Classified as Microsoft Confidential 1



**Microsoft Azure Cosmos DB IoT Solution Accelerator**

Version 1.0  
December, 2019

Would you like to provide feedback or do you have an idea or suggestion based on your experience with Cosmos DB? We would love to hear from you! Please email [AzureIoTRefArcVoice@microsoft.com](mailto:AzureIoTRefArcVoice@microsoft.com)

Contents

[About the solution accelerator 3](#_Toc27989513)

[High-level concepts 4](#_Toc27989514)

[Non-relational data and NoSQL 4](#_Toc27989515)

[Event sourcing pattern 6](#_Toc27989516)

[Serverless and no/low code processing 7](#_Toc27989517)

[IoT reference architecture 8](#_Toc27989518)

[High-level architecture for the solution accelerator 13](#_Toc27989519)

[Adapting the sample scenario to your own 13](#_Toc27989520)

[Architecture components 13](#_Toc27989521)

[Detailed walk-through of the solution accelerator components 15](#_Toc27989522)

[List of deployed Azure resources 15](#_Toc27989523)

[Azure Cosmos DB: Data is at the core 17](#_Toc27989524)

[Cloud gateway: Managing and devices and ingesting telemetry with IoT Hub 22](#_Toc27989525)

[Stream processing, event sourcing, and data management 30](#_Toc27989526)

[Logic Apps 40](#_Toc27989527)

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.

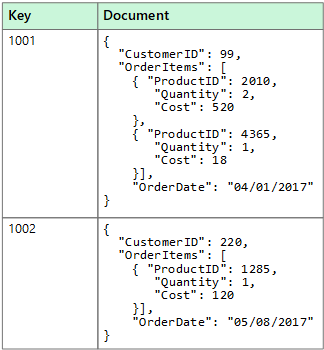
What all of these data stores have in common is that they don't use a relational model. Also, they tend to be more specific in the type of data they support and how data can be queried. For example, time-series data stores are optimized for queries over time-based sequences of data, while graph data 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.



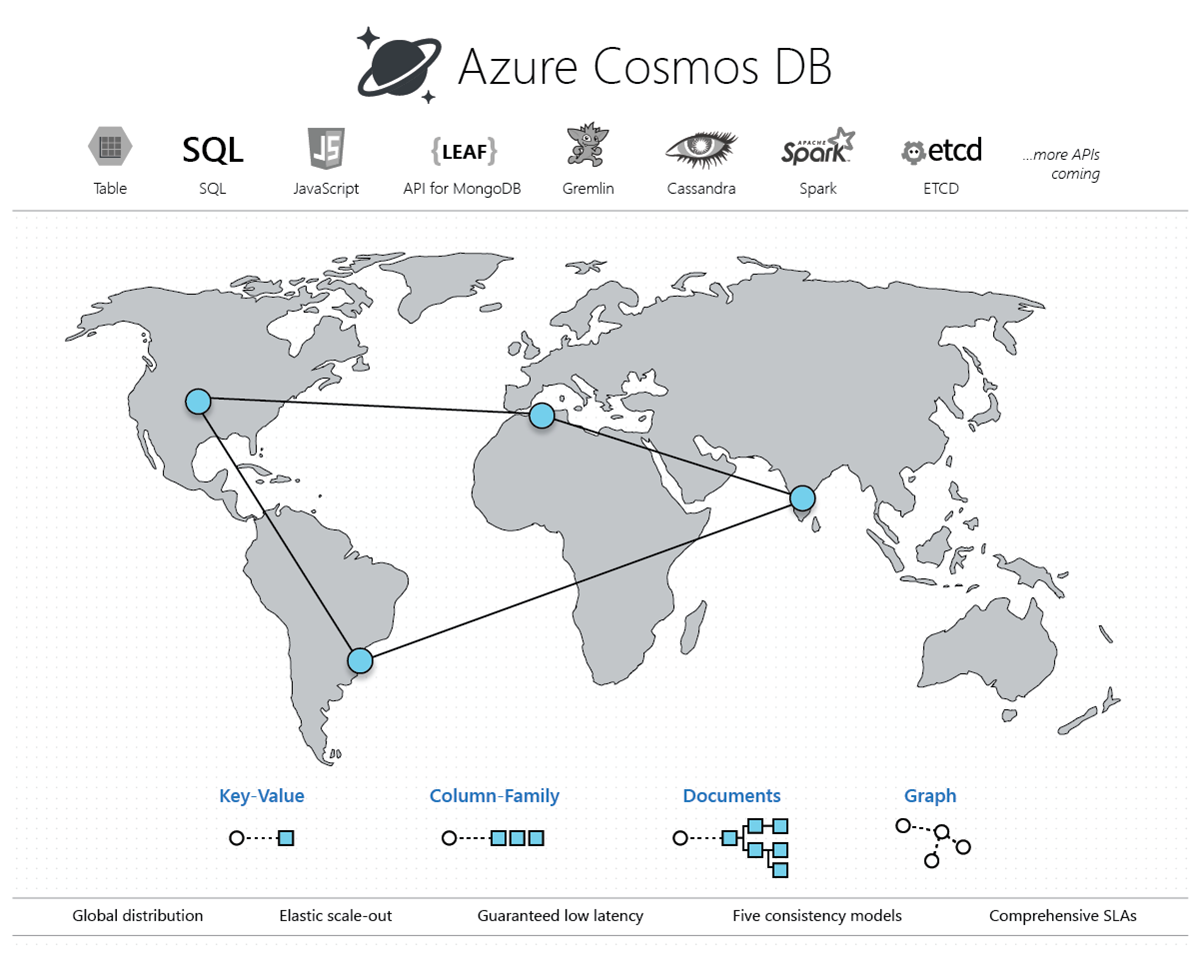
The application can retrieve documents by using the document key. This key is a unique identifier for the document, which is often hashed, to help distribute data evenly. Some document databases create the document key automatically. Others enable you to specify an attribute of the document to use as 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.

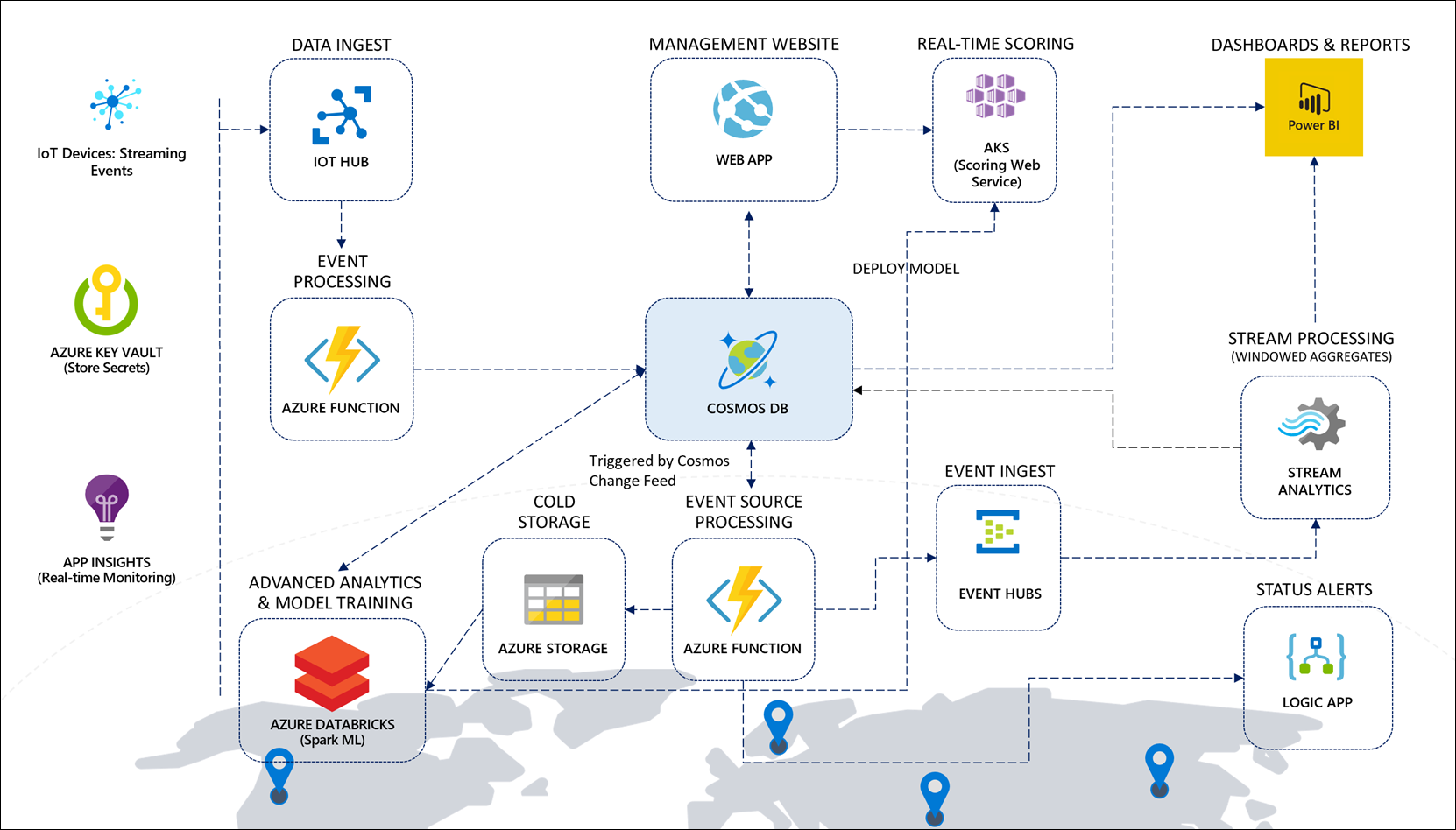
**Logging**

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:

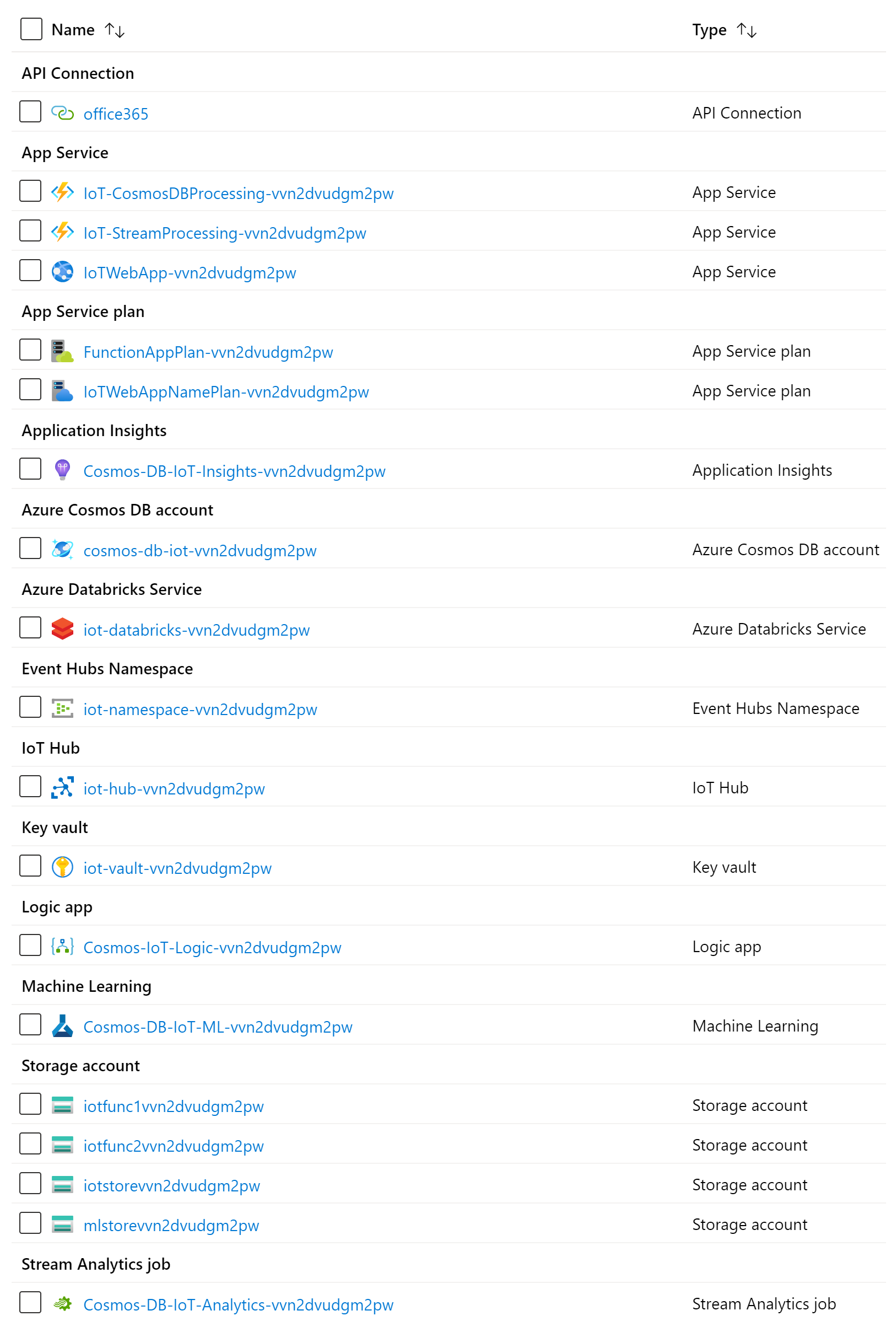
A **Web App** allows Contoso Auto to manage vehicles and view consignment, package, and trip information that is stored in Cosmos DB. The Web App is also used to make real-time battery failure predictions while viewing vehicle information. **Azure Key Vault** is used to securely store centralized application 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

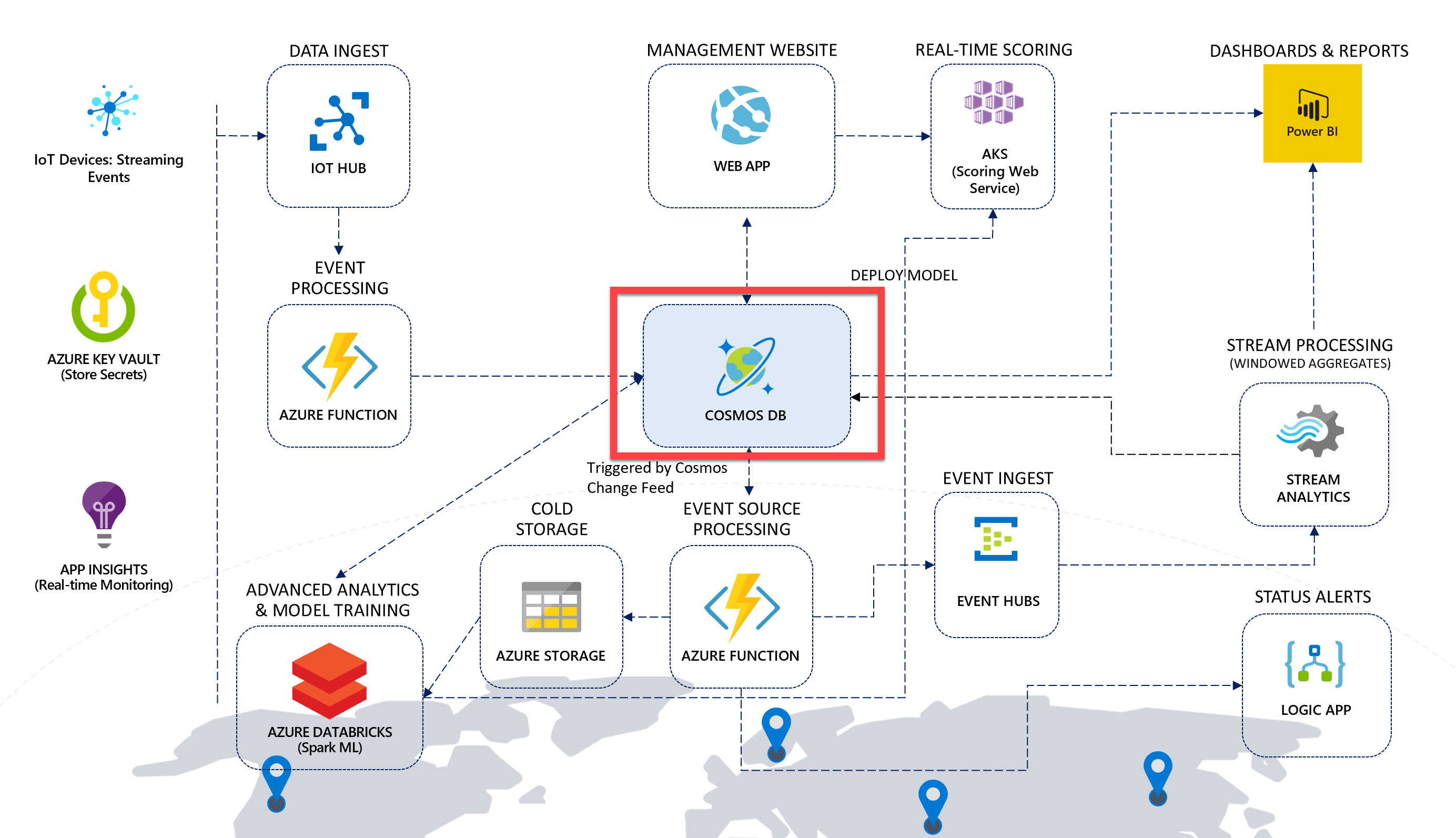
Now that we have established a baseline for high-level concepts and how they apply to the sample scenario, this section provides a detailed walk-through of each of the components and how information flows from one end of the architecture to the other.

## List of deployed Azure resources

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



## 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.

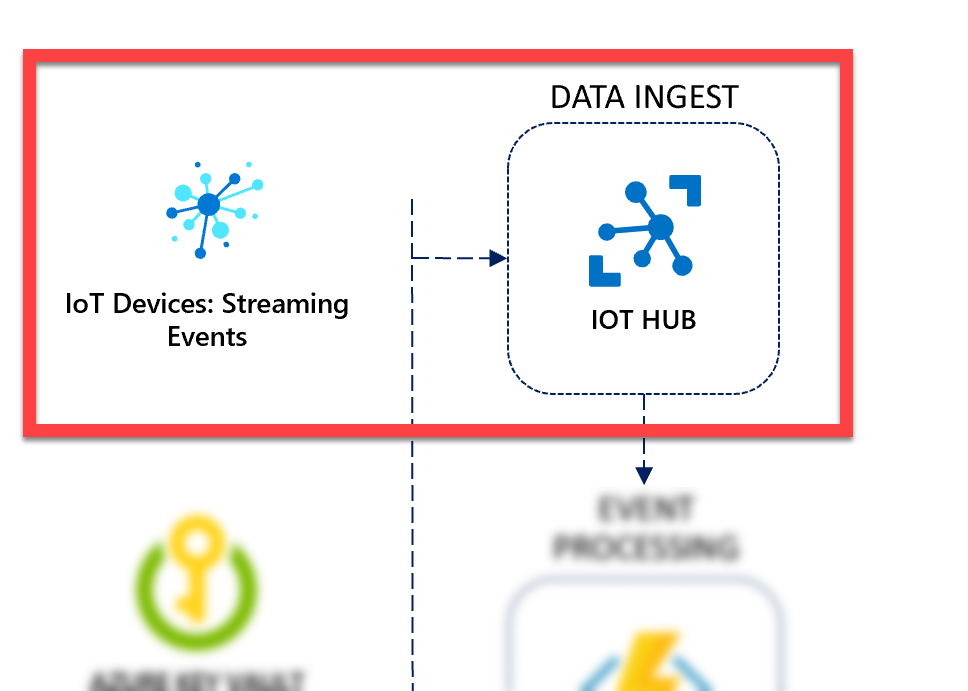
**About the Cosmos DB indexing policies**

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**

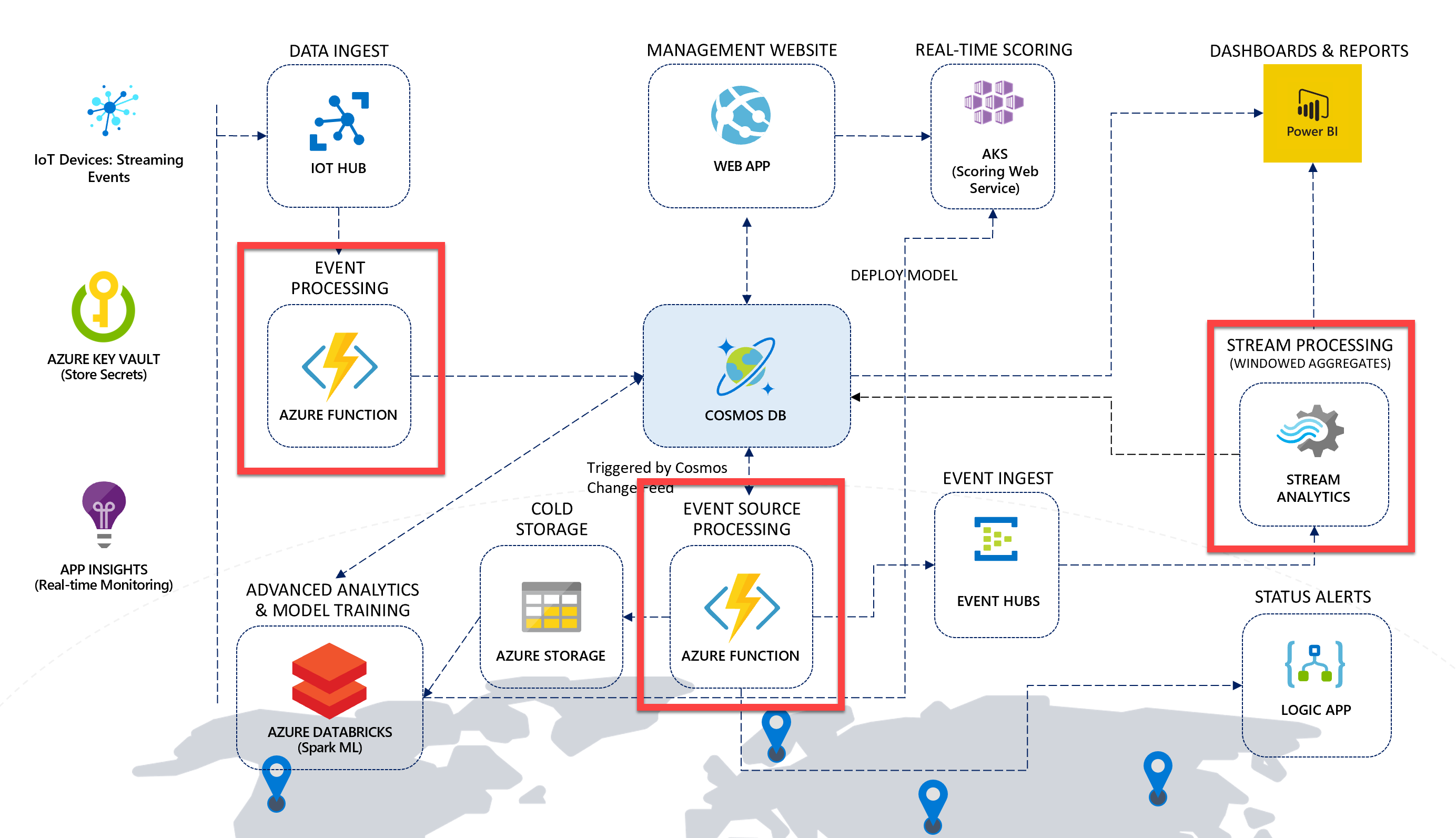
| **Capability** | **Azure Stream Analytics** | **Apache Spark in Azure Databricks** | **HDInsight with Storm** | **Azure Functions** | **Azure App Service WebJobs** |
| --- | --- | --- | --- | --- | --- |
| Inputs | Azure Event Hubs, Azure IoT Hub, Azure Blob storage | Event Hubs, IoT Hub, Kafka, HDFS, 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

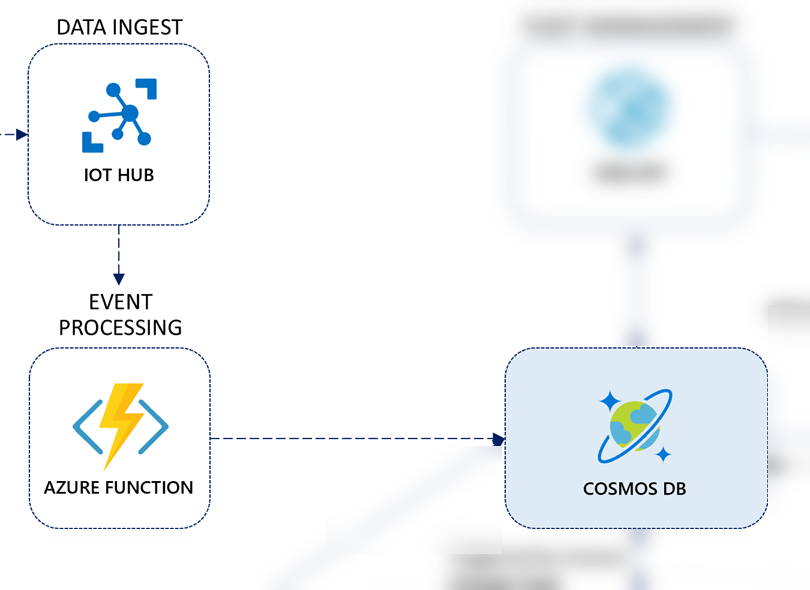


In the architecture for this scenario, Azure Functions play a major role in event processing. These functions execute within an Azure Function App, Microsoft's serverless solution for easily running small pieces of code, or "functions," in the cloud. You can write just the code you need for the problem 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.

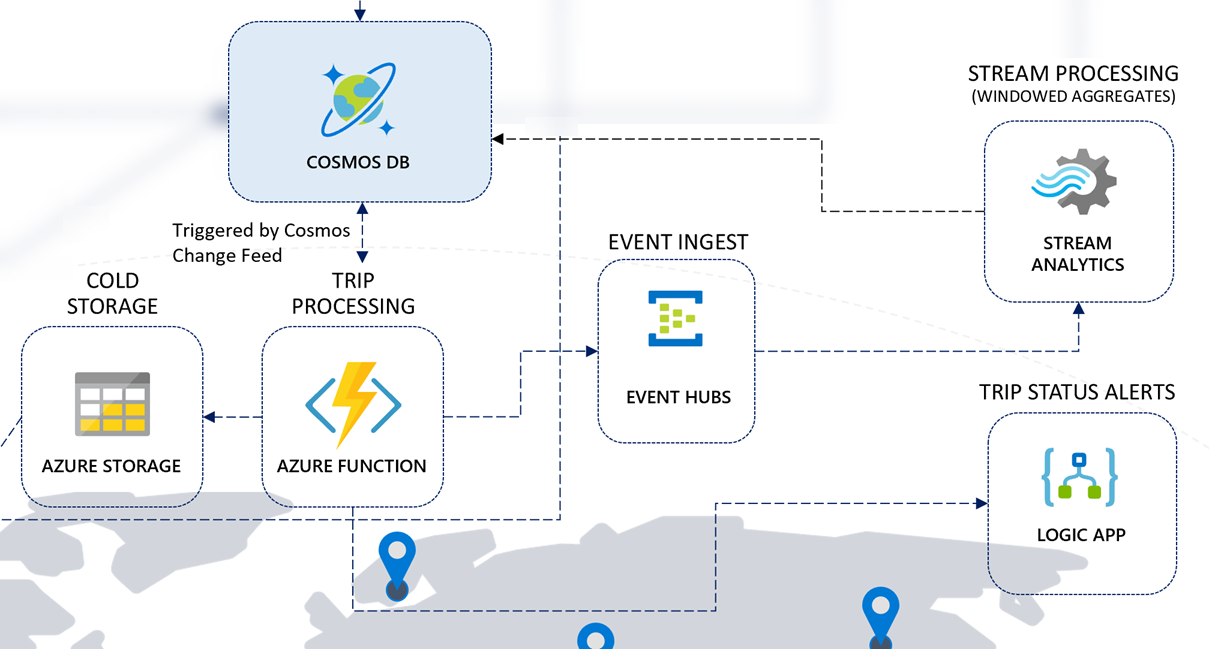
You may wonder, if a Function App contains several functions within, *why do we need two Function Apps instead of one*? The primary reason for using two Function Apps is due to how functions scale to meet demand. When you use the Azure Functions consumption plan, you only pay for the time your code runs. 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.

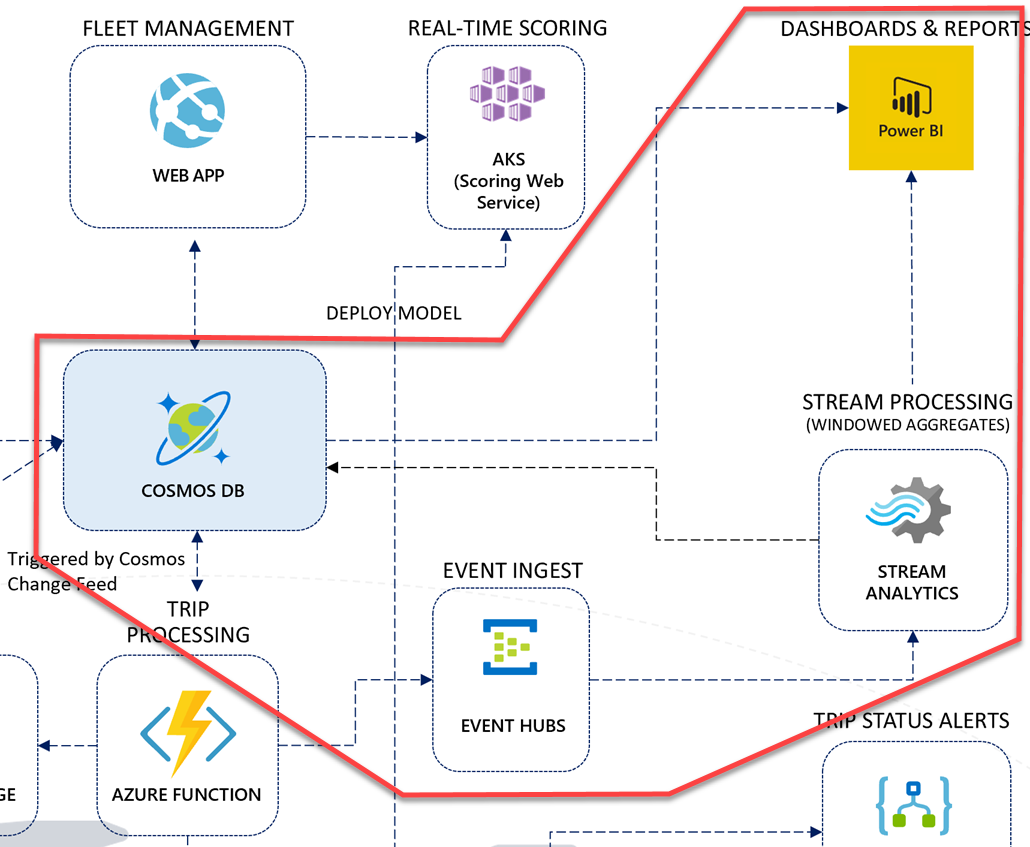
* **IoT-StreamProcessing Function App**: This is the Stream Processing Function App, and it contains two functions:
  + **IoTHubTrigger**: This function is automatically triggered by the IoT Hub's Event Hub endpoint as vehicle telemetry is sent by the data generator. The function performs some light 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

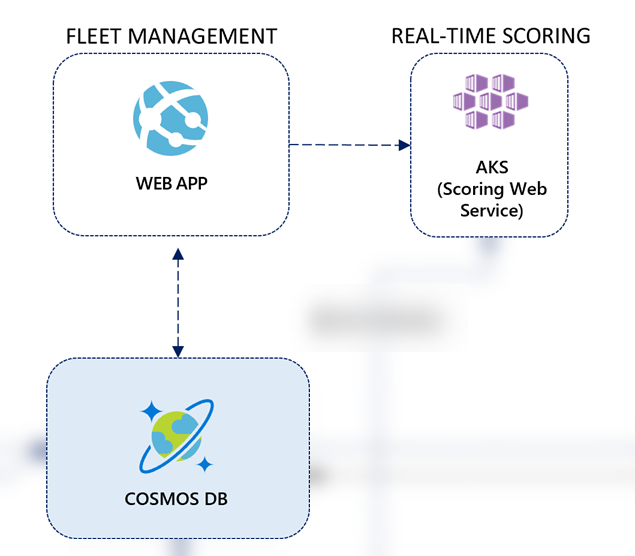
cosmosdb

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/).



**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 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) |