre-trained model.
6. Saves the prediction results in the Cosmos DB maintenance container for reporting purposes.

#### Model Deployment notebook

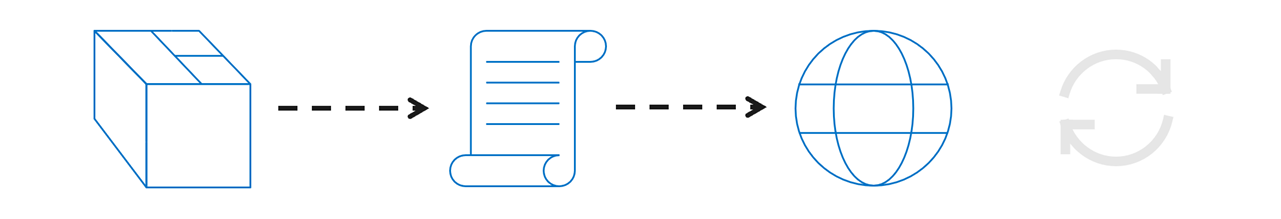
In addition to batch scoring, Contoso Auto would like to predict battery failures on-demand in real time for any given vehicle. They want to be able to call the model from their Fleet Management website when looking at a vehicle to predict whether that vehicle's battery may fail in the next 30 days.

The Model Deployment notebook deploys the pre-trained model to a web service hosted by Azure Container Instances (ACI), using your Azure ML workspace. While it is possible to deploy the model to a web service running in Azure Kubernetes Service (AKS), we deploy to ACI instead since doing so saves 10-20 minutes. However, once deployed, the process used to call the web service is the same, as are most of the steps to do the deployment.

#### General deployment process

Regardless of which framework you use to deploy a model, like Azure Machine Learning, you generally do the following to deploy a model:

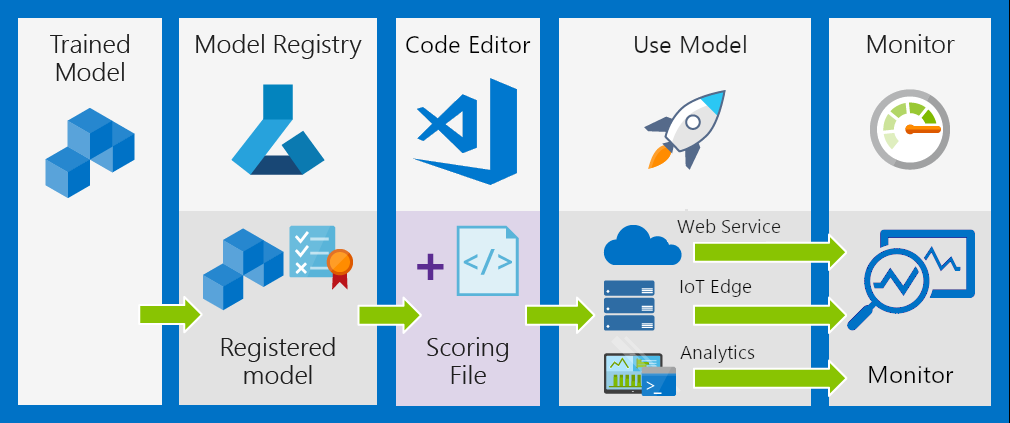
* Get the model file (any format)
* Create a scoring script (.py)
* Optionally create a schema file describing the web service input (.json)
* Create a real-time scoring web service
* Call the web service from your applications
* Repeat the process each time you re-train the model

[](https://github.com/solliancenet/Azure-Machine-Learning-Dev-Guide/blob/master/model-deployment/media/model-deployment-process.png)

This process is very time-consuming if done manually, especially in creating the real-time scoring web service.

#### Deploying with Azure Machine Learning service

Azure Machine Learning simplifies model deployments by providing tools that help automate the deployment steps. The series of steps using Azure Machine Learning fall in line with the deployment process above, but most of the work is done for you. You follow these steps as part of the deployment process, regardless of whether you are using your own models, or models you obtain from somewhere else.

[](https://github.com/solliancenet/Azure-Machine-Learning-Dev-Guide/blob/master/model-deployment/media/aml-deployment-process.png)

1. **Register the model** in a registry hosted in your Azure Machine Learning workspace
2. **Prepare to deploy** by creating a scoring file, specifying assets, usage, and a compute target
3. **Use** the model in a web service in the cloud, on an IoT device, or for analytics with Power BI
4. **Monitor and collect data**
5. **Update** deployment to use a new image

#### Available compute targets

You can use the following compute targets to host your web service deployment:

| **Compute target** | **Usage** | **Description** |
| --- | --- | --- |
| [Local web service](https://docs.microsoft.com/azure/machine-learning/service/how-to-deploy-and-where#local) | Testing/debug | Good for limited testing and troubleshooting. |
| [Azure Kubernetes Service (AKS)](https://docs.microsoft.com/azure/machine-learning/service/how-to-deploy-and-where#aks) | Real-time inference | Good for high-scale production deployments. Provides autoscaling, and fast response times. |
| [Azure Container Instances (ACI)](https://docs.microsoft.com/azure/machine-learning/service/how-to-deploy-and-where#aci) | Testing | Good for low scale, CPU-based workloads. |
| [Azure Machine Learning Compute](https://docs.microsoft.com/azure/machine-learning/service/how-to-run-batch-predictions) | (Preview) Batch inference | Run batch scoring on serverless compute. Supports normal and low-priority VMs. |
| [Azure IoT Edge](https://docs.microsoft.com/azure/machine-learning/service/how-to-deploy-and-where#iotedge) | (Preview) IoT module | Deploy & serve ML models on IoT devices. |

#### Deploying to Azure Container Instances

The **Model Deployment** notebook defines the following Python function to deploy the trained model to Azure Container Instances (ACI):

def deployModelAsWebService(ws, model\_name,

scoring\_script\_filename="scoring\_service.py",

conda\_packages=['numpy','pandas','scikit-learn','py-xgboost<=0.80'],

pip\_packages=['azureml-train-automl==1.0.60','inference-schema'],

conda\_file="dependencies.yml", runtime="python",

cpu\_cores=1, memory\_gb=1, tags={'name':'scoring'},

description='Forecast',

service\_name = "scoring"

):

# retrieve a reference to the already registered model

print("Retrieving model reference...")

registered\_model = Model(workspace=ws, name=model\_name)

# create a Conda dependencies environment file

print("Creating conda dependencies file locally...")

from azureml.core.conda\_dependencies import CondaDependencies

mycondaenv = CondaDependencies.create(conda\_packages=conda\_packages, pip\_packages=pip\_packages)

with open(conda\_file,"w") as f:

f.write(mycondaenv.serialize\_to\_string())

# create container image configuration

print("Creating container image configuration...")

from azureml.core.image import ContainerImage

image\_config = ContainerImage.image\_configuration(execution\_script = scoring\_script\_filename,

runtime = runtime,

conda\_file = conda\_file)

# create ACI configuration

print("Creating ACI configuration...")

from azureml.core.webservice import AciWebservice, Webservice

aci\_config = AciWebservice.deploy\_configuration(

cpu\_cores = cpu\_cores,

memory\_gb = memory\_gb,

tags = tags,

description = description)

# deploy the webservice to ACI

print("Deploying webservice to ACI...")

webservice = Webservice.deploy\_from\_model(

workspace=ws,

name=service\_name,

deployment\_config=aci\_config,

models = [registered\_model],

image\_config=image\_config

)

webservice.wait\_for\_deployment(show\_output=True)

return webservice

The function performs the following:

* Retrieves a reference to the trained ML model from the Azure Machine Learning workspace.
* Creates a Conda dependency file that references the Conda and pip packages and versions defined in the conda\_packages and pip\_packages parameters, respectively.
* Creates the Docker container image configuration by defining the scoring script (scoring\_service.py), Python runtime, and the Conda file.
* Creates the Azure Container Instances configuration. ACI is the deployment target for the web service that hosts the model for real-time scoring.
* Deploys the web service to ACI and returns a reference to the web service after the deployment completes.

The scoring file (scoring\_service.py) that gets added to the web service can be used for any of the valid deployment targets:

#save script to $deployment\_folder/scoring\_service.py

scoring\_service = """

import json

import pickle

import numpy as np

import pandas as pd

import azureml.train.automl

from sklearn.externals import joblib

from azureml.core.model import Model

from inference\_schema.schema\_decorators import input\_schema, output\_schema

from inference\_schema.parameter\_types.numpy\_parameter\_type import NumpyParameterType

from inference\_schema.parameter\_types.pandas\_parameter\_type import PandasParameterType

input\_sample = pd.DataFrame(data=[{"Date":"2013-01-01T00:00:00.000Z","Battery\_ID":0,"Battery\_Age\_Days":0,"Daily\_Trip\_Duration":67.8456075842}])

def init():

global model

# This name is model.id of model that we want to deploy deserialize the model file back

# into a sklearn model

model\_path = Model.get\_model\_path(model\_name = 'batt-cycles-6')

model = joblib.load(model\_path)

@input\_schema('data', PandasParameterType(input\_sample))

def run(data):

try:

#y\_query = data.pop('y\_query').values

#result = model.forecast(data, y\_query)

result = model.predict(data)

except Exception as e:

result = str(e)

return json.dumps({"error": result})

#forecast\_as\_list = result[0].tolist()

#index\_as\_df = result[1].index.to\_frame().reset\_index(drop=True)

#return json.dumps({"forecast": forecast\_as\_list, # return the minimum over the wire:

# "index": json.loads(index\_as\_df.to\_json(orient='records')) # no forecast and its featurized values

# })

return json.dumps({"result": result.tolist()})

"""

with open("scoring\_service.py", "w") as file:

file.write(scoring\_service)

#### Modify the deployment target to Azure Kubernetes Service (AKS)

If you prefer to deploy the model to AKS, replace the ACI portions of the **deployModelAsWebService** method as follows:

ACI portions to replace:

# create ACI configuration

print("Creating ACI configuration...")

from azureml.core.webservice import AciWebservice, Webservice

aci\_config = AciWebservice.deploy\_configuration(

cpu\_cores = cpu\_cores,

memory\_gb = memory\_gb,

tags = tags,

description = description)

# deploy the webservice to ACI

print("Deploying webservice to ACI...")

webservice = Webservice.deploy\_from\_model(

workspace=ws,

name=service\_name,

deployment\_config=aci\_config,

models = [registered\_model],

image\_config=image\_config

)

webservice.wait\_for\_deployment(show\_output=True)

Replace with the following to target AKS as the deployment target:

First, create an AKS cluster with the Azure Machine Learning SDK:

from azureml.core.compute import AksCompute, ComputeTarget

# Use the default configuration (you can also provide parameters to customize this)

prov\_config = AksCompute.provisioning\_configuration()

aks\_name = 'myaks'

# Create the cluster

aks\_target = ComputeTarget.create(workspace = ws,

name = aks\_name,

provisioning\_configuration = prov\_config)

# Wait for the create process to complete

aks\_target.wait\_for\_completion(show\_output = True)

Once you have created the AKS cluster, you can now deploy to it using the SDK:

from azureml.core.webservice import AksWebservice, Webservice

aks\_target = AksCompute(ws,"myaks")

deployment\_config = AksWebservice.deploy\_configuration(cpu\_cores = 1, memory\_gb = 1)

service = Model.deploy(ws, "aksservice", [model], inference\_config, deployment\_config, aks\_target)

service.wait\_for\_deployment(show\_output = True)

# Additional references

## Appendix A: What is Azure Machine Learning?

Before we delve into the components of Azure Machine Learning service and its tools, let us first describe the landscape of artificial intelligence, what it means to you as a data scientist, data engineer, or developer, and then how Azure ML fits into the landscape and addresses your needs.

What Artificial Intelligence/Machine Learning/Deep Learning mean to the data scientist, data engineer, and developer

Artificial Intelligence (AI) is a term that has been around since the 1950s to describe a set of processes that enable computers to think more like humans and to learn on their own. Computers have long been used to solve problems, but the field of AI aims to have machines use information from the past and use that data to inform future decisions. Machine learning is one example of a form of artificial intelligence that falls underneath the broader umbrella term of "AI".

In this article, we will focus on one of AI's most important disciplines: **machine learning (ML)** and its unique subdisciplines: supervised learning, unsupervised learning, reinforcement learning, deep learning (DL), and transfer learning. As machine learning becomes critical to the success of organizations, data scientists, data engineers, and developers are expected to expand their knowledge to meet these needs and achieve the requirements for digital transformation. AI becomes critical to an organization when the capabilities the technology provides is either at the core of that organization's business or enables the organization to innovate and gain a competitive edge in the marketplace. Let us continue by defining the primary fields of machine learning and deep learning.

### Machine learning

Machine learning is a data science technique used to extract predictions or patterns from statistical models by allowing computers to use existing data to forecast future outcomes, behaviors, and trends. This method of learning is accomplished without explicitly programming routines for computers to follow. There are too many variables to reliably account for every potential data point, logic flow, and decision to effectively program an application that is flexible enough to work with any given problem set and knowledge domain. Machine learning overcomes the terse and rigid constraints of explicitly programmed instructions by using special algorithms to find patterns and insights in a wide range of data. Machine learning systems harness this flexibility to use data from sources such as apps, sensors, historical data, networks, and devices to build its own logic to solve a problem or extract insight.

There are several approaches to machine learning that focus on different sets of problems. Some of the most widely used approaches are:

### Supervised learning

Supervised learning means that you have access to data where the outcomes are already known. You use this labeled data to teach the algorithm what conclusions to arrive to when you train your model. This means that the data should have target values that describe the prediction, such as whether a flight was delayed or information defined that are essential data points that can be used to make that prediction. In the case of predicting flight delays, this could include the origin and destination airports, date, airline, weather conditions, and whether the flight was delayed. You are responsible for selecting these data points, otherwise known as features, choosing a suitable algorithm, using a portion of the historical data for training the model, and a portion to test the model with data it has not yet seen. The trained algorithm, or model, can then be used to make predictions on new data that contain the same features.

Refer to the [Azure Machine Learning Algorithm Cheat Sheet](https://docs.microsoft.com/azure/machine-learning/studio/algorithm-cheat-sheet) to view a list of some of algorithms you can use to conduct supervised learning. All of the algorithms listed on the sheet, except for K-means clustering, are used in supervised learning, with the regression and classification categories of algorithms used most often.

### Unsupervised learning

The majority of data generated in the world today is unlabeled. Labeling data for training a machine learning model can be very costly, and even the best-curated data sets have only thousands of labels. Since these labels are crucial to supervised learning, an initial investment must be made to apply them. When you have data that is neither classified nor labeled, you could use an unsupervised algorithm to act on the information without guidance. One goal of the algorithm is to group data samples according to patterns in similarities and distances among them. This could mean grouping the data into clusters, as K-means does, or finding different ways of looking at complex data so that it appears more uncomplicated than in its unstructured form. Unlike supervised learning, you do not provide any prior teaching, which leaves the machine to find the hidden structure in unlabeled data by itself.

One example of unsupervised learning is used in healthcare. Analysts detect causality and identify correlations humans may miss by [inputting health data like blood pressure, heart rate, weight, and prescription data](http://people.csail.mit.edu/dsontag/courses/mlhc_summer18/day2/causal_inference.pdf) into an unsupervised learning algorithm.

Other applications for unsupervised learning include:

* *Data drift*: You initially train your machine learning models on data whose characteristics may change over time. This drift can lead to lower-quality predictions over time. One way to address this problem is to create probability distributions using unsupervised learning to assess how different new data is from the training data. If there is a significant difference, then the model should be retrained with current data.
* *Outliers*: Use unsupervised learning to detect outliers within a data set. This detection can lead to improved training by ignoring or removing outliers and addressing them separately, as well as other applications such as anomaly detection.
* *Overfitting*: One of the challenges of training a machine learning algorithm is overfitting to the training data, leading to poor performance on data it has never seen. This is oftentimes caused by extracting too much from noise in the training data and missing the signal, or essence, of the information. In these cases, unsupervised learning can be used as a *regulator* to reduce the complexity of the machine learning algorithm by removing excess noise from the training data. This unsupervised pretraining transforms the original data, and this generated data is fed into the supervised learning algorithm. The algorithm has less noise to contend with, allowing it to capture more of the signal and improve its generalization error.

When deciding between supervised and unsupervised learning algorithms, the general rule is to use supervised learning when you have narrowly-defined tasks for which you have distinct patterns that do not change much over time. The datasets you use in this case should be sufficiently large and have well-defined labels. However, if the problem you are trying to solve has patterns that are continually changing or unknown, and you do not have large, labeled datasets, unsupervised learning will give you the best outcome.

### Reinforcement learning

Reinforcement learning takes a more organic, almost human approach to learning. This class of algorithms interacts with a simulated or real environment to explore different strategies that result in a maximum reward. The choices are reinforced by either a favorable or unfavorable outcome in the form of a reward signal, sometimes called a *reinforcement signal*. Each choice the algorithm, or agent, makes when it encounters a new data point is impacted by how great the reward was in its last decision. If the action led to lower performance compared to an agent that acts optimally, this leads to the notion of regret. In effect, the algorithm continually modifies its strategy with the driving goal to achieve the highest long-term reward.

An optimal reinforcement learning algorithm addresses the explore vs. exploit tradeoff when making decisions each step of the way. It will explore low-value options, even if selecting one of these options leads to a low short-term reward so that it can gain a higher long-term reward. This is because the optimized agent reasons about the long-term consequences of its actions when making a decision. Instead of choosing to explore low-value options, the algorithm might choose to exploit the knowledge it has gained to maximize the reward, given its current learnings.

Reinforcement learning is highly prevalent in robotics, where the set of sensor readings at one point in time is a data point the algorithm uses to choose the robot's next action. Other examples include learning how to master chess or how to drive a vehicle without crashing into obstacles. Sometimes the reinforcement comes from user interaction. For instance, reinforcement learning can be used in product recommendations where shoppers have the chance to say in the user interface to either "show more like this" or "do not show me any more products like this".

### Deep learning

One of the most successful classes of machine learning algorithms in recent years is the neural network, or deep neural network (DNN). These algorithms encompass the deep learning subdiscipline of machine learning, as it uses what is called a neural network architecture that was originally inspired by how a brain works, rather than using traditional statistical frameworks. This architecture contains several stacked layers on top of each other, with higher levels of abstraction occurring within each layer. This layering is where the term "deep" comes from. The more layers that you use, the deeper the neural network architecture, and the more abstract interpretations of data can be made. This approach is especially useful when working with data without structured attributes or features, leaving it up to the algorithm to come up with its own interpretation of what the input represents. In comparison to most conventional machine learning techniques, deep learning requires massive amounts of compute power, more training time, and enormous datasets. Deep learning techniques have been around for many years, but only because of recent breakthroughs in the size of available datasets and computational resources has it become possible to apply them to hard, real-world problems.

Some applications of deep learning include speech recognition, image and object recognition, and Natural Language Processing (NLP). These capabilities are achieved through the use of neural network architectures, such as convolutional, recurrent neural networks, and multilayer perceptron.

### Transfer learning

Transfer learning is a deep learning process that allows you to reuse a model that has already been pre-trained to solve a similar problem, potentially saving you significant time and resources. This is done by retraining all or some of the layers of the existing model so that it solves your new problem.

To give you an example of how transfer learning works, imagine that you have a requirement to train a new image classifier to recognize a handful of categories, such as bicycles, tennis shoes, and skateboards. You could either train a new model from scratch or start with an existing model that has already been trained to recognize large classes of images. One such pre-trained model exists that you can use in this example. It is a [deep convolutional neural network](https://en.wikipedia.org/wiki/Convolutional_neural_network) model, named [Inception](https://storage.googleapis.com/download.tensorflow.org/models/inception5h.zip), that was trained to recognize and classify images to identify thousands of objects like surfboards, pizza, and giraffes. Because of the type of algorithm used and the amount of training conducted to build this model, the lower image feature layers recognize simple features of the images, such as edges, and the higher layers can recognize complex features, such as shapes. When you use this existing model that is pre-trained with thousands of images as a starting point, as in this example, and retrain just the final layer of the model, this is called transfer learning. You have cut out extensive training time by using a much smaller image set to teach your new model to recognize a narrow set of categories. In effect, you transferred the Inception model's ability to classify a wide range of images to the narrow set of categories of your new image classifier. This way, you can drastically reduce the time and resources used to train the new model because you do not need to train all of the layers of the neural net.

### The Microsoft AI and ML spectrum

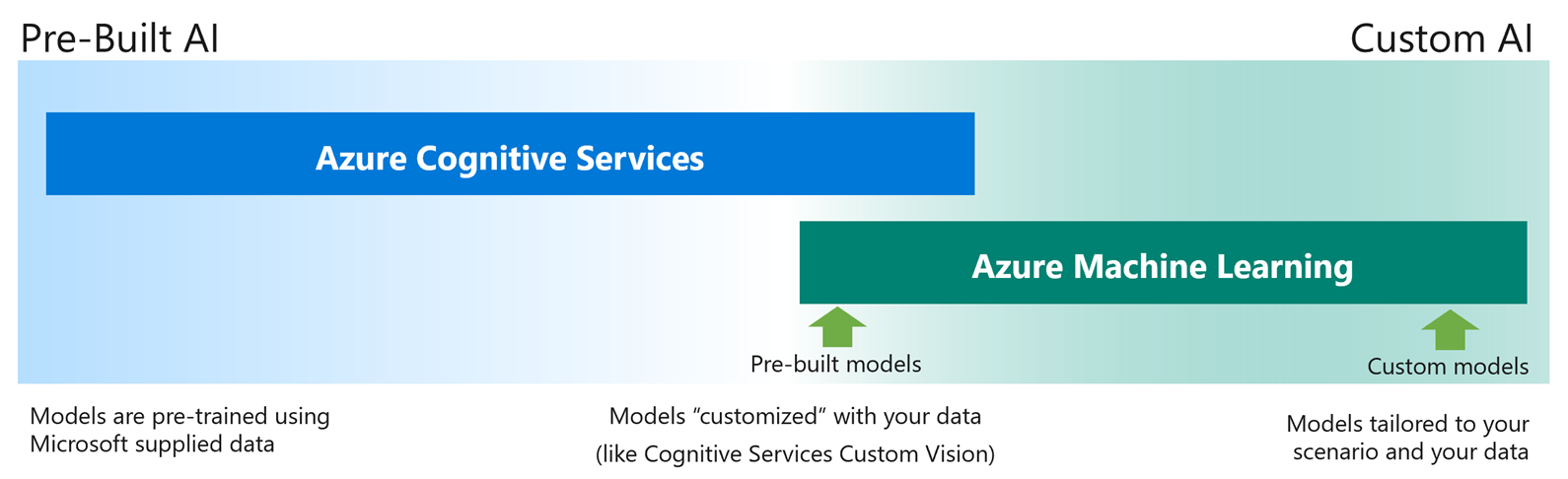
Microsoft is a leader in AI and in providing the tools and services needed to help organizations of all sizes benefit from the capabilities it provides. Microsoft CEO, Satya Nadella, summarizes Microsoft's vision for AI:

"It's not enough just to sort of have AI capability that we can exercise—you also need the **ability to democratize it** so that **every business can truly benefit from it**... That to me is our identity around AI." *–*[*Satya Nadella*](https://www.forbes.com/sites/bobevans1/2018/06/04/microsoft-ceo-satya-nadella-on-the-extraordinary-potential-of-ai/#986818e162ff)

Microsoft provides a vast number of technologies that you, as a developer, data scientist, or data engineer, can use to harness the power of AI and machine learning (ML). The number of options can be overwhelming if you are not a seasoned AI expert and Azure professional, and know which services to use when so many tend to have capabilities that overlap. A few of these options are Azure Machine Learning, Cognitive Services, Azure Databricks, HDInsight, SQL Server, ML.NET, Azure Batch, and Power BI.

You can view these offerings from a very high level in the context of a spectrum of AI and ML choices. This spectrum begins with pre-built and easily accessible models that require no training or data science expertise, provided by Microsoft, and ends with entirely custom models trained, evaluated, and deployed by developers, sysadmins, data engineers, or data scientists.

The following diagram helps visualize the spectrum of choices:

[](https://github.com/solliancenet/Azure-Machine-Learning-Dev-Guide/blob/master/intro/media/ai-spectrum.png)

The left side of this spectrum, which includes the pre-built models, is provided by [Azure Cognitive Services](https://azure.microsoft.com/services/cognitive-services/). Cognitive Services also covers the middle of the spectrum, which still uses pre-built models, but they are customizable with your own data, enabling you to train the model without needing to program or host the model in your own environment. Currently, this level of customization can be enabled by the Vision and Speech APIs. The middle of the spectrum also is where you can use pre-built models with transfer learning to customize them for your needs. This left to middle range of the spectrum may be sufficient for your business needs, and you can easily consume them through simple REST calls from your applications. The pre-trained machine learning models are built, maintained, and trained by Microsoft to cover a broad range of scenarios. However, you might encounter a situation where your challenges are too specific to effectively use the models Cognitive Services provides, even the customizable ones. When this happens, you need to set up an environment where you can write custom code, access your data, and use 3rd-party libraries to solve your problem. There are a plethora of libraries you can use to aid your development, and a large number of pre-built models you can use to simplify the code you need to write. However, sometimes, even these options are not sufficient for your problem. When this happens, you need to explore your options on the right-hand side of the spectrum to create custom models tailored to your scenario and your data. It is on this side of the spectrum where you write most of the code needed to solve your problem.

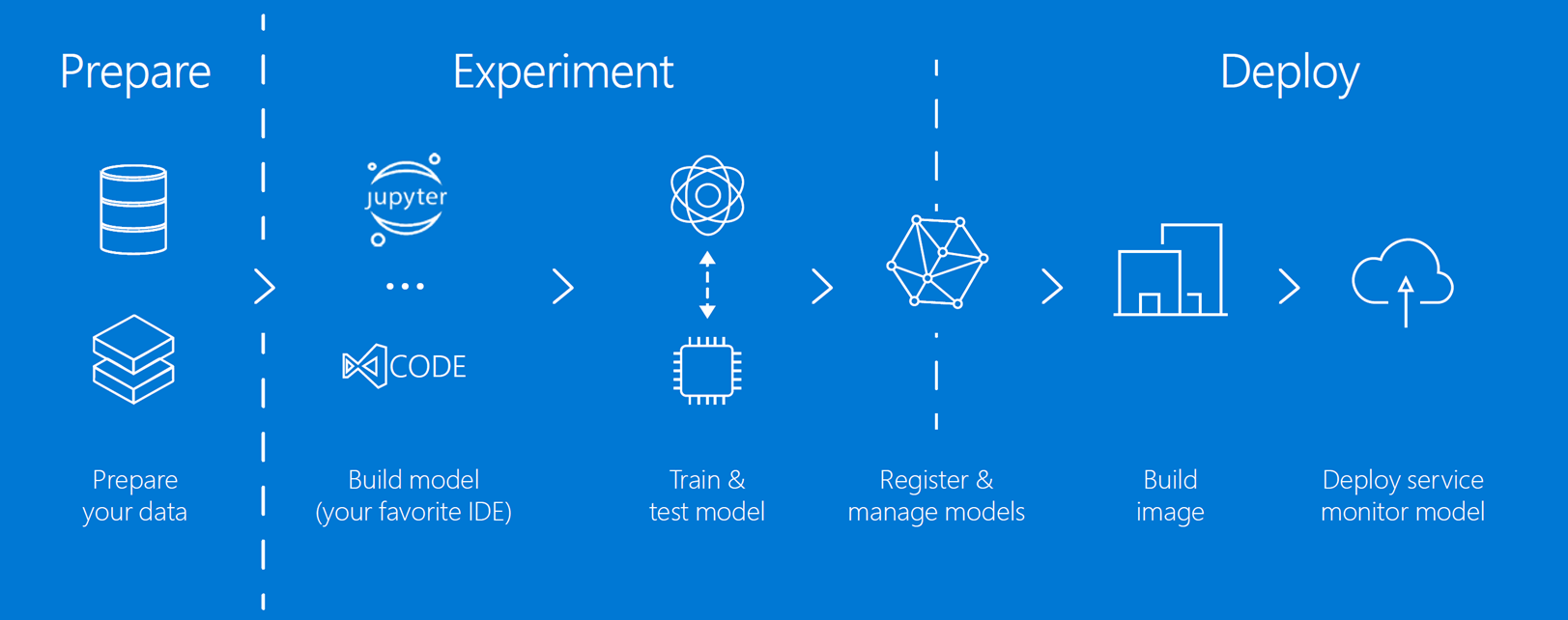
This right-hand side of the spectrum is where [**Azure Machine Learning**](https://docs.microsoft.com/azure/machine-learning/service/overview-what-is-azure-ml) comes to play.

### How the Azure Machine Learning fits in the picture and addresses your needs

[Azure Machine Learning](https://docs.microsoft.com/azure/machine-learning/service/overview-what-is-azure-ml) is a fully managed cloud service used to train, deploy, and manage machine learning models at scale. Given the variety of services within Azure that you can choose from to create AI solutions, Azure Machine Learning service is the one platform in Azure that "glues" together the entire custom AI/ML development story. When you couple the rich set of tools with the service, you have everything you need to get started with experimenting and creating AI solutions on the right-hand side of the AI spectrum.

Azure Machine Learning service fully supports open-source technologies so that you can use tens of thousands of open-source Python packages such as TensorFlow, PyTorch, MXNet, and scikit-learn. All you need is a Python-enabled environment to get started with setting up and using Azure Machine Learning through its [Python SDK](https://docs.microsoft.com/python/api/overview/azure/ml/intro?view=azure-ml-py). Powerful tools are also available, such as [notebook VMs](https://docs.microsoft.com/azure/machine-learning/service/how-to-configure-environment#notebookvm), [Azure notebooks](https://notebooks.azure.com/), [Jupyter notebooks](http://jupyter.org/), or the [Azure Machine Learning for Visual Studio Code](https://aka.ms/vscodetoolsforai) extension to make it easy to explore and transform data, and then train and deploy models. Azure Machine Learning service includes features that automate model generation and tuning with ease, efficiency, and accuracy.

Use Azure Machine Learning to train, deploy, and manage machine learning models using Python and CLI at cloud scale. For a low-code or no-code option, use the interactive, [visual interface](https://docs.microsoft.com/azure/machine-learning/service/ui-quickstart-run-experiment) to easily and quickly build, test, and deploy models using pre-built machine learning algorithms.

[](https://github.com/solliancenet/Azure-Machine-Learning-Dev-Guide/blob/master/intro/media/steps-to-using-azureml.png)

As shown in the diagram above, Azure Machine Learning service helps you perform each step of the data science process. These steps include:

* **Data preparation**: Prepare your data for model training, using notebooks that can run on [compute resources](https://github.com/solliancenet/Azure-Machine-Learning-Dev-Guide/blob/master/intro/tools.md) ([Azure Machine Learning Compute](https://docs.microsoft.com/azure/machine-learning/service/how-to-set-up-training-targets#amlcompute), Azure Databricks, VMs, etc.) to explore and prepare your data quickly and cost-effectively by autoscaling using CPUs and GPUs. Get started with a [tutorial that uses the data prep package from the Azure Machine Learning SDK](https://docs.microsoft.com/azure/machine-learning/service/tutorial-data-prep).
* **Experimentation**: Use notebook VMs, Azure notebooks, Jupyter notebooks, or Visual Studio Code with the Azure Machine Learning extension to explore your data, transform it, and train and test your ML models. You can start training on your local machine and then scale out to the cloud, using one of the compute resources listed above. Also, the automated machine learning feature can improve productivity by using automatic model selection and hyperparameter tuning to accelerate the model training process. With automated machine learning, it's possible to evaluate the importance of model features and measure the relationships between those features and a model's outputs automatically. This allows a data scientist to identify potential areas of improvement more quickly.
* **Model management**: Use the Azure Machine Learning SDK to register your trained models in your workspace. If you have a model that you store in multiple files, you can register it as a single model as well. Model registration allows you to store and version your models, helping you organize and keep track of your trained models. Azure Machine Learning uses the [Machine Learning Operations (MLOps) approach](https://docs.microsoft.com/azure/machine-learning/service/concept-model-management-and-deployment), which improves the quality and consistency of your ML solutions.
* **Deployment**: The Azure Machine Learning SDK makes it easy to package your trained models in Docker containers and deploy them to [AML Compute](https://docs.microsoft.com/azure/machine-learning/service/how-to-set-up-training-targets#amlcompute), [Azure Kubernetes Service](https://docs.microsoft.com/azure/aks/intro-kubernetes) (AKS), [Azure Container Instances](https://docs.microsoft.com/azure/container-instances/container-instances-overview) (ACI), and [IoT Edge](https://docs.microsoft.com/azure/iot-edge/about-iot-edge).

Data scientists can use Azure Machine Learning to build their custom machine learning and deep learning models, then register them to run and track experiments that are associated with each version of the model as they continue to refine and retrain their models. The deployment options provided by Azure ML help them deploy new versions of their models without needing to take down existing services hosting their models, disrupting current users and applications.

Developers and data engineers benefit from improved productivity with autoscaling compute and DevOps for machine learning. The Azure Machine Learning SDK makes it easy to script one-click deployments to the cloud and the edge, and use DevOps tools to automate that process if desired. Data engineers can harness the power of Apache Spark from Azure notebooks and Jupyter notebooks to perform data exploration and preparation at scale. All of these capabilities are accessible from your favorite Python environment using the latest open-source frameworks, such as TensorFlow, PyTorch, scikit-learn, and MXNet.

## Appendix B: Logic Apps

Description:

Connectors:

Connectors provide quick access from Azure Logic Apps to events, data, and actions across other apps, services, systems, protocols, and platforms. By using connectors in your logic apps, you expand the capabilities for your cloud and on-premises apps to perform tasks with the data that you create and already have.

While Logic Apps offers [hundreds of connectors](https://docs.microsoft.com/connectors), this article describes popular and more commonly used connectors that are successfully used by thousands of apps and millions of executions for processing data and information. To find the full list of connectors and each connector's reference information, such as triggers, actions, and limits, review the connector reference pages under [Connectors overview](https://docs.microsoft.com/connectors). Also, learn more about [triggers and actions](https://docs.microsoft.com/en-us/azure/connectors/apis-list#triggers-actions), [Logic Apps pricing model](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-pricing), and [Logic Apps pricing details](https://azure.microsoft.com/pricing/details/logic-apps/).

 Note

To integrate with a service or API that doesn't have a connector, you can either directly call the service over a protocol such as HTTP or create a [**custom connector**](https://docs.microsoft.com/en-us/azure/connectors/apis-list#custom).

Connectors are available either as built-in triggers and actions or as managed connectors:

* **[Built-ins](https://docs.microsoft.com/en-us/azure/connectors/apis-list" \l "built-ins)**: These built-in triggers and actions are "native" to Azure Logic Apps and help you create logic apps that run on custom schedules, communicate with other endpoints, receive and respond to requests, and call Azure functions, Azure API Apps (Web Apps), your own APIs managed and published with Azure API Management, and nested logic apps that can receive requests. You can also use built-in actions that help you organize and control your logic app's workflow, and also work with data.
* **Managed connectors**: Deployed and managed by Microsoft, these connectors provide triggers and actions for accessing cloud services, on-premises systems, or both, including Office 365, Azure Blob Storage, SQL Server, Dynamics, Salesforce, SharePoint, and more. Some connectors specifically support business-to-business (B2B) communication scenarios and require an [integration account](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-enterprise-integration-create-integration-account) that's linked to your logic app. Before using certain connectors, you might have to first create connections, which are managed by Azure Logic Apps.

For example, if you're using Microsoft BizTalk Server, your logic apps can connect to and communicate with your BizTalk Server by using the [BizTalk Server on-premises connector](https://docs.microsoft.com/en-us/azure/connectors/apis-list#on-premises-connectors). You can then extend or perform BizTalk-like operations in your logic apps by using the [integration account connectors](https://docs.microsoft.com/en-us/azure/connectors/apis-list#integration-account-connectors).

Connectors are classified as either Standard or Enterprise. [Enterprise connectors](https://docs.microsoft.com/en-us/azure/connectors/apis-list#enterprise-connectors) provide access to enterprise systems such as SAP, IBM MQ, and IBM 3270 for an additional cost. To determine whether a connector is Standard or Enterprise, see the technical details in each connector's reference page under [Connectors overview](https://docs.microsoft.com/connectors).

You can also identify connectors by using these categories, although some connectors can cross multiple categories. For example, SAP is an Enterprise connector and an on-premises connector:

|  |  |
| --- | --- |
| [**Managed API connectors**](https://docs.microsoft.com/en-us/azure/connectors/apis-list#managed-api-connectors) | Create logic apps that use services such as Azure Blob Storage, Office 365, Dynamics, Power BI, OneDrive, Salesforce, SharePoint Online, and many more. |
| [**On-premises connectors**](https://docs.microsoft.com/en-us/azure/connectors/apis-list#on-premises-connectors) | After you install and set up the [on-premises data gateway](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-gateway-connection), these connectors help your logic apps access on-premises systems such as SQL Server, SharePoint Server, Oracle DB, file shares, and others. |
| [**Integration account connectors**](https://docs.microsoft.com/en-us/azure/connectors/apis-list#integration-account-connectors) | Available when you create and pay for an integration account, these connectors transform and validate XML, encode and decode flat files, and process business-to-business (B2B) messages with AS2, EDIFACT, and X12 protocols. |

For the full list of connectors and each connector's reference information, such as actions and any triggers, which are defined by an OpenAPI (formerly Swagger) description, plus any limits, you can find the full list under the [Connectors overview](https://docs.microsoft.com/en-us/connectors/). For pricing information, see [Logic Apps pricing model](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-pricing), and [Logic Apps pricing details](https://azure.microsoft.com/pricing/details/logic-apps/).

### Built-ins

Logic Apps provides built-in triggers and actions so you can create schedule-based workflows, help your logic apps communicate with other apps and services, control the workflow through your logic apps, and manage or manipulate data.

|  |  |  |  |
| --- | --- | --- | --- |
| [**Schedule**](https://docs.microsoft.com/en-us/azure/connectors/connectors-native-recurrence) | - Run your logic app on a specified schedule, ranging from basic to complex recurrences, with the **Recurrence** trigger.  - Pause your logic app for a specified duration with the **Delay** action.  - Pause your logic app until the specified date and time with the **Delay until** action. | [**HTTP**](https://docs.microsoft.com/en-us/azure/connectors/connectors-native-http) | Communicate with any endpoint over HTTP with both triggers and actions for HTTP, HTTP + Swagger, and HTTP + Webhook. |
| [**Request**](https://docs.microsoft.com/en-us/azure/connectors/connectors-native-reqres) | - Make your logic app callable from other apps or services, trigger on Event Grid resource events, or trigger on responses to Azure Security Center alerts with the **Request** trigger.  - Send responses to an app or service with the **Response** action. | [**Batch**](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-batch-process-send-receive-messages) | - Process messages in batches with the **Batch messages** trigger.  - Call logic apps that have existing batch triggers with the **Send messages to batch** action. |
| [**Azure Functions**](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-azure-functions) | Call Azure functions that run custom code snippets (C# or Node.js) from your logic apps. | [**Azure API Management**](https://docs.microsoft.com/en-us/azure/api-management/get-started-create-service-instance) | Call triggers and actions defined by your own APIs that you manage and publish with Azure API Management. |
| [**Azure App Services**](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-custom-hosted-api) | Call Azure API Apps, or Web Apps, hosted on Azure App Service. The triggers and actions defined by these apps appear like any other first-class triggers and actions when Swagger is included. | [**Azure Logic Apps**](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-http-endpoint) | Call other logic apps that start with a Request trigger. |

### Control workflow

Logic Apps provides built-in actions for structuring and controlling the actions in your logic app's workflow:

|  |  |  |  |
| --- | --- | --- | --- |
| [**Condition**](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-control-flow-conditional-statement) | Evaluate a condition and run different actions based on whether the condition is true or false. | [**For each**](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-control-flow-loops#foreach-loop) | Perform the same actions on every item in an array. |
| [**Scope**](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-control-flow-run-steps-group-scopes) | Group actions into *scopes*, which get their own status after the actions in the scope finish running. | [**Switch**](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-control-flow-switch-statement) | Group actions into *cases*, which are assigned unique values except for the default case. Run only that case whose assigned value matches the result from an expression, object, or token. If no matches exist, run the default case. |
| [**Terminate**](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-workflow-actions-triggers#terminate-action) | Stop an actively running logic app workflow. | [**Until**](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-control-flow-loops#until-loop) | Repeat actions until the specified condition is true or some state has changed. |

### Manage or manipulate data

Logic Apps provides built-in actions for working with data outputs and their formats:

|  |  |
| --- | --- |
| [**Data Operations**](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-perform-data-operations) | Perform operations with data:  - **Compose**: Create a single output from multiple inputs with various types. - **Create CSV table**: Create a comma-separated-value (CSV) table from an array with JSON objects. - **Create HTML table**: Create an HTML table from an array with JSON objects. - **Filter array**: Create an array from items in another array that meet your criteria. - **Join**: Create a string from all items in an array and separate those items with the specified delimiter. - **Parse JSON**: Create user-friendly tokens from properties and their values in JSON content so you can use those properties in your workflow. - **Select**: Create an array with JSON objects by transforming items or values in another array and mapping those items to specified properties. |
| **Date Time** | Perform operations with timestamps:  - **Add to time**: Add the specified number of units to a timestamp. - **Convert time zone**: Convert a timestamp from the source time zone to the target time zone. - **Current time**: Return the current timestamp as a string. - **Get future time**: Return the current timestamp plus the specified time units. - **Get past time**: Return the current timestamp minus the specified time units. - **Subtract from time**: Subtract a number of time units from a timestamp. |
| [**Variables**](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-create-variables-store-values) | Perform operations with variables:  - **Append to array variable**: Insert a value as the last item in an array stored by a variable. - **Append to string variable**: Insert a value as the last character in a string stored by a variable. - **Decrement variable**: Decrease a variable by a constant value. - **Increment variable**: Increase a variable by a constant value. - **Initialize variable**: Create a variable and declare its data type and initial value. - **Set variable**: Assign a different value to an existing variable. |

### Managed API connectors

Logic Apps provides these popular Standard connectors for automating tasks, processes, and workflows with these services or systems.

|  |  |  |  |
| --- | --- | --- | --- |
| [**Azure Service Bus**](https://docs.microsoft.com/en-us/azure/connectors/connectors-create-api-servicebus) | Manage asynchronous messages, sessions, and topic subscriptions with the most commonly used connector in Logic Apps. | [**SQL Server**](https://docs.microsoft.com/en-us/azure/connectors/connectors-create-api-sqlazure) | Connect to your SQL Server on premises or an Azure SQL Database in the cloud so you can manage records, run stored procedures, or perform queries. |
| [**Office 365 Outlook**](https://docs.microsoft.com/en-us/azure/connectors/connectors-create-api-office365-outlook) | Connect to your Office 365 email account so you can create and manage emails, tasks, calendar events and meetings, contacts, requests, and more. | [**Azure Blob Storage**](https://docs.microsoft.com/en-us/azure/connectors/connectors-create-api-azureblobstorage) | Connect to your storage account so you can create and manage blob content. |
| [**SFTP**](https://docs.microsoft.com/en-us/azure/connectors/connectors-create-api-sftp) | Connect to SFTP servers you can access from the internet so you can work with your files and folders. | [**SharePoint Online**](https://docs.microsoft.com/en-us/azure/connectors/connectors-create-api-sharepointonline) | Connect to SharePoint Online so you can manage files, attachments, folders, and more. |
| [**Dynamics 365 CRM Online**](https://docs.microsoft.com/en-us/azure/connectors/connectors-create-api-crmonline) | Connect to your Dynamics 365 account so you can create and manage records, items, and more. | [**FTP**](https://docs.microsoft.com/en-us/azure/connectors/connectors-create-api-ftp) | Connect to FTP servers you can access from the internet so you can work with your files and folders. |
| [**Salesforce**](https://docs.microsoft.com/en-us/azure/connectors/connectors-create-api-salesforce) | Connect to your Salesforce account so you can create and manage items such as records, jobs, objects, and more. | [**Twitter**](https://docs.microsoft.com/en-us/azure/connectors/connectors-create-api-twitter) | Connect to your Twitter account so you can manage tweets, followers, your timeline, and more. Save your tweets to SQL, Excel, or SharePoint. |
| [**Azure Event Hubs**](https://docs.microsoft.com/en-us/azure/connectors/connectors-create-api-azure-event-hubs) | Consume and publish events through an Event Hub. For example, get output from your logic app with Event Hubs, and then send that output to a real-time analytics provider. | [**Azure Event** **Grid**](https://docs.microsoft.com/en-us/azure/event-grid/monitor-virtual-machine-changes-event-grid-logic-app) | Monitor events published by an Event Grid, for example, when Azure resources or third-party resources change. |

### On-premises connectors

Here are some commonly used Standard connectors that Logic Apps provides for accessing data and resources in on-premises systems. Before you can create a connection to an on-premises system, you must first [download, install, and set up an on-premises data gateway](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-gateway-connection). This gateway provides a secure communication channel without having to set up the necessary network infrastructure.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **BizTalk** **Server** | [**File System**](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-using-file-connector) | [**IBM DB2**](https://docs.microsoft.com/en-us/azure/connectors/connectors-create-api-db2) | [**IBM** **Informix**](https://docs.microsoft.com/en-us/azure/connectors/connectors-create-api-informix) | **MySQL** |
| [**Oracle DB**](https://docs.microsoft.com/en-us/azure/connectors/connectors-create-api-oracledatabase) | **PostgreSQL** | [**SharePoint Server**](https://docs.microsoft.com/en-us/azure/connectors/connectors-create-api-sharepointserver) | [**SQL Server**](https://docs.microsoft.com/en-us/azure/connectors/connectors-create-api-sqlazure) | **Teradata** |

### Integration account connectors

Logic Apps provides Standard connectors for building business-to-business (B2B) solutions with your logic apps when you create and pay for an [integration account](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-enterprise-integration-create-integration-account), which is available through the Enterprise Integration Pack (EIP) in Azure. With this account, you can create and store B2B artifacts such as trading partners, agreements, maps, schemas, certificates, and so on. To use these artifacts, associate your logic apps with your integration account. If you currently use BizTalk Server, these connectors might seem familiar already.

|  |  |  |  |
| --- | --- | --- | --- |
| [**AS2 decoding**](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-enterprise-integration-as2) | [**AS2 encoding**](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-enterprise-integration-as2) | [**EDIFACT decoding**](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-enterprise-integration-edifact-decode) | [**EDIFACT encoding**](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-enterprise-integration-edifact-encode) |
| [**Flat file decoding**](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-enterprise-integration-flatfile) | [**Flat file encoding**](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-enterprise-integration-flatfile) | [**Integration account**](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-enterprise-integration-metadata) | [**Liquid** **transforms**](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-enterprise-integration-liquid-transform) |
| [**X12 decoding**](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-enterprise-integration-x12-decode) | [**X12 encoding**](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-enterprise-integration-x12-encode) | [**XML** **transforms**](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-enterprise-integration-transform) | [**XML validation**](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-enterprise-integration-xml-validation) |

### Enterprise connectors

Logic Apps provides these Enterprise connectors for accessing enterprise systems, such as SAP and IBM MQ:

|  |  |  |
| --- | --- | --- |
| [**IBM 3270**](https://docs.microsoft.com/en-us/azure/connectors/connectors-run-3270-apps-ibm-mainframe-create-api-3270) | [**IBM MQ**](https://docs.microsoft.com/en-us/azure/connectors/connectors-create-api-mq) | [**SAP**](https://docs.microsoft.com/en-us/azure/logic-apps/logic-apps-using-sap-connector) |

## Appendix C: IoT device and solution design considerations

### Device intelligence

Connected devices aim to form intelligent systems. A key question is how intelligent devices should be versus how intelligent the system as a whole should be. The answer will be different based on the specific device purpose, design, available computing resources, and power; however, there a set of common trade-offs for IoT systems to consider.

Designing a device happens at the beginning of its life cycle. Design failures at this stage can be impossible or very costly to correct after the device is manufactured, though some device behavior can be changed through firmware/software updates or through a configuration change. Software changes are much easier than changing or replacing hardware, so designing remote software update capabilities for devices is helpful.

Even if a device has software update capabilities, managing updates of potentially millions of edge components is far more complex than updating a centralized solution backend. In general, more intelligence on the edge equates to potentially more software updates of edge components at higher frequencies, while more intelligence on the solution’s backend means the maintenance can be performed in a centralized fashion. Without question, having more intelligence on the backend will most likely increase the dependency of edge components, although when designed properly they should be able to perform autonomously even without online connection to the backend.

Regardless of the functional capabilities of the devices, centralizing the security of the software operations in the backend typically allows for better security controls across the entire system (especially when devices are in an untrusted zone).

During the lifetime of an IoT solution, multiple device types of different generations and versions will potentially be connected to the system. Even if an IoT solution starts with one device type, heterogeneity of the deployed device population should be expected. With increased heterogeneity the maintenance of the edge components is expected to increase significantly, while maintenance of the backend software shouldn’t be impacted to the same extent.

Maintaining simple, stable interfaces between the device and the backend will help in the long run.

In general, changes are gradually easier when moving from device hardware, to device/edge software, to the cloud backend. For this reason, it’s always a good practice to start designing in this sequence—that is, to design for devices first. The available power on the device, computing resources, as well as the choice of communication technology will affect how and when devices communicate with the service. In many cases certain processing will need to happen on the edge, such as when guaranteed response times are needed, or to perform filtering of data sent to the backend.

Having less intelligence on the devices may increase the dependency on the cloud backend but helps improve the agility of the system and reduces maintenance and operations cost.

These trade-offs should be considered in the specific context and business requirements of an IoT solution and may vary from scenario to scenario.

### Device telemetry

The type and frequency of telemetry data to be collected are fundamental aspects of an IoT solution. This decision process should be driven by the business requirements. Before deciding what information to collect, the business motivation and goals—such as transforming a business model toward a service provider, adding new services, improving customer engagement, or optimizing operations and maintenance—should be clarified. The requirements about what telemetry is needed should be derived from the business goals.

There is a key trade-off between the volume of data collected and its cost. Data that is not collected cannot be analyzed, but you pay for collected data in terms of performance and cost. Trying to collect as much data as possible doesn’t always guarantee that the right business questions can be answered when needed. Also, collecting too much or unnecessary data makes it more difficult to differentiate useful information from “noise,” and also impacts the operations and management cost. In many cases, understanding the value of the collected data might be an iterative process.

One possible strategy is to program the devices to emit different granularity of telemetry data and then control that level from the cloud as needed. A configuration change command can then be used to instruct the device to change the collection profile and to start transmitting different levels of telemetry data.

In addition, different categories of data can be treated differently. Devices might split the data for hot-path processing— being sent in real time to the cloud—and cold telemetry, which can be collected locally and transferred on a delayed basis. For example, a device using network-condition detection can send hot-path data across a mobile network and transfer cold telemetry data after a Wi-Fi or wired connection is established.

In the case of complex devices containing multiple subcomponents (such as industrial equipment devices), the device telemetry most likely will need to be processed for each subcomponent separately (and treated logically as a separate device by the solution). As described previously, those telemetry streams can be segregated by using a protocol header property (such as “stream\_id”) to allow for differentiation and appropriate processing on the backend.

Another aspect to consider is how data will be correlated between devices, device topologies, components, and systems. The telemetry flow should contain appropriate attributes to enable linking the information on the backend for holistic insights across the entire system.

### Edge connectivity

Different topologies for direct or indirect device connectivity were discussed previously. When using Azure IoT Hub as the cloud gateway, the edge connectivity options are shown in Figure 1.

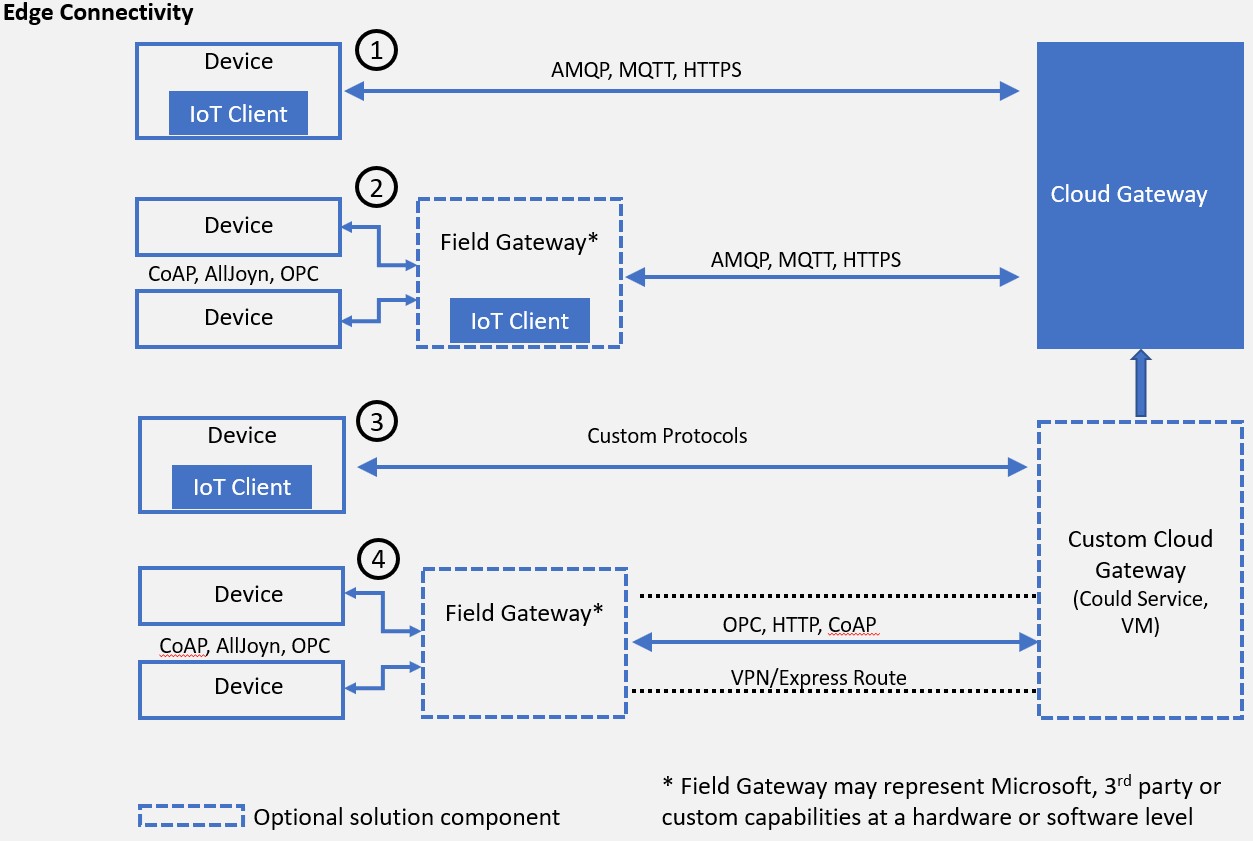


Figure 1 - Edge connectivity with IoT Hub

In addition to the topology, a number of other aspects need to be designed for the solution, as described in the following sections.

### Transport protocol

In the context of this architecture the focus of transport and messaging protocols is on IP-based communication between devices (including those acting as field gateways) and cloud gateways. Peer-to-peer and local network communication standards, link-layer protocols, and physical data transfer (wire, radio) are out of scope.

Note: There are additional efforts related to local network communications—Microsoft engagement with the AllSeen Alliance, which develops the AllJoyn standard (covering local network interaction of devices with an optional cloud gateway)—and for Industrial IoT scenarios— Microsoft engagement with the OPC Foundation around the OPC Unified Architecture (OPC UA). **Both of these standards** cover local network discovery and device interaction.

### Device interaction

The following principles define the Service Assisted Communication1 approach for establishing trustworthy bidirectional communication with devices that are potentially deployed in untrusted physical space:

* + - Devices do not accept unsolicited network connections. All connections and routes are established in an *outbound-only* fashion.
    - Devices generally *only connect to or establish routes to well-known service gateways* that they are peered with. In case they need to feed information to or receive commands from a multitude of services, devices are peered with a gateway that takes care of routing information downstream and ensures that commands are only accepted from authorized parties before routing them to the device.
    - The communication path between device and service or device and gateway is *secured at the transport and application protocol layers*, mutually authenticating the device to the service or gateway and vice versa. Device applications do not trust the link-layer network.
    - System-level authorization and authentication should be based on *per-device identities*, and access credentials and permissions should be near-instantly revocable in case of device abuse.
    - Bidirectional communication for devices that are connected sporadically due to power or connectivity concerns may be facilitated through holding commands and notifications to the devices until they connect to pick those up.
    - Application payload data may be separately secured for protected transit through gateways to a particular service.

1 <http://blogs.msdn.com/b/clemensv/archive/2014/02/10/service-assisted-communication-for-connected-devices.aspx>

**Physical and link layer considerations.** While guidance on physical transfer, link layer, and local network technology choices and usage are out of scope for this document, it is important to understand the potential impact of their usage on the communication at and above the transport layer. The underlying communication technology will impact not only the quality of service but will also dictate the frequency and communication patterns used by devices.

Networks built on radio and power-line technology are susceptible to interference and other signal quality issues, which can cause data frame and packet corruption and loss. Battery-operated devices will optimize the send/receive patterns for lower power consumption, and often will use radio on-demand, which means that devices will not be reachable at all times. Mobile operator networks (such as GSM, 3G, 4G) compensate for many of the effects of radio-based technologies, but higher packet latencies and packet loss are nevertheless common. With domestic roaming, whereby devices are allowed to roam across different operator networks, devices may quite frequently switch connections and IP addresses. A moving vehicle may switch between base stations every couple of minutes, depending on the reach of the particular frequency band.

These examples provide context for the common need to design the device communication patterns based on the underlying communication technology. A device can detect which type of connectivity is used (network condition detection) and switch the communication mode. For example, large binary transfers can be performed over Wi-Fi or a wired connection, while on a cellular or radio network the device will implement a reduced communication profile.

Another common pattern for implementing the service-assisted communication principles described in the Device interaction section above for power-constrained or battery-operated devices that are not always reachable, is to use an out-of- band communication channel (such as mobile carrier SMS) to “wake up” a device and instruct it to establish an outbound network connection to its “home” gateway, when time-critical commands need to be transmitted.

Virtual private network (VPN) technology allows for integrating and isolating a network, creating a single address space functionally equivalent to a local network, while in reality spanning multiple underlying networks. It provides mechanisms to securely join and participate in an isolated network but does not secure the traffic inside the network. Without further components like per-endpoint firewalls, it intentionally does not limit how the participants of the virtual network can communicate with each other. In scenarios where devices participating in a VPN are in physical control of users or potentially unknown intruders, the virtual network environment must be considered as hostile as the Internet environment.

VPNs can provide stable addressing for devices; but even with fixed assignment of addresses to each device inside a VPN context, the addresses are only useful when the device is actively connected. More commonly, devices will be assigned a dynamic address in a VPN, via DHCP, and will then be registered in DNS for discovery. This model is common for information devices, but typically causes significant management burden for individual IoT devices, especially when devices are mobile and frequently drop connections. VPN has significant network transfer volume and computation overhead for establishing and reestablishing connections, as well as for all communications. Hence, it’s more appropriate for use with (field) gateways and powerful devices.

While a VPN is a recommended technology choice for integrating existing datacenter assets into Azure IoT solutions, it is not recommended for integrating mobile or wirelessly connected devices, or an exceptionally large numbers of devices, into the Azure cloud. In industrial automation and other environments with stable and reliable connectivity, where relatively few devices (dozens, not hundreds) or environments need to be attached into the cloud, using network integration into Azure should be considered: Azure VPN,2 ExpressRoute3. For higher bandwidth, reliability, and lower latency needs ExpressRoute should be considered.

Field gateways (edge devices) at the edge of a production network should have separate access paths for the two environments. The edge device should broker information exchange between both environments at the application level, through a messaging application protocol. The gateway can be joined into an Azure point-to-site or site-to-site VPN,4 and will then also be addressable and accessible from within the cloud solution. On the cloud side, the VPN and on-site endpoints must be integrated through a custom cloud gateway solution hosted in Cloud Services or an Azure VM.

**Transport protocol options.** This document discusses the two most widely used transport level protocols: TCP and UDP. Other protocols at that level, like SCTP (IETF RFC4960) or Multipath TCP (IETF experimental RFC6824), or high-bandwidth applications of UDP (such as UDT) may play a role in select custom cloud gateway applications for special scenarios or existing protocol support but are out of scope for this document.

TCP (IETF RFC7936) provides stream integrity, stream order, and flow control between two network endpoints and is the default transport option for all scenarios except those called out in the UDP section.

UDP (IETF RFC7687) is a simple datagram (frame of bytes) transport model as a thin layer over IP and has minimal overhead. Therefore, it is a popular candidate for constrained device applications. UDP does not deal with packet order or packet loss and does not have a feedback-based flow control scheme. These are desirable properties for scenarios where a signal needs to be transmitted with very minimal end-to-end latency, where loss is acceptable, and where order of the signal components can be reconstituted at the receiver side when necessary.

Audio and video signals are often organized in stream container formats (such as MPEG transport stream) that can be transferred via UDP or any other unidirectional transport method with potential data loss.

UDP routes should be secured in accordance with the overlaid application protocol’s rules, most commonly using DTLS (IETF RFC63478). For applications where it is not acceptable to incur loss of data for sustained periods of time, and where latency is not the highest priority, TCP-based communication should generally be preferred outside local network applications. Inside local networks, UDP can be a helpful option to limit the compute and memory footprint for extremely constrained devices and is a viable choice in combination with the CoAP protocol discussed in the next section.

2 <http://azure.microsoft.com/services/virtual-network>

3 <http://azure.microsoft.com/services/expressroute>

4 <https://docs.microsoft.com/azure/vpn-gateway/vpn-gateway-howto-site-to-site-resource-manager-portal>

5 <http://en.wikipedia.org/wiki/IPv6_transition_mechanisms>

6 <http://tools.ietf.org/html/rfc793>

7 <http://tools.ietf.org/html/rfc768>

8 <http://tools.ietf.org/html/rfc6347>

Devices and services that actively listen for UDP packets are prone to flooding attacks, which includes the DTLS handshake. In those scenarios additional protection, such as isolated network tunnels in a trusted network relationship, should be applied.

### Messaging protocol

**Hypertext Transfer Protocol (HTTPS).** HTTP (IETF RFC7230, RFC7231, RFC7232, RFC7233, RFC7234, RFC7235) is the core protocol of the web, optimized for request/response interactions. HTTP is secured using TLS with the binding defined in IETF RFC2818 (HTTPS). The HTTP 1.1 protocol is purely text-based and simple. The designated HTTP/2 successor protocol is more concise, supports all HTTP 1.1 capabilities, and provides quite sophisticated framing and connection management solutions allowing for multiplexing and for modeling bidirectional data flow. HTTP/2 implementations must support TLS 1.2 protection. HTTP in the context of this document generally refers to HTTPS, that is, HTTP 1.1 + TLS 1.2 (RFC7230ff + RFC2818). HTTP/2 has very useful features for IoT scenarios and its use and adoption in the IoT space should be monitored. HTTP/2 is not the focus of this conversation.

The HTTP connection management model is optimized around relatively short client and server interactions. The interaction pattern supported by HTTP is a request/response model with responses correlated to requests by stream order. There are a number of techniques for modelling additional interaction patterns, such as notifications, or asynchronous message delivery over HTTP, such as “long polling.”

HTTPS can be considered a good option for scenarios where devices send data to the cloud gateway occasionally and as single messages or multi-record “uploads,” and where low-latency, bidirectional communication is not required. “Occasionally” means that the device sends data infrequently enough that maintaining an ongoing connection between the device and cloud gateway is not economical or technically feasible.

Secure, high-throughput event flow into an Azure-based solution using HTTPS is natively supported on Azure IoT Hub and Event Hubs. A device can receive commands or other information using periodical HTTPS lookups on a defined IoT Hub endpoint. If the device needs to receive remote commands with minimal latency instantly, a persistent bidirectional connection with a readily available network route to the device is required.

**Advanced Message Queueing Protocol (AMQP).** AMQP 1.0 (ISO/IEC 19464:2014, OASIS9) is a robust, connection- oriented, bidirectional, multiplexing message transfer protocol with inherent, compact data encoding. It provides optimizations for continuously connected devices, high-throughput communication, and has integrated flow control to protect sender and receiver from “overloading” each other. Libraries for AMQP 1.0 are available for a number of languages and runtimes, across numerous operating systems.

AMQP 1.0 is a good choice for scenarios where devices keep a long-lived connection, communicate with the cloud gateway on an ongoing basis, and potentially transfer large amounts of data.

**WebSocket protocol.** The WebSocket protocol (IETF RFC645510) is a bidirectional layer over TCP with negotiation over HTTP/HTTPS. It allows for sharing (multiplexing) the HTTP/HTTPS infrastructure and ports with other protocols that run over TCP, even though those protocols and their implementations require explicit support for WebSockets.

9 <http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-overview-v1.0-os.html>

10 <http://tools.ietf.org/html/rfc6455>

The most common use case scenarios for WebSocket are enabling bidirectional communication in HTTP/HTML web contexts and tunneling other application protocols such as AMQP 1.0 through HTTP/HTTPS infrastructure and ports. The AMQP 1.0 protocol has explicit binding for the WebSocket protocol for purposes of firewall traversal through HTTP/HTTPS infrastructure. All direct applications of the WebSocket protocol, like flowing frames of one of the aforementioned data encodings directly over WebSocket frames, are considered custom protocols because they do not have a standardized way for addressing or metadata framing that AMQP or other messaging protocols provide.

**MQ Telemetry Transport (MQTT).** MQTT 3.1.1 (ISO/IEC 20922, OASIS MQTT 3.1.111) is a lightweight client-server transfer protocol for messages. MQTT is attractive for constrained devices, because it is extremely dense with a very small footprint on the device, and for message frames (and respectively network bandwidth).

One design trade-off to be noted is that MQTT uses a very compact header format, but has no support for message metadata, such as a custom content-type header, requiring out-of-band agreements between the sender and receiver.

There are a few MQTT features that represent a challenge when used in large-scale distributed, high-availability IoT infrastructures. In a multi-node messaging system, the QoS2 “exactly once” delivery assurance would require a fully consistent system (across multiple nodes) at all times. While this is technically possible, such an implementation would be highly complex and will impact the latency and availability of the entire system (for more details refer to the CAP theorem12). Hence, the usage of QoS2 is not recommended for large IoT deployments. Common alternatives to exactly once delivery are deduplication at the receiver, or the use of idempotent operations. For example, for systems that are modeled to exchange state changes, “at least once” semantics is sufficient, because receiving the same state more than once will lead to the same result (assuming message delivery order is preserved, which is commonly the case, including when using MQTT).

Another challenge represents the usage of “retain” messages. This imposes unbounded state management requirements on the server (for duration and number of messages), which conflicts with resource governance requirements in high- scale systems and provides potential denial-of-service attack vectors through forced resource exhaustion. Appropriate authorization models could be considered and applied to mitigate those risks.

If MQTT is a candidate in a particular scenario for its footprint advantages, the recommendation is to constrain the usage to QoS 0 “at most once delivery” or QoS 1 “at least once delivery,” and avoid the usage of the “retain” feature.

**Constrained Application Protocol (CoAP).** The Constrained Application Protocol (CoAP, IETF RFC725213) is a datagram- based protocol that can be implemented over UDP or any other datagram transport, including GSM short-message service (SMS). CoAP is a very compact reformulation of the principles and methods of HTTP over a datagram transport. The Open Mobile Alliance’s (OMA) Lightweight M2M protocol (LWM2M) layers on top of CoAP (see the Management protocol section below for more details on OMA LWM2M).

11 <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html>

12 <http://www.julianbrowne.com/article/viewer/brewers-cap-theorem>

13 <http://tools.ietf.org/html/rfc7252>

The advantage of CoAP is its compactness compared to HTTP and other protocols. Because it is datagram based, there is also no immediate need to set up or maintain a connection or any state that spans multiple datagrams—until security is added in the form of DTLS, which introduces node affinity for the security context and which has properties of a connection. Supporting UDP-based protocols and DTLS is not trivial in conjunction with cloud gateways, because all communicating parties are easily susceptible to flooding attacks when traffic is admitted from across the open network (and large solutions can easily become a target for such attacks). Furthermore, packet loss on congested routes can be significant. TCP and TLS should be considered more robust for long-haul transfers.

For supporting LWM2M/CoAP on mobile carrier networks or to connect to field devices, the CoAP traffic can be isolated and reliably virtualized over VPN or ExpressRoute, either to a VPN gateway at the site where the devices reside or to the mobile carrier’s “private APNs.” (*Note:* CoAP is shown in Figure 1 as being implemented over a VPN tunnel between a field gateway and a custom cloud gateway.)

**OPC Unified Architecture (OPC UA).** The Open Platform Communications (OPC) Foundation’s Unified Architecture (OPC UA) includes a data model, security and a set of transport protocol mappings. An OPC UA server is an addressable entity that needs to be connected to in order to obtain data. An OPC UA field gateway can read data from an OPC server over a local network and forward it to the cloud gateway through a TLS-protected path using OPC UA PubSub, a JSON payload over MQTT or AMQP, like our open-source OPC Publisher. For individual machines, it’s also possible to use our OPC Proxy, to securely “bridge” the communication between an OPC UA server on the factory network and a, OPC UA client embedded in a web app in the Azure cloud, without opening the factory firewall.

### Security

**Trustworthy and secure communication**. All information received from and sent to a device must be trustworthy if anything depends on that information. *Trustworthy communication* means that information is of verifiable origin, correct, unaltered, timely, and cannot be abused by unauthorized parties in any fashion.

Even telemetry from a simple sensor that reports a room’s temperature every five minutes should not be left unsecured. If any control system reacts to this input, or draws any other conclusions from it, the device and the communication paths from and to it must be trustworthy.

Many IoT devices, such as inexpensive sensors or common consumer or industry goods enriched with digital service capabilities, will be optimized for cost, which commonly results in trading compute power and memory for cost savings. However, this also means trading away cryptographic capability and, more generally, resilience against potential attacks.

Unless a device can support the following key cryptographic capabilities, its use should be constrained to local networks and all internetwork communication should be facilitated through a field gateway:

* + Data encryption with a provably secure, publicly analyzed, and broadly implemented symmetric-key encryption algorithm, such as AES with at least 128-bit key length.
  + Digital signature with a provably secure, publicly analyzed, and broadly implemented symmetric-key signature algorithm, such as SHA-2 with at least 128-bit key length.
  + Support for either TLS 1.2 (IETF RFC524614) for TCP or other stream-based communication paths or DTLS 1.2 (IETF RFC6347) for datagram-based communication paths. TLS-typical support of X.509 certificate handling is optional and can be replaced by the more compute-efficient and wire-efficient pre-shared key mode for TLS (“TLS/PSK,” IETF RFC427915), which can be implemented with support for the aforementioned AES and SHA-2 algorithms.
  + Updateable key-store and per-device keys. Each device must have unique key material or tokens that identify it toward the system. The devices should be able to store the key securely on the device (for example, using a secure key-store). The device should be able to update the keys or tokens periodically, or reactively in emergency situations in case of system breach. Key update might occur over the air or through some other means, but updateability is required.
  + The firmware and application software on the device must allow for updates to enable the repair of discovered security vulnerabilities.

As a foundational principle, *all cloud communication with devices or field gateways must occur through secure channels* when the devices talk directly to endpoints provided by Microsoft Azure platform services.

If (legacy) devices must use insecure or nonstandard and proprietary communication paths into the cloud system, they should be connected through a separately hosted custom protocol gateway or a local field gateway.

There are and will be many cases where access control for devices on local networks has been solely realized through network-level access control, and all admitted members of the network can communicate freely without any, or with naïve, authentication and authorization. Devices existing in such networks *must communicate via a field gateway* at the edge of the insecure network.

14 <http://tools.ietf.org/html/rfc5246>

15 <http://tools.ietf.org/html/rfc4279>

### Physical tamper proofing and safety

Sensors and devices can and must often be placed in public areas, where anyone may potentially have physical access to them. Also, tampering with the device is not just the act of manipulating the device hardware or software. A digitally trustworthy sensor may be tricked into reporting misleading data by dismounting and relocating it. Or an attacker could impact the environment around the device, creating misleading physical conditions in the immediate proximity of the device, pushing the overall system into an erroneous reaction. A lit lighter held near a smoke or temperature sensor might, for instance, trick a digital building control system into flooding a hotel hallway with the sprinkler system.

IoT introduces a new dimension of security because IoT devices are used in a broad range of personal, commercial, and industrial applications, and not only does the threat landscape differ between the respective scenario environments, it also differs depending on the condition of the device related to its environment. For example, a vehicle in motion has a different threat landscape than a vehicle idling in front of a traffic light, and yet another from a parked vehicle. Securing the digital components of the vehicle is therefore much more complex than securing a “classic” software application— and this applies to many IoT scenarios in a similar fashion.

As the IoT space blurs digital and physical concerns, it also blurs security with safety. Suddenly, security threats become safety threats. If something “goes wrong” with automated or remote controllable devices—from physical defects to control logic defects to willful unauthorized intrusion and manipulation—production lots may be destroyed, buildings may be looted or burned down, and people may be injured or die. That is a different class of damage than someone maxing out a stolen credit card limit. The security bar for commands that make things move, and also for sensor data that eventually results in commands that cause things to move, must be higher than in any e-commerce or banking scenario.

Even though clearly beyond the control of a cloud-based system, it is therefore *strongly recommended that the device design incorporates features which defend against physical manipulation attempts* to help ensure the security integrity and trustworthiness of the overall system.

Some exemplary measures that can be taken to improve the security of the physical device are:

* Choosing microcontrollers/microprocessors or auxiliary hardware that provide secure storage and use of cryptographic key material, such as trusted platform module (TPM)16 integration.
  + Secure boot loader and secure software loading, anchored in the TPM.
  + Using sensors to detect intrusion attempts and attempts to manipulate the device environment with alerting and potentially “digital self-destruction” of the device.

16 <http://www.trustedcomputinggroup.org/developers/trusted_platform_module>

### Data encoding

There is a large and growing number of data encoding formats available. The optimal data encoding choice differs from use-case to use-case and is sometimes even constrained by factors like how much space is available for extra footprint on a device.

XML and JSON are ubiquitous on the server and on many client platforms. Both enjoy very broad library or platform- inherent support but have very significant wire footprint due to their text-based nature.

CSV is simple, interoperable, and compact (for text), but it’s structurally constrained to rows of simple value columns—which, however, is very often enough for time-series data.

BSON and MessagePack are efficient binary encodings that lean on the JSON model but have great encoding size advantages. Both require their own libraries and have some distinctive choices like lack of first-class array support in the case of BSON.

Google’s Protocol Buffers (“Protobuf”) and Apache Thrift yield very small encoding sizes but require distribution of an external schema (or even code) to all potential consumers, which represents challenges in systems of nontrivial composition complexity with multiple readers/consumers.

Apache Avro is generally as efficient—or more efficient—than the prior options and also natively supports layered-on compression. With Avro, the schema is embedded as a preamble for a set of records. This preamble requirement puts Avro at a disadvantage compared to MessagePack or BSON for small or highly structured payloads with minimal structural repetition.

### Data layout

Just as important as the encoding is the data layout, which can also have major impact on the encoded data size. A naïve JSON encoding approach where telemetry data is sent in the form of an array of objects, whereby each object carries explicit properties for all values, has enormously greater metadata overhead than a data layout mimicking CSV with a shared list of headers followed by an array carrying the row data.

The data layout convention defines how the structure of the data is constrained within the scope of the solution, so that data can be handled across the entire system, including devices, backend processing, analytics, and user interface. All those components will need to rely on a common model/schema.

An important principle in event-driven systems is that the data unit handled and processed in the context of a model is a record. A message, a storage block, or a document may contain one or multiple data records (or “events”). A sequence of records may span multiple messages or storage units.

The row/column structure of CSV provides a natural set of constraints for the layout and allows for a not-explicitly bounded list of rows (each equating to a record), with a not-explicitly bounded set of columns, whereby each column value is of primitive type.

For the map/array/value structural model supported by the JSON, Avro, AMQP, and MessagePack data encodings, there are the following common layout options: single record, record sequence, or record sequence with metadata preamble that, similar to CSV, contains a header describing columns, followed by the rows representing the record sequence.

The following matrix may help in choosing an appropriate encoding. In the layout column, “flat” refers to records that consist solely of primitive data types. “Complex” refers to data where records are structured beyond primitive types.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Layout** | **JSON** | **CSV** | **Avro** | **AMQP** | **MsgPack** |
| Single-Record, Flat Data | ++ | + | - | +++ | +++ |
| Single-Record, Complex Data | ++ | n/a | - | +++ | +++ |
| Record Sequence, Flat Data | + | ++ | +++ | + | + |
| Record Sequence, Complex Data | + | n/a | +++ | ++ | ++ |
| Record Sequence w/ Metadata Preamble | ++ | +++ | +++ | +++ | +++ |

Table 1 Comparison of data layout encodings

**Legend**: (-) poor; (+) good; (++) better; (+++) best

### Edge processing

In many application scenarios, especially those where devices communicate with their cloud backend systems via metered networks, it is not desirable to send raw sensor readings or status information across the communication link to the cloud because of the associated cost and load put on the cloud-system, when very many unprocessed data streams must be handled in parallel.

Often, IoT solutions specifically require evaluation of signal data streams, with video and audio covering particular signal shapes and spectrums, by application of digital signal processing algorithms or pattern matching or discovery, so it is required to treat these kinds of signals in a first-class fashion.

A temperature sensor provides periodic readings, maybe at 1 Hz, that manifest in a number per reading. A vibration sensor on a ventilation fan helping with determining equipment health in an industrial environment provides periodic readings, maybe at 500 Hz, that manifest in a number per reading. An audio sensor—a microphone—provides periodic readings that manifest in a number per reading, but at 44 kHz. A video sensor provides periodic readings that manifest in a very large matrix per reading and it does so at 60 Hz or 50 Hz.

All these signals benefit from preprocessing and compression before transfer and depending on the kind of signal there may already be broadly accepted and applied industry standards for how to encode, encapsulate, and carry the signals. If this is the case, like the MPEG standards for audio and video signal compression and encoding, they should be preferred, and passed through all parts of the system that are not equipped to interpret them unchanged.

The most trivial aggregation that can be applied to any time-series is to group several point-in-time records into a single record for either a particular period where the reading has remained stable, or for a fixed period by providing the average or median for the readings.

Furthermore, many devices are quite capable of pre-analyzing the raw data and using local compute capability, should it be generally preferred over sending large amounts of data across a metered network.

In many cases devices can use network condition detection and apply different pre-analysis, aggregation, and compression algorithms based on the type of connectivity used. For example, if the raw data is needed for later at-rest analysis, hybrid models can be used, where data that is required instantly and in near real time is transferred through the mobile network and the raw data is held locally and transferred at a later point in time over wired (or Wi-Fi) connection.

The edge processing is typically coupled with appropriate components on the backend that are responsible for interpreting the received data before downstream processing. For example, compressed data needs to be decompressed, and encodings need to be decoded appropriately. In many cases, this is done in the stream processor, reading data from the cloud gateway. If there are multiple consumers of the data, one stream processor can be dedicated to interpreting the incoming data (such as decompressing or deserializing) and to output the transformed data to an internal flow buffer (such as Azure Event Hub). This way, it will act as the primary source of incoming traffic for all other event stream processors.

In some situations, this type of processing can be done in a custom gateway, before reaching the ingestion point of the cloud gateway. The Azure IoT protocol gateway showcases how this type of custom processing can be implemented using a concept of a processing pipeline, where different modules can be plugged in to perform specialized processing before passing the data to the next one.

### Management protocol

There is an evolving set of device management protocols in the industry. The use of predefined device models enables efficient use of network, processing, and power resources on IoT devices. OMA LWM2M (or Lightweight M2M) is a standard defined by the Open Mobile Alliance that uses a compact resource model and interactions between device and server in order to support very constrained devices. OMA LWM2M provides a transport binding with CoAP for constrained devices.

Other device management protocols, including OMA DM, TR-069, and CoMI define device models and interactions with devices. OMA DM, used in mobile device management and some IoT implementations, uses XML (defined by SyncML) to enable device management and therefore is more verbose than OMA LWM2M. TR-069 is a technical specification published by the Broadband Forum that uses a bidirectional SOAP/HTTP-based protocol to manage devices. In addition to the high number of device management standards, a number of custom device management protocols exist where device vendors have needed to provide system capabilities between devices and servers/services. Different layers of those device management protocols would impact implementations of this reference architecture. For example, an implementation of OMA LWM2M in the context of this reference architecture will have implications on the device IoT client, provisioning API, gateway(s), event processor, and device registry components.

### Device management

The IoT device landscape is vastly heterogeneous considering the variety of hardware options, environments, operating systems and programing languages, and means of communications between device and services. The use of device management protocols provides an abstraction simplifying this complexity by defining a protocol, from the lower transport layer to the higher application layer, such that a service can provide the necessary information to a device to ensure the health of that device.

Service providers and enterprises need to enroll and discover, enable connectivity, remotely configure, and update software on devices in a way that is specified by defined policies and business processes. For example, depending on the industry, there will be vastly differing policies for the circumstances under which devices can be remotely configured and changed, with approval chains, regulatory auditing requirements, presence of physical safeguards, and more.

Every IoT system can provide a set of device management capabilities in order to ensure the health of devices and related business processes. The notion of a device registry is critical to enabling device management capabilities on remote devices and enabling a service-side interface for cloud applications to use the capabilities provided by remote devices. The following is a list of device management capabilities that may be enabled by the IoT system:

1. Device provisioning and discovery
2. Device access management
3. Remote control
4. Remote administration and monitoring
5. Remote configuration
6. Remote firmware and software update

### Device provisioning and discovery.

Most IoT device life cycles show that devices are manufactured and deployed to a variety of locations worldwide. The deployment location may not be known at the time of manufacturing; therefore, it may be important to enable a multiphase bootstrap process where devices are manufactured with the knowledge of a bootstrap service, which later provides connectivity details. When the device is deployed, the organization deploying the device provides further information, including device location and any other required information to the bootstrap service. The bootstrap service is configured to respond with the cloud gateway that will be used for this device. The device may need to repeat the provisioning process, for example in the scenario where device ownership changes.

In order to enable cloud applications to perform device management activities, the device may describe itself to the cloud when it creates a session with the cloud gateway. There are three core concepts related to how a device is described to the system:

* **Self-defined device model**

A device engineer (or developer using a device simulator) uses the self-defined device model through the process of iterating on the capabilities of the device as they build the device. A device engineer could start by creating a device that has few properties and supported commands and later add more. Similarly, that device engineer may have many devices, each of which provides unique capabilities; using the self-defined model, the device engineer is not required to register the structure of the device model.

* **Predefined device model**A production IoT deployment that operates under network and power/processing constraints greatly benefits from a predefined device model where minimal use of the device’s processing and power consumption are used. Similarly, minimal network traffic enables devices to transmit through heterogeneous networks (Wi-Fi, 2G/3G/4G, BLE, Sat, etc.) especially when using limited and expensive infrastructure (such as a satellite). When implementing a predefined device model, a device engineer might send device information encoded in one ortwo bytes that serve as a key into the predefined device model. The brevity of this approach results in efficiencies of one to two orders of magnitude compared to the self-defined device model.
* **Predefined master model**The device model and metadata related to the device are stored and maintained on the cloud side, but the device will remain unaware of those. This pattern is especially useful in brownfield scenarios where the device firmware cannot be modified, or the device should not store metadata.

**Device access management.** Devices (potentially managed by multiple parties) may enforce control of their own properties and commands, including create, read, and write access rights for device properties and execute rights for device commands. Depending on the IoT application, multiple authority levels may need to exist in order to control access to device resources appropriately.

**Remote control.** In IT scenarios, remote control is often used to assist remote users or remotely configure remote servers. In IoT scenarios, most devices do not have engaged users, therefore remote control is a scenario that enables remote configuration and diagnostics. Remote control can be implemented using two different models:

* **Interactive connection**In order to enable remote control through a direct connection to a device (for example, SSH on Linux or Remote Desktop on Windows) you need to create a connection to the device. Given the security risk of exposing a device to the open Internet, it’s recommended to use a relay service (such as Azure Service Bus relay service) to enable the connection and traffic to/from the device. Because a relay connection is an outbound connection from the device, it helps limit the attack surface of open TCP ports on the device.
* **Device command**Remote control through device command uses the existing connection and communications channel established between the device and Azure IoT Hub. In order to enable device command-based remote control, the following requirements need to be implemented:
  + The IoT backend is aware of device commands available on the device. This is usually defined as part of the device model.
  + The software that runs on the device needs to implement the remote-control commands. These device commands should follow a request (from the IoT backend to the device) and a response (from the device to the IoT backend) pattern.

The IoT service backend can keep record of historical response messages from device commands for auditing purposes. Updates to device state are made through device commands. Changes to the device metadata and state need to be pushed to the device registry and state stores, respectively. Updating the device state can be forced by a request from the IoT backend to the device, or the device can automatically update the backend upon recognizing a change in state. Automatic updating of the backend from the device should be done sparingly because it may generate network traffic and increase usage of the device processor and available power.

**Remote administration and monitoring.** Because most IoT devices do not have a direct user after deployment in a solution, remote administration is the experience where administrators can monitor the state of their devices and remotely update the state or configuration of devices through the use of device commands.

The health of devices can be determined by monitoring the data they are sending to the backend. This may include both operational data and metadata.

**Remote configuration.** Remotely changing a device’s configuration is a requirement for several stages in a device’s life cycle: provisioning, diagnostics, or integration with business processes.

**Remote firmware and software update.** Software defects can be security vulnerabilities, which makes the update of firmware or software to fix defects or deliver new functionality a critical capability of every IoT system. Remotely updating firmware and software on a device is an example of a distributed, long-running process that usually involves business processes. For example, updating the firmware on a device that controls a high-powered fuel pump may require steps in adjacent systems for rerouting fuel while the update is performed and verified.

Devices that support firmware and software updates are defined through the device model (or through a device type that is associated with a device model). Device updates are initiated at the IoT backend and devices are informed at an appropriate time through a device command. When a device explicitly supports remote update of firmware or software, the IoT backend should deliver the update commands based on defined business processes and policies. Upon receiving the device command to update, the device needs to download the update package, deploy the update package, reboot to the newly deployed (in the case of firmware update) or start the new software package, and verify that the new firmware or software is running as expected. Throughout this multistep process, the device should inform the IoT backend of the updated state of the device as it progresses through the multiple steps.

Delivering the update package can be done through a storage service like Azure Storage or through a CDN. Verifying the integrity of the downloaded package is important to ensure that the package originated from the expected source.

After completing a firmware update, the device must be able to verify and identify a good state. If the device does not successfully enter that good state, the software on the device should initiate a rollback to a known good state. The known good state could be the last known good state or a device firmware image known as a “golden state” stored in a storage partition.

**High Availability and Disaster Recovery (HA/DR) Deployment topologies**

IoT assets and devices commonly form distributed environments. They can be stationary or moving, dispersed or collocated and sometimes associated with local sites. Based on solution requirements, the devices might connect to a single centralized or distributed backend deployment.

The are several cloud backend deployment topologies and options for distribution of work across the different sites:

* **Single-site.** This is the simplest model and, in this case, the cloud gateway(s) and all device-related stores are collocated in a single datacenter region, while leaning on the high availability of the services used and on the platform-inherent support for disaster recovery. Because of its simplicity, this topology is often the starting point for most solutions.
* **Regional failover.** In a regional failover model, the solution backend will be running primarily in one datacenter location as in the single-site model, but the solution’s cloud gateway and backend will be deployed in an additional datacenter region for failover purposes, in case the cloud gateway in the primary datacenter suffers an outage or the network connectivity from the device to the primary datacenter is somehow interrupted. The devices will need to use a secondary service endpoint whenever the primary gateway cannot be reached. With a cross-region failover capability, the solution availability can be improved beyond the high availability of a single region. Disaster recovery and geo-failover concepts will be covered more deeply later in this section.
* **Multisite.** In the multisite topology, the solution runs concurrently and largely independently in multiple sites, but it is conceptually a single solution. Multiple sites can be collocated in the same datacenter regions to form “scale units” for which the entire data processing pillar can be stress-tested to maximum capacity, and then more capacity can be added safely by adding further scale units. Sites of the system may also be located across different datacenter regions for a variety of reasons, including proximity for reduced latency to the devices or policy concerns around data location. Each of these sites may also have a regional failover site. In the multisite model, devices are registered and thus “homed” in one of the sites.
* **Multisite with roaming.** In this variant of the multisite model, devices are homed in one of the sites (scale units) but may connect to the closest datacenter location based on some form of proximity estimation. The collected information is routed to the “home” site of the device.
* **Multisite, multihome.** In this variant, the device may roam across sites and captured data is stored across the various sites the device connects to and can be collected and consolidated as required.

This list of topologies is not exhaustive but helps to illustrate key patterns and trade-offs when planning an IoT deployment. Sometimes a certain topology can be applied to a subset of services and components, while other parts of the solution might use different deployment topology based on the specific solution requirements.

From a device perspective there are three possibilities of how a device or field gateway communicates with the service backend: to a single “home” endpoint, to a primary or secondary endpoint (for geo-failover), or to a set of endpoints in the multisite, multi-home scenario. The configuration of those endpoints can be static (for example, set on the device during provisioning) or managed as dynamic device configuration using commands from the solution backend.

There is also an additional option for devices using a token service. If a device cannot reach the destination endpoint, it can contact the token service to acquire a new endpoint and the appropriate token for it. This mechanism provides for dynamic reactive redirection of devices if needed (in contrast to proactive changes of a predefined configuration held on the device). It can be applied in addition to managing the endpoint configuration on devices. The token service can intelligently manage the map of sites, but can also be reconfigured for, as an example, maintenance purposes.

Apart from managing the endpoints that devices use to connect, Domain Naming System (DNS) entries and related services such as Azure Traffic Manager can be used for redirection of traffic to the desired backend endpoints. It is important to note that this technique depends on the ability of devices to use DNS and that its accuracy is driven by the Time-to-Live (TTL) value of the DNS host entries kept in local DNS caches.

**Cross-region availability**

Applications running in Azure benefit from the high availability (HA) of the underlying services provided by Azure. For many Azure services and solutions, high availability is provided by using redundancies at the Azure region level. In addition, Azure offers a number of features that help to build solutions with disaster recovery (DR) capabilities or cross- region availability if required. Solutions need to be designed and prepared to take advantage of those features in order to provide global, cross-region high availability to devices or users. The article “Azure Business Continuity Technical Guidance”17 describes built-in Azure features for business continuity and DR. The paper “Disaster Recovery and High Availability for Azure Applications”18 provides architecture guidance on strategies for Azure applications to achieve HA/DR.

Because cloud solutions are composed of multiple services, it is important to consider what is necessary to achieve HA/DR for the individual services or components of the solution, instead of thinking about one approach for the entire solution. Before deciding on techniques to be applied, it is important to define the requirements and expected availability for the subservices/components of the solution. Typically, subcomponents will have different requirements for availability, scalability, performance, and consistency. For example, device telemetry, commands to devices, backend analytics, LOB system transactions, and end-user UI will all have different availability, latency, and consistency targets.

Even different telemetry streams or command types will have different requirements (for example, a telemetry stream for the infotainment system of a vehicle has different processing requirements than telemetry coming from the engine). In case of a disaster, some components can be operated in degraded mode, or some of them might not even be required for a certain period of time. The disaster recovery techniques need to be designed for each category/type of service or function of the system individually, based on the specific requirements. There is always a trade-off between availability, other system requirements (including per-CAP theorem), and the cost of implementation and operations.

17 <https://msdn.microsoft.com/library/azure/hh873027.aspx>

18 <https://msdn.microsoft.com/library/azure/dn251004.aspx>

A major factor for geo-distributed topologies to consider is where state is stored and if services perform stateless or stateful processing. Stateless processing can be redirected (or failed over) to another scale unit, site, or region by ensuring the appropriate services (such as compute nodes, websites) are provisioned there. These can be actively running all the time, be available but not active (that is, in standby mode), or can be provisioned on demand as part of a disaster recovery procedure. Stateful services, however, represent a bigger challenge because, in addition to the service runtime, state and data need to be replicated and synchronized. Dependent on the consistency level required, state and data can be replicated synchronously or asynchronously where eventual consistency is sufficient. In some cases, data might not need to be replicated to each site if it’s sufficient to collect and consolidate the data to a centralized location at a later point. This dependents on the amount of data and specific solution needs.

With the proposed IoT reference architecture, the relevant state is kept in the following components and an appropriate technique for state replication should be defined for each category:

* **Device identity store.** The device identities and the associated security material need to be known at each site to which a device is expected to establish a connection. This includes secondary sites for failover or any other site a device can connect to in multisite scenarios. Typically, identities are slowly changing data managed through the provisioning workflow. The provisioning API provides a good abstraction layer that encapsulates the provisioning operations and is a natural place to extend and manage cross-site identities as needed. For example, when a new device is created, the identity record can be immediately written to a secondary site. For standby or on-demand deployed DR sites, it might be also sufficient to perform a regular export/import of the primary into the secondary store. The time interval between exports will define the recovery point objective.  
    
  In many cases devices will be provisioned long before they effectively try to connect to an endpoint. In those cases, eventual consistency between the identity stores is acceptable. Batch provisioning operations might be even performed in parallel against multiple locations. Using techniques such as check pointing, and appropriate error handling should ensure that a consistent state is established across the sites.
* **Topology store.** The topology store serves as an index for device discoverability, and for the majority of scenarios, implementations can be assumed to be eventually consistent, with a well-known time-to-live for replicated records. This means that an update of metadata may take up to this time-to-live limit to replicate. The DNS infrastructure uses a similar strategy. It will be beneficial if at least an initial record for each device is inserted into the device registry through the same mechanism used for the identity store (that is, the provisioning API). Attributes and metadata changes can be replicated asynchronously through the system.  
    
  In many cases the device registry contains only slowly changing data, and regular import/export might represent a sufficient alternative to a continuous replication of entries.
* **State store.** Device operational data is commonly characterized as high-volume and high-velocity data. As discussed previously, this data will be segregated in different stores based on needs and access patterns. The need for transferring or replicating each data category should be analyzed. Raw telemetry data most likely doesn’t need to be available on a secondary site. Aggregated data will represent a reduced data volume which might be easier to replicate if needed.  
    
  Often, the history or previous state might not be necessary for the application backend logic. In many scenarios, alerts, notifications, or even command and control events to devices can be applied just based on device metadata (such as type of device, group, category) or attributes (such as state received in a telemetry message).  
    
  Once it’s decided which types of operational data will need to be replicated, one of multiple options can be applied:
  1. Use the built-in capabilities of the underlying storage service (Cosmos DB, Azure Storage, and SQL Database, for example, already provide built-in geo-replication capabilities).
  2. Use a dedicated event processor that picks up the relevant information and transfers it to an Event Hub in the remote site (this represent a special data replication channel that will need to be processed by another event processor on the remote site and transformed into the desired storage format).
  3. Use some other mechanism built as part of the application layer (for example, write operations to a remote storage account or scheduled regular export and respective import on the remote site, which can be performed regularly or on demand).
* **Brokered messaging.** The cloud gateway and other backend internal queues, topics, or Event Hubs that are used to decouple components of the solution, durably store messages. Typically, after a message is accepted from the cloud gateway it will be processed by the solution backend and there is no need to replicate such a message to another site. In a case of a broker outage, the messages are still protected, but unavailable to be read. If those messages “in-flight” are considered absolutely critical for cross-region failovers, then they might need to be replicated to a secondary site. However, typically the messages are brokered for a very short period of time and then consumed and transferred into one of the described persistent data stores. Protecting data in the persistent stores is the typical strategy because the total latency of replicating messages in flight will be almost the same as the latency for protecting the persistent data stores. Thus, protecting messages in flight typically won’t significantly impact the RPO/RTO targets.
* **Hot-path analytics state.** Analytics and complex event processing engines keep in-memory state for aggregations or over certain time periods. There is no easy way to restore the in-memory state of those engines without a sophisticated event-replay mechanism. If critical business logic relies on this type of state, alternative calculations based on persisted historical data might be necessary.
* **Actor state.** Actor state is typically backed up by durable storage that should be replicated to a remote site if needed. In case of recovery, the actors can reload their state on the remote site.
* **System configuration.** Changes to the solution configuration (for example, changing threshold limits or business rules) will need to be propagated to secondary sites as needed.

Independent of the individual design choices, in distributed computing environments, it is always a good practice to use idempotent operations to minimize side effects not only from eventual consistent distribution of events, but also from duplicates or out-of-order delivery of events. In addition, the application logic should be designed to tolerate potential

inconsistencies or “slightly” out-of-date state, because of the additional time it takes for the system to “heal” or based on recovery point objectives (RPO). The following article provides more guidance on this topic: “Failsafe: Guidance for Resilient Cloud Architectures.”19

**Data protection and privacy**

As IoT scenarios receive growing attention from consumer protection groups and data protection regulators of various governments, it is expected data collection as well as remote control scenarios to be subject to increased regulation.

Solution builders must anticipate regionally differing regulation on what data collection is allowed by default, where owners or equipment operators have a right to opt-out, or where they must opt-in for data collection even to be permitted. They should also anticipate that any data collection and remote-control capability must allow temporal suspension, and that owners or equipment operators may want to erase collected data for past periods. Anonymized data collection may be an option in this context.

In spite of equipment manufacturers, insurers, leasing firms, and other corporations driving the data collection initiatives, it is not clear whether the data collected from a vehicle is legally owned by the collecting company. Solution builders must anticipate that regulation, varying by jurisdiction, will empower owners and equipment operators to have full control over the usage rights and retention duration of their data.

Assuming the example of a vehicle, there are many cases where the geolocation of the vehicle at any given time may be a very private matter that the current driver would not want to make known to anybody. That means that the geolocation cannot appear, associated with the vehicle, as long as there is a way to correlate the driver with the vehicle, and it also cannot appear to be associated with the driver in any permanent record.

However, the geolocation may be important and potentially acted upon should the driver get into an accident, because harm to him/herself, potential passengers, and other third parties will likely constitute an overriding priority.

Solution builders also need to anticipate opt-in and opt-out scenarios and (potentially regulation-mandated) opt-out actions to occur retroactively long after the data has been collected. Some data held in the system may also not be legally owned by the collecting party and may therefore not be present in the system after an opt-out.

It is therefore recommended for non-aggregated information existing anywhere in the system to retain attestation and link to its source and to any immediately related parties that may have entitlement to the data being erased. In order to enforce strong segregation of data and protection of sensitive information through bulk theft, it may further be required to encrypt information on a source-by-source basis.

It should also be anticipated that IoT systems and the collection of data might play a critical role in investigation scenarios as well as in the analysis of accidents or other mishaps and may become grounds for litigation. Therefore, strong attribution including proof of authenticity that disallows repudiation of the data origin will be of high value and likely required in regulated scenarios.

19 [https://docs.microsoft.com/aspnet/aspnet/overview/developing-apps-with-windows-azure/building-real-world-cloud-apps-with-windows-azure/more-patterns-](https://docs.microsoft.com/aspnet/aspnet/overview/developing-apps-with-windows-azure/building-real-world-cloud-apps-with-windows-azure/more-patterns-and-guidance) [and-guidance](https://docs.microsoft.com/aspnet/aspnet/overview/developing-apps-with-windows-azure/building-real-world-cloud-apps-with-windows-azure/more-patterns-and-guidance)

When building IoT solutions, it is important to consider compliance and certification requirements layer by layer. In order to achieve compliance for the overall IoT solution, each underlying layer will have to fulfil specific requirements. Typically, not all solution and platform components or services have the same requirements or fulfil the same accreditations. For example, not all Azure services have the same certifications (for example, ISO 27001, SSAE 16) and the solution builder should take into account which ones they need to use to allow them to achieve the intended solution accreditations.