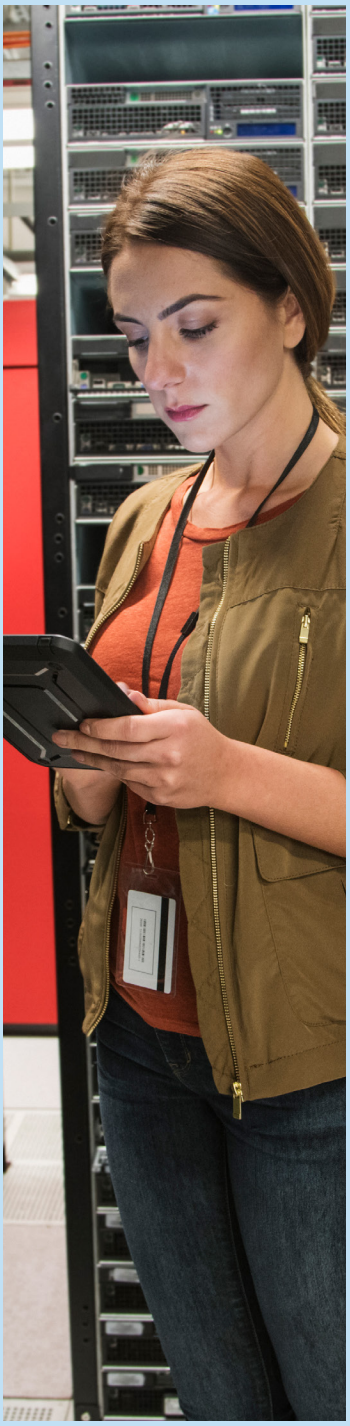
VIEW POINT



TIME SERIES STORAGE SOLUTION USING AZURE COSMOS DB CASSANDRA API

Abstract

This article provides an overview of the IoT time series data storage solution by Infosys for one of their customers, an Oil & Gas company. The solution leverages Azure Cosmos DB Cassandra API and other Azure services to monitor, manage, deploy IoT devices remotely.

After reading this article, you will learn:

- How Azure Cosmos DB is leveraged for storing time series data
- The design, challenges, and optimizations for a typical time series data solution
- Data processing, backup and restore processes
- Learnings and considerations while utilizing Azure Cosmos DB for time-series data

Infosys®
Navigate your next

Overview

In the IoT world, managing an IoT device means to provision, deploy, monitor, store & control the device and device data remotely. Provision & deploy refers to enabling the systems to track device, consume data and deploy additional modules & features to the device. The IoT devices emit time-series data which are then monitored & stored to provide actionable information to manage the device remotely. We can categorize the IoT solution into 3 key areas

1. Edge – physical IoT device (also referred to as IoT gateway) deployed in the field that deals with device registration, configuration, collating, converting the sensor analytical data into digital data and transmitting the data
2. Cloud – the digital side of device dealing with maintaining the device twins, configuration, interpreting and processing the transmitted time-series data
3. Application – the processed and stored data is consumed for analytics purposes and to derive actionable information

Here we will limit our discussion to the cloud part of the IoT solution. We will discuss how the data from IoT gateways is processed, the Azure services involved and their roles leading to finally storing the data in Azure Cosmos DB.

The IoT solution is intended to provide a managed service solution for other energy sector organizations. The product would be installed at the field locations for various energy customers, and this would be utilized to manage, monitor and control the field devices remotely.

The storage solution was implemented as a part of overall product redesign and refactoring from one of the major cloud service providers to Microsoft Azure. The availability of the required services across multiple regions specifically in countries with data residency requirements acted as a key driver and major differentiator in implementing this solution in Microsoft Azure. Along with this, the Azure Cosmos DB with Cassandra API enables the solution to be portable and supports multi-cloud solution.

Solution architecture

In IoT solutions, the common data processing pattern is to ingest the time-series data, handle anomalies and manage the devices remotely. The IoT solution ingests large amount of time-series data from devices and connected sensors hosted across geographies. The ingested data is interpreted and processed based on the relationship between IoT gateways, associated sensors, configurations and frequency. Finally, the time series data is stored with all available observations for further analytics and processing. The system also provides the capability to manage the devices by sending commands back to the devices. A user portal integrating these services is provisioned to manage these devices. These services enable the end-to-end solution that supports cloud-native, API-first and data-driven capabilities.

The product is scalable, cloud agnostic, distributed solution that can be implemented in both public cloud and on-premises.

The IoT solution is developed with Azure services including Azure IoT Hub, Azure Kubernetes Services, Azure Service Bus, Azure Cosmos DB, Azure Storage, Azure Gateway Ingress Controller and Azure API Management. This enables key features for our solution such as:

- Connect to IoT devices hosted in remote locations
- Custom microservices to manage identity, configuration, gateway and device mappings, logs, deployments etc.
- Processing and storing the time-series data
- Protect the services from external vulnerabilities
- Use the stored data for predictive maintenance & downtime identification
- End user portal with the ability to consume the services and manage the devices

Below is the overall product solution –

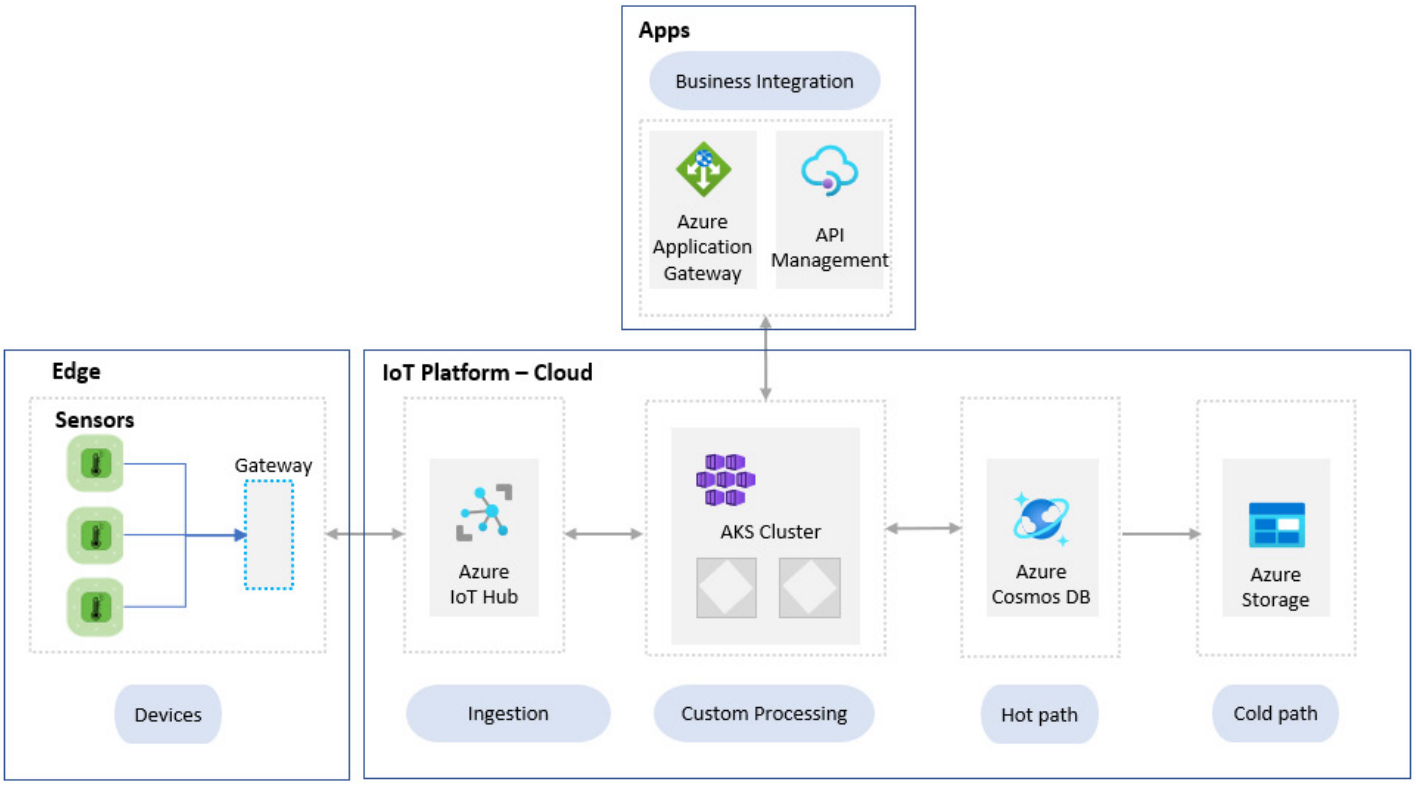


Figure 1. Product Solution

Reference Architecture

In the ingestion section, the IoT devices generated time-series data, sent as messages, is processed via the IoT Hub. Each IoT device generates messages with custom structure based on the sensors tagged to it. Each message is enriched with a custom device id enabling us to identify the exact device and the structure of the message for us to process.

In the custom processing section, the custom message was further processed to unmarshall the message based on the configurations stored. Multiple microservices were implemented on Azure Kubernetes Services to provide various features such as configuration, mapping, deployment, storage, notification, and version management for the product.

The processed message contents are then pushed further to Azure Cosmos DB for hot path storage and into the blob storage for cold path storage. The storage related microservice would handle both the read and write of the time series data to and from Azure Cosmos DB. The overall Open API specs is exposed via Swagger for external usage for the Apps.

For the storage solution, some of the some of the key requirements were–

1. Distributed time-series database that could be setup for both public and private clouds
2. Cloud agnostic solution enabling portability
3. Fully managed database service with minimal operations involvement
4. Automatic and Instant scalability along with data replication capabilities
5. Option to identify and scale for customer specific usage patterns
6. HA-DR information

Azure Cosmos DB's Cassandra API was ideal for these requirements as it offers most of the required features for the product. Cosmos DB is a managed service and is not available in on-premises or in countries with no Azure presence. However, since we are using Cassandra API, we could implement a Cassandra Managed Instance or a custom Cassandra cluster in the private cloud and continue to use the product without any issues

Database and Schema design

In this section, we will discuss the type of ingested data, ingestion pace, query patterns, schema design, throughput allocation and optimization for the application.

As mentioned earlier, the gateways were connected to various kinds of sensors, sensors that could capture images, pressure, temperature, vibration frequency, flow speed etc. The data and data type for each sensor could be different and the overall storage design was expected to handle this successfully.

The IoT gateways could be configured to transmit the data at varied frequency and varied structure. The frequency of telemetry was from every 1 second to over 5 minutes. The overall application was expected to be write heavy with 90% operations on the database to be write operations with over 1 million writes to the database every minute.

The design for data within Azure Cosmos DB was built based on the usage patterns that were expected for the solution. The application had usage patterns to write and read

1. Latest and history data generated by sensor
2. Batch of latest and history data for set of sensors or IoT gateways
3. Need to scale and manage individual customer data independently
4. The history data query would be mostly for the recent history

Based on the usage patterns and overall requirements, the design storing data, for each customer a table was created within a single key-space in Cosmos DB. The table structure of for storing data is depicted below.

CREATE TABLE <keyspace>.<CustomerId> (

deviceId	*text,*
tagName	*text,*
value	*double,*
timeStamp	*timeuuid,*
quality	*bool,*
primary key timeStamp)	*(deviceId, tagName),*

) WITH CLUSTERING ORDER BY (timeStamp DESC);

This design ensured –

- The ability to segregate individual customer data into separate tables and the ability to scale them independently was a significant benefit for choosing this solution
- The cluster key was ordered in descending order to ensure the latest data is still on the top
- History values of a particular tag can be queried for required duration
- Latest value of a particular tag can be queried based on top 1 entry for the primary key
- History/latest values for a batch of tags can be queried

Now, let's assume a specific use case of getting the values for a particular device without providing the tag. The above design doesn't support it. For such scenarios, we could implement an intermediate tag supplier for getting the data for all tags.

The below image, provides you a scenario where the microservice processing the time-series data and pushing to Azure Cosmos DB, takes the incoming tag names and updates them in Azure Table Storage with a mapping against the device. During reads, if the tag name is not provided, the service queries that tag names for a particular device from Azure table and utilizes the tag names as input to query the Azure Cosmos DB. This way, Azure Table Storage acts as a provider layer.

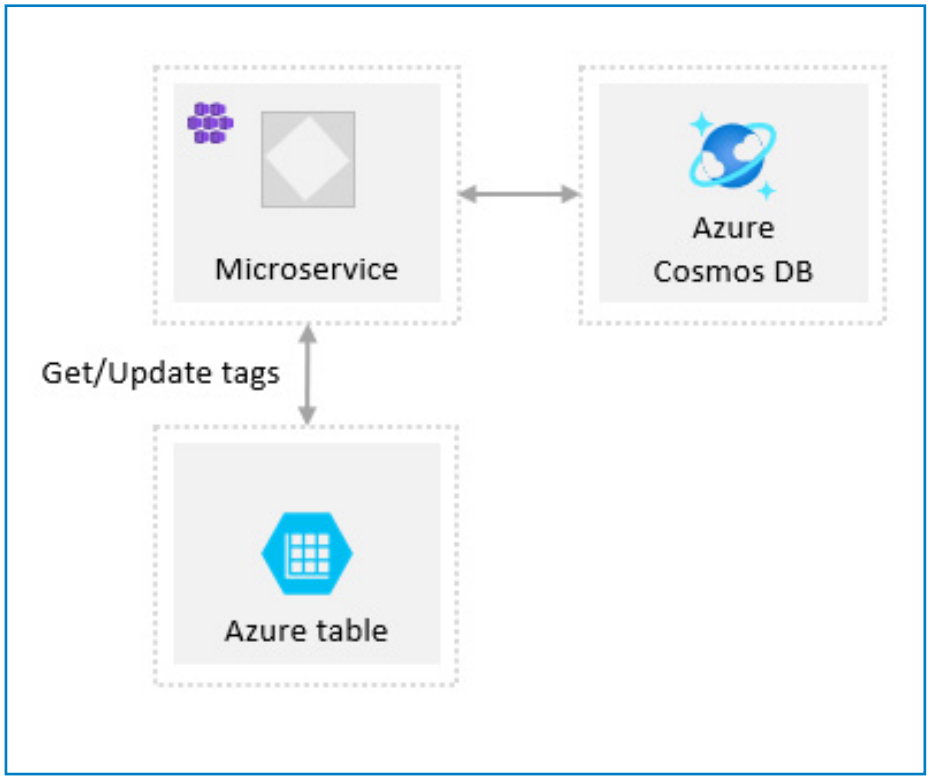


Figure 2. Processing Scenario

Reference Architecture

Azure Cosmos DB throughput is measured in request units per second (RU/s). RU/s are provisioned at the key-space level with manual scaling. Additional monitoring of actual RU/s consumption is implemented to scale the amount of RU/s as required.

For testing the overall storage performance, custom data ingestion was implemented. The custom application generated messages of 8-10KB in size. The overall solution was tested at various scales of 10,000 rows/min up to 1,500,000 rows/min into Azure Cosmos DB.

For 1.5M records, it was noticed that the average time to insert around 50+ tags was around 110-150ms with P95 < 10ms and P99 < 20ms. The actual RU/s utilized for this scale was around 135,000 RU/s.

Use case

In this section we will drill into one of many potential use cases of this solution. Consider a vibration sensor and a temperature sensor associated to a gateway deployed at the field location. The drilling machine will generate a certain level of vibrations that is considered normal and operates at an optimal temperature. Higher than normal vibrations or deviation in temperature is considered a potential issue that could be hazardous to the drilling machine. In such cases, we want the rotations on the drilling machine to reduce or completely cease to bring the vibrations/temperature under control.

The IoT gateway collects the data from sensors and continuously reports the data to the cloud. The processed data is finally stored in Azure Cosmos DB as individual records identifying the customer, device, the tag (i.e., temperature/vibration frequency/rotations per minute values) with appropriate time stamps.

The data stored in Azure Cosmos DB is further utilized for decision making and take corrective actions based on the scenario and requirements.

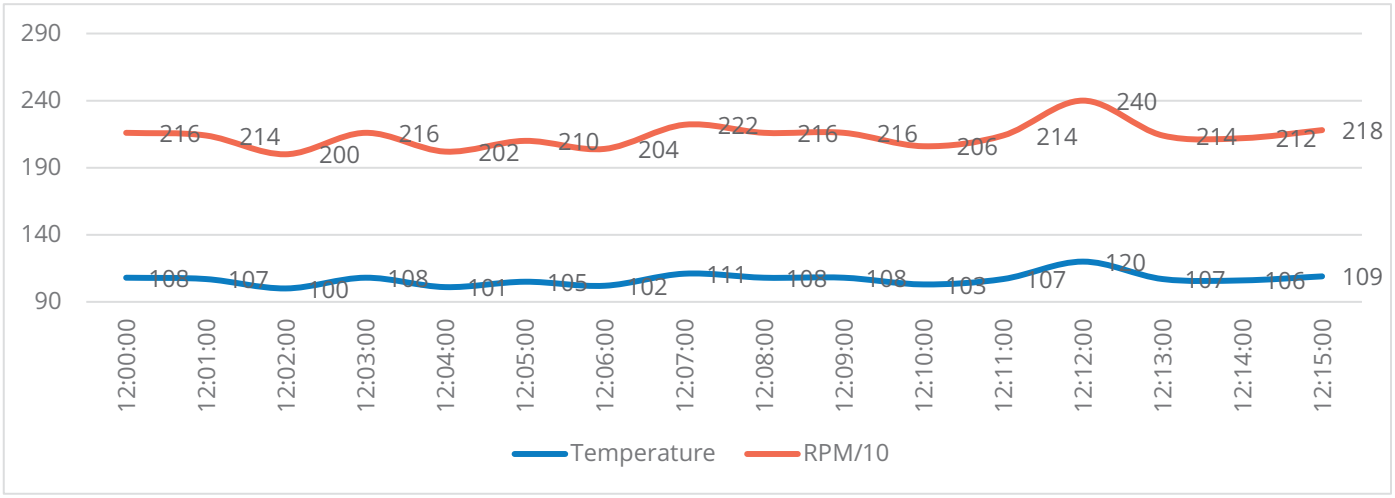


Figure 3. Telemetry data

For example, in this case, the intent was to maintain the temperature of the driller below 120 Celsius. However, as the RPM of the driller varied, the driller hit the defined threshold. The IoT device monitored the telemetry and reported it to the application.

This information was further processed by custom application to send a controlled action back to the IoT gateway to reduce the RPM, bringing the temperature back under control.

Backup and Restore

Backup: Azure Cosmos DB provides default option of taking 2 backups every 4 hours, the same was configured for this solution.

Restore: Microsoft Azure support request is the only way to restore the data. In case you are initiating a support request to restore point in time data, it is advised to disable the backup process to ensure the data is not overwritten and a support request is raised within the retention time. Microsoft team does not provide any SLA for the time it takes to do a restoration as the amount of time it takes depends on how much data is to be restored

Key Azure Cosmos DB design considerations in the solution

- Individual table for each customer: Azure Cosmos DB provides us the ability to provision RU/s for each table. Each customer has their data independently stored in their own table and can be scaled independent of other customers. In scenarios where a particular customer has significantly higher number of gateways, the specific customer table can be provisioned with additional RU/s as needed. This can also be helpful in the case of a business-critical event, where there is a need to increase telemetry frequency, individual tables can be scaled, and the charges could be tracked for each customer.
- RU/s allocation and configuration: Azure Cosmos DB charges are based on the Request Unit (RU) consumed/allocated per second.
    - If the data ingestion is at a predictable rate, it is advised to use standard (manual) provisioned throughput.
    - If the data ingestion is un-predictable, auto scale throughput is advised. This means the maximum expected RU/s will be defined. This can be monitored to ensure the actual RU/s are below the maximum expected RU/s.
    - With the key space level RU allocation, every table within the key space will be able to utilize the available RUs if it requires. For example, if we have 4 tables within a key space, the allocated RUs are equally made available for all 4 tables. If one of these tables tend to utilize more than the initially awarded RUs, they can do that provided the remaining tables have not completely consumed the allocated RUs.
    - Along with the key space level RU allocation, we can additionally allocate RUs at individual table level. This will provide the additional support in cases where a specific table might need additional RU allocation
    - In use cases where we have limited deviation in the ingestion pace, the RU requirement is predictable and is more controlled. In such scenarios, allocating RUs at the key space level will support most of the scenarios. This is more beneficial compared to allocating RUs at individual level for use cases like this. This is because, with varied ingestion pace across multiple tenants, not all systems will need the same number of RUs dedicated all the time. With the RU allocation at key space level, you can potentially support for peak usage without having to allocate RUs at table level for peak usage.
- Automating throughput provisioning: Considering the desired RU/s was manual for this solution, during peak periods for the customers, additional throughput was allocated at table level via automatically executed scripts. The actual RU/s consumed was monitored and when it reached 70% of the provisioned amount, additional throughput was added to ensure the overall solution is not impacted.
- Archive strategy: Considering the allocation of minimum RU/s is dependent on the read-write frequency and on the stored data in the database. As the storage increases, the required minimum RU/s increases as well. It is essential to have an archival and TTL strategy that will enable us to optimize the throughput utilization. The change feed feature in Azure Cosmos DB can help in moving the data out of Azure Cosmos DB into blob storage.

About the Author

Mandanna Appanderanda Nanaiah
Principal - Enterprise Applications; CLOUD PROFESSIONAL

About the Mentor

Ravi Joshi
Principal Technology Architect; OPEN SOURCE PROFESSIONAL

For more information, contact askus@infosys.com

Infosys®
Navigate your next