December 2019

Cloud Based IoT Architecture

NATHAN ROEHL
University of Wisconsin-Milwaukee

CLOUD BASED IOT ARCHITECTURE

by

Nathan Roehl

A Thesis Submitted in

Partial Fulfillment of the

Requirements for the Degree of

Master of Science

in Computer Science

at

The University of Wisconsin-Milwaukee

December 2019

ABSTRACT

CLOUD BASED IOT ARCHITECTURE

by

Nathan Roehl

The University of Wisconsin-Milwaukee, 2019
Under the Supervision of Professor Tian Zhao

The Internet of Things (IoT) and cloud computing have grown in popularity over the past decade as the internet becomes faster and more ubiquitous. Cloud platforms are well suited to handle IoT systems as they are accessible and resilient, and they provide a scalable solution to store and analyze large amounts of IoT data. IoT applications are complex software systems and software developers need to have a thorough understanding of the capabilities, limitations, architecture, and design patterns of cloud platforms and cloud-based IoT tools to build an efficient, maintainable, and customizable IoT application. As the IoT landscape is constantly changing, research into cloud-based IoT platforms is either lacking or out of date. The goal of this thesis is to describe the basic components and requirements for a cloud-based IoT platform, to provide useful insights and experiences in implementing a cloud-based IoT solution using Microsoft Azure, and to discuss some of the shortcomings when combining IoT with a cloud platform.

1 Introduction

IoT applications are complex software systems that include software components for devices, gateways, message aggregation, data analysis, storage, and visualization. As an emerging technology, IoT has been used in areas such as predictive maintenance [43, 127], smart metering [1, 39], asset tracking [129, 130], connected vehicles [9, 35, 117], and fleet management [10]. These applications collect vast volumes of telemetry messages from distributed IoT devices and processed them in real-time to gain business insights and to take actions based on the insights. The sensitive data collected by IoT devices must be protected during data transmission and processing. Stable storage is needed to archive historical data. Scalable data analysis tools are needed to learn knowledge in real-time from streaming data or as offline analysis from historical data. Data visualization and graphic user interface are needed to provide visual feedback and user control of data analysis and IoT devices [36, 120]. Each one of these components is a software program that requires significant engineering effort. Composing them so that they work together properly is even more challenging. In this context, developers of IoT applications may find it more productive to use existing IoT frameworks and other commercial off-the-shelf software components to build their final products.

1.1 IoT platforms

In recent years, many IoT platforms have emerged to answer the demand for rapid prototyping of robust and scalable IoT applications. These IoT platforms include cloud-based system such as AWS, Google, Azure, IBM IoT [36, 118, 120] and standalone systems such as NodeRed [15], Thingworx [36, 120], and Thingsboard [38]. While differing in interfaces,

implementation, performance, and costs, these IoT systems all offer device clients, messaging hubs, stable storage, data analysis modules, data routing mechanisms, and data visualization tools. From the users' perspective, experience gained from working with one platform can be translated to others since these IoT platforms have similar design principles.

IoT applications are large software systems that are difficult to develop from the ground up. IoT platforms are designed to alleviate this burden. Despite the recent advances of IoT platforms, the complexity of IoT applications should not be underestimated. IoT applications have to be compatible with diverse and distributed sensors, able to process large volume of data in real-time, be tolerant to missing or erroneous data, runtime exceptions, and unreliable networks, scalable to big data analysis and storage, and extensible to complex workflow and business logic [36, 120]. As common to any complex software, utilization of appropriate software architecture and design patterns are essential to the success of IoT development projects. When making choices of the architecture and design patterns, it is important to understand the requirements of the IoT applications and balance them against the performance, limitations, and design tradeoffs of the software components of an IoT platform [36].

1.2 Requirements of IoT Applications

Different types of IoT applications have different functional and non-functional requirements. Based on these requirements, developers need to select the suitable architecture, design patterns, and software components to be used on an IoT platform. Functional requirements refer to the specific functionalities of a software system while non-functional requirements refer to the properties of the software application such as performance, security, fault tolerance, and cost.

For example, predictive maintenance and smart metering applications require device monitoring, abnormal event detection, and real-time and offline data analysis [1, 39, 43]. For applications such as asset tracking and fleet management, spatial locations of the sensor devices needs to be maintained [10, 129, 130]. For connected vehicles, field gateways are often required to collect data from vehicles of each road segment and generate traffic alerts and send aggregate information to the cloud servers [9, 35, 117]. For connected vehicles, there are hard real-time constraints, which require low latencies [9, 35, 117]. For smart metering and asset tracking, high throughput is needed to process large volume of telemetry data [1, 39, 129, 130]. For asset tracking and fleet management, offline analysis such as mixed integer linear programming is needed to find optimal scheduling [10, 129]. For predictive maintenance, offline analysis such as machine learning is needed to generate models that can be used to suggest proactive measures in real-time [43]. All these applications require graphic user interface (GUI) for data visualization and control, user management, device management, device provisioning, and data security. While functional requirements often are related to individual IoT components, the non-functional requirements such as data security, latency, scalability, and fault tolerance are cross-cutting concerns that affect the entire IoT system. Below is a summary of the functional and non-functional requirements of IoT applications.

1. Functional requirements

    (a) Real-time and/or offline data analysis.

    (b) Sensor data archive. Some types of data may be compressed.

    (c) Requirement and availability of gateways for IoT devices.

    (d) Real-time control of IoT devices.

    (e) GUI for data visualization and control.

(f) Data processing workflow and/or business logic.

(g) Distributed machine learning.

(h) Device management and provisioning.

(i) User management.

2. Non-functional requirements

(a) Scalability to large number of IoT devices or telemetry messages.

(b) Strong data security.

(c) Hard real-time constraints with low latency or high throughput with soft real-time constraints.

(d) Tolerance of faults such as erroneous data and unreliable network.

(e) Cost requirements.

1.3 Cloud-based IoT platform

IoT applications can be implemented using on-premise servers or cloud-based platforms. The choice mostly depends on the non-functional requirements of the application such as latency and security. Cloud-based software systems have gained tremendous growth in recent years with many legacy software systems migrated to cloud platforms to offer software as services rather than installation-based products. The benefits of cloud-based software systems include transparent updates, distributed and load-balanced processing, scalable computing and storage, and flexible offering of diverse products and services [111]. The drawback is that users of the cloud-based system need to maintain network connectivity for some product features to function correctly. Also, unstable network or latencies of a slow network can cause unpredictable issues for applications with hard real-time constraints.

Fundamentally, a cloud platform is a collection of network-connected distributed computing and storage servers that provide virtual environments for different operation systems, application containers (such as Google's *app engine*), and computing services [111]. A software system can run on a cloud platform by deploying its instances on virtual machines of the cloud platform, though the advantage of this is limited to eliminating the need to acquire and maintain physical hardware. Users still have to set up the required computation environments such as dependency libraries, networking, and storage within the cloud servers. Application containers reduce some of these burdens in that the containers provide interfaces to computation environments that users do not need to separately set up for each deployment of the software system on a cloud platform.

Utilization of cloud-based platforms can be divided into four types: IaaS, PaaS, FaaS, and SaaS with increasing level of dependency on the cloud platform [28, 30, 111].

1. IaaS (Infrastructure as a Service) is the ability of a cloud platform to act as virtual machines for software applications that normally run in local servers. *Amazon EC2* (Elastic Compute Cloud) is such an example, where users can deploy software applications by launching instances of virtual servers, uploading the applications, and executing them. The virtual machines can be distributed and can use stable storage such as *Amazon S3* (Simple Storage Service) buckets.

2. PaaS (Platform as a Service) is a managed computation environment for specialized applications such as Web servlets. Google's *app engine* is such an example, where users can develop software programs using *app engine* development tool and deploy the programs to the *app engine* to execute. In PaaS, users still need to develop complete software programs as in IaaS except that users rely on the API of the platform for runtime support, network, and storage.

3. FaaS (Function as a Service) such as *AWS Lambda* and *Azure Functions* allow users to implement lightweight applications as stateless functions, which can be used for high throughput processing such as data transformation, filtering, and event detection. FaaS functions connect to storage and network through APIs. FaaS functions are usually considered serverless since there is no dedicated servers allocated for running the functions and the cost is based on calls to the functions.

4. SaaS (Software as a Service) refers to software applications hosted by cloud servers such as Dropbox, Microsoft Office 365, and Google Apps that deliver functionalities over the network.

Most of the software components of a cloud-based IoT platform fit the mode of PaaS (e.g. Azure *Event Hubs*, *Cosmos DB*, *Redis Cache*, and *Stream Analytics*), FaaS (e.g. *Azure Functions*), or SaaS (e.g. *Azure IoT Central*). *Azure IoT Central* is a software service for developing IoT applications through a Web interface, which is supposed to be simple to use with the least amount of control and customizability. PaaS and FaaS are used for customizable IoT applications that can be reasonably handled by the services such as *IoT Hub* and *Stream Analytics*. Solutions based on IaaS are usually reserved for customized and complex IoT applications that require a high-level of control. Developers can utilize open-source software stack such as Spark, Mesos, Akka, Cassandra, and Kafka to build customized IoT applications and deploy them in IaaS virtual machines. Since IaaS-based IoT applications need to reserve virtual machines, it is more suitable for computation-intensive applications with high rate of utilization of the virtual machines. SaaS such as *Azure IoT Central* is intended for simple IoT applications with limited functionalities while IaaS-based IoT applications are essentially locally developed applications deployed on IaaS virtual machines. Therefore, this research will focus on cloud-based IoT components provided in forms of PaaS and FaaS.

The contributions of this thesis are as follows.

1. The IoT landscape is constantly evolving, where past survey papers are either out of date or their scope of research focused on a few key aspects and not the entire system. This thesis presents a current in-depth analysis and description on many of the capabilities, limitations, architecture, design patterns, and services of modern cloud-based IoT platforms and applications, providing a better understanding of each part and how it relates to the whole.

2. Cloud platforms can offer an overwhelming amount of services to choose from, creating a steep learning curve for new users. Chapter 5 gives guidance for how to navigate a large cloud platform such as Microsoft Azure, and use it to implement an IoT application and avoid many of the pitfalls that are not immediately noticeable.

3. The combination of IoT with cloud platforms create an accessible, resilient, and scalable solution that can handle many devices and can store and process large volumes of data. The combination is not without its shortcomings, as there are limits to throughput and query efficiency. Two proposals are described, one distributed and one unique to the scenario focusing on reducing costs.

4. Lastly, corporations such as Amazon, Microsoft, and Google have become leading providers for cloud-based IoT platforms as demand for IoT solutions from industries such as healthcare, automotive, industrial, security, and many more increase. Many of the services from each platform are compared, showing current similarities and differences between the platforms.

The remainder of this thesis is organized in the following order. Chapter 2 reviews prior surveys on IoT systems. Chapter 3 describes the main aspects of IoT platforms using

Microsoft's Azure as a model. Chapter 4 shares our experiences using Microsoft Azure to create a small IoT application. Chapter 5 discusses the short comings of cloud-based IoT platforms and possible solutions. Chapter 6 compares the services offered by Amazon, Microsoft, and Google with respect to their cloud-based IoT platforms. Chapter 7 concludes. Portions of this thesis were published in a chapter for a book, where I was a contributing author [11].

3 Azure IoT Platform

The architecture of cloud-based IoT platforms are mostly driven by the functional requirements of IoT applications. The main requirements are to collect and analyze information collected from IoT sensors, to store the collected information, to report the results of analysis to users, and to integrate the analysis results with business systems [40]. The non-functional requirements such as security and latency are cross-cutting concerns that impact multiple components of the IoT platform. In this chapter, we discuss the main components of an IoT application, design choices, and related crosscutting concerns using Azure IoT platform for examples and as a guide.

Azure IoT platform refers to a suite of software tools offered on Azure cloud platform that can be used in combination to build IoT applications. While some software tools such as Azure *IoT Hub* [62], *Stream Analytics* [77], and *Power BI* [80] are designed IoT applications, others such as *Blob storage* [61], *Cosmos DB* [78], and *Azure Functions* [47] are for general-purpose applications but frequently used in IoT applications as part of the solution. The overall architecture of a cloud-based IoT system is shown in Figure 3.1.

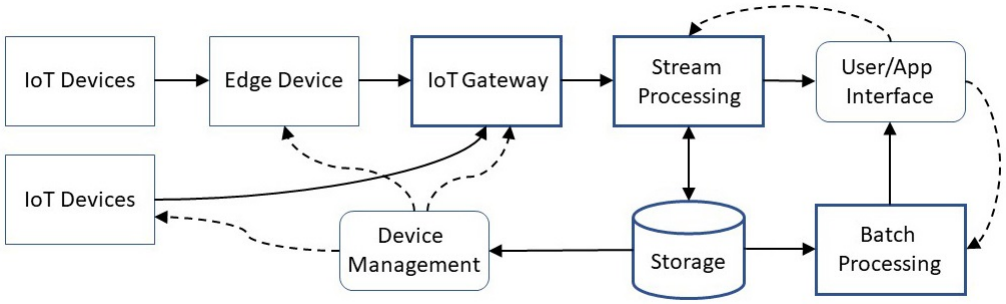

Figure 3.1: Cloud IoT platform architecture, where solid line denotes direction of data flow and dashed line denotes direction of control.

3.1 IoT Devices and Gateways

The primary functionality of an IoT application is to enable IoT devices to send telemetry messages to a cloud gateway which enables further processing of the messages. To implement this functionality, a messaging protocol must be chosen for communication between IoT devices and cloud gateways. IoT client must be implemented for the IoT devices so that they can establish secure communication with a specific cloud gateway. In order to manage the IoT devices, an IoT application must also maintain storage for device identities, device entities, and device provisioning.

Messaging protocol IoT devices communicate with IoT gateways through messaging protocols such as MQTT, AMQP, and HTTP as shown in Figure 3.2.

1. MQTT (Message Queue Telemetry Transport) is a lightweight client-server messaging protocol. It has a small memory footprint on the device and sends compact messages that use less network bandwidth [119]. Thus, MQTT is a popular choice for devices with limited computing resources and network connectivity. However, the compact messaging format also presents challenges to IoT applications. Since MQTT does not support message metadata in its header, IoT clients and gateways must agree on the

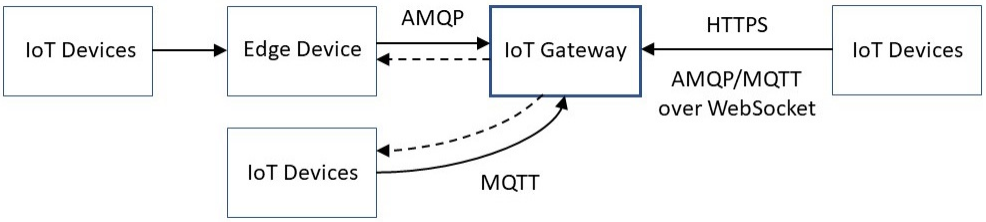

Figure 3.2: Communication protocols between IoT devices and cloud gateway, where solid line denotes the direction of device-to-cloud messages and the dash line denotes the direction of cloud-to-device messages

format of the message content. MQTT has three levels of *quality of service* related to the message delivery: *at most once*, *at least once*, and *exactly once* delivery [114]. Azure *IoT Hub* supports the first two but not the last one since it increases the latency while reducing the availability of distributed IoT infrastructure.

2. AMQP (Advanced Message Queuing Protocol) is a connection-oriented, multiplexing messaging protocol with compact message format, which is suitable for devices that require long-running connections and high-throughput communication with cloud gateways [112]. Both AMQP and MQTT are implemented over TCP layer. While MQTT is more lightweight and suitable for devices that connect to cloud gateway directly with per-device credentials, AMQP allows more complex device topology so that multiple devices can connect with cloud gateway using the same secure connection.

3. IoT devices can also directly communicate with Azure *IoT Hub* via HTTP though the implementation is more complex and the message overhead is higher than the binary format of MQTT and AMQP messages [112]. Moreover, cloud-to-device messaging is less efficient over HTTP since the device has to periodically poll messages while AMQP and MQTT can push cloud messages to a device with far less latency [112]. AMQP and MQTT can run over WebSocket for devices with restriction on open ports.

Device Client There are a variety of IoT devices with different processors, operating systems, networking capabilities, and runtime support of programming languages. For example, Intel NUC runs Ubuntu Linux while Raspberry Pi can run Windows 10. For devices that run an operation system, IoT clients may be implemented using Azure IoT SDK [55] using various programming languages such as Java, C, .NET, Python, and JavaScript (Node.js). For the devices supported by Azure IoT SDK, the implementation of IoT clients is a relatively straightforward process that first establishes connection to cloud gateway with device

credentials and then starts the loop of encoding messages and sending them asynchronously to the gateway. For FPGA-based devices such as CompactRIO, IoT clients can be developed using vendor-specific languages such as LabView, though such client may not be compatible with specific cloud gateways such as Azure *IoT Hub* due to its security requirement.

Cloud Gateway IoT devices may be connected to cloud gateway directly or connect through field gateways such as edge devices, which then connect to the cloud gateway. In the former case, IoT client program runs in IoT devices while in the latter case, the IoT client runs in the field gateways. Field gateways may be used in cases where the IoT devices are low-power sensors that lack the computing power and network connectivity to reliably transmit IoT messages. Field gateways may be used to aggregate, filter, and preprocess raw telemetries from IoT sensors to reduce network congestion and workload of cloud gateway. Field gateway may also be used to connect to devices such as CompactRIO that lack suitable implementation of secure communication layers or to devices that can only be connected with industrial protocols such as OPC UA. In some cases, a custom cloud gateway may be developed using *Azure IoT protocol gateway* [48] to perform the functions of a field gateway where the connection from IoT devices to the custom gateway may be secured using network tunneling such as VPN (Virtual Private Network).

Azure platform supports two types of IoT gateways: Azure *IoT Hub* and Azure *Event Hubs* with available connection patterns shown in Figure 3.3. Azure *IoT Hub* is more suitable for applications that require bidirectional messaging between devices and cloud gateways [62]. Azure *Event Hubs* is more suitable for applications that only need to send large number of messages to cloud gateway [58] and it does not support MQTT protocol. MQTT is based on publisher-subscriber model where message producers register as publishers in a MQTT broker and message consumers also register at the broker to listen on messages of

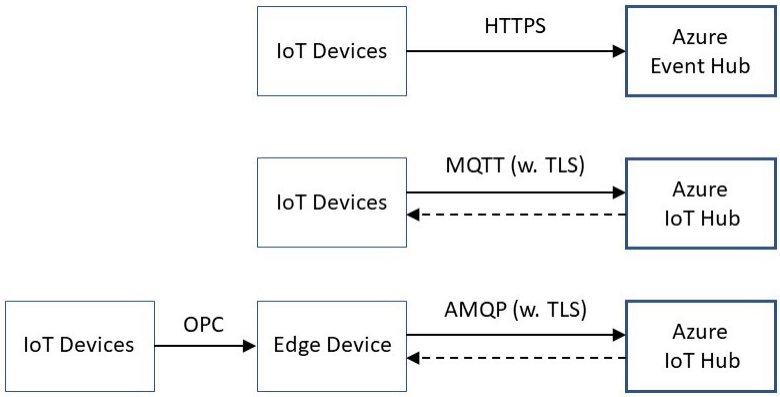

Figure 3.3: Communication patterns between IoT devices and Azure *IoT/Event Hubs*, where solid line denotes the direction of device-to-cloud messages and the dash line denotes the direction of cloud-to-device messages

selected topics. Azure *IoT Hub* is such a broker except that MQTT protocol does not specify security requirement. To protect sensitive IoT data transmitted over untrusted networks, Azure *IoT Hub* requires connections be established over TLS (Transportation Layer Security) protocol [60]. In order to connect to *Event Hubs*, the clients need to have the access keys associated with the *IoT Hub* account that they will connect to. Applications can retrieve the messages through *IoT Hub* using the same access keys.

After installing IoT clients on devices and setting up cloud gateway such as Azure *Event Hubs*, developers need to make sure that only authorized devices can be connected, the messages from the devices can be understood by the rest of the application, and the devices can be managed and controlled remotely through cloud gateway and other components. To this end, an IoT application needs to have device identity store, device provisioning service, and device and application models as shown in Figure 3.4.

While developers can configure small number of devices manually, for commercially produced IoT devices such as smart meters and smart thermostats, bulk processing of the device

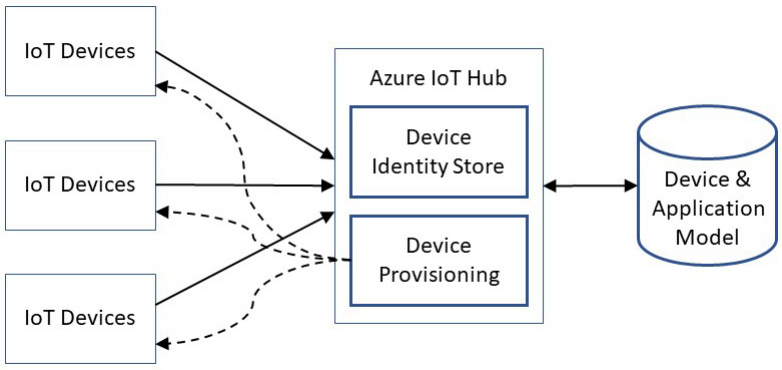

Figure 3.4: Relation between IoT devices and device identity store, device provisioning service, and device and application model.

configuration and setup may be necessary. Azure *IoT Hub* provides *Device Provisioning Service* (DPS), which is a global service to support device registration and configuration [76]. DPS provides API for device configuration and registration using Azure *IoT Hub*'s *device identity store* to provide per-device security credentials for authentication and authorization. DPS can also automate the registration of the devices to store the device model and application model.

The device model defines the schemas of device metadata, output, control parameters, and control actions. The application model contains the semantic relation between devices such as device topology and the relation between device and other application components. The device and application models are core design abstractions of each IoT application. For example, devices such as smart thermostats are modeled in relation to the room and building where they are located. A building management system may define a static hierarchical structure to model the devices based on their physical locations [130]. For applications such as fleet management, the device topology is more dynamic since the IoT devices may change their grouping as the vehicles carrying the devices are assigned to different tasks. Regardless of the specific designs, these models should be made persistent in storage such

as SQL database or Azure *Cosmos DB*, which includes a graph API for convenient query of device topology.

3.2 Storage

IoT applications generate increasingly large amounts of data that must be processed in real-time or by offline batch applications. Storing IoT data in an on-premise database is not suitable for industrial IoT applications due to the lack of scalability of local storage. Cloud-based storage offers elastic capability that can scale up without significantly degrading performance [111]. Cloud-based storage can be divided into warm and cold storage with some tradeoff between latency, query capabilities, and cost. Warm storage such as *Cosmos DB* and *Azure SQL Database* offer lower latency and flexible query interface but at a higher cost. Cold storage such as *Blob storage* and *Data Lake Storage* have higher latency and simpler interface but lower cost. Figure 3.5 shows the possible dataflow path between cloud gateway, data processing computation, and storage.

The distribution of data between warm and cold storage should be based on the requirement of IoT applications. To reduce cost, data that is not frequently used should be stored

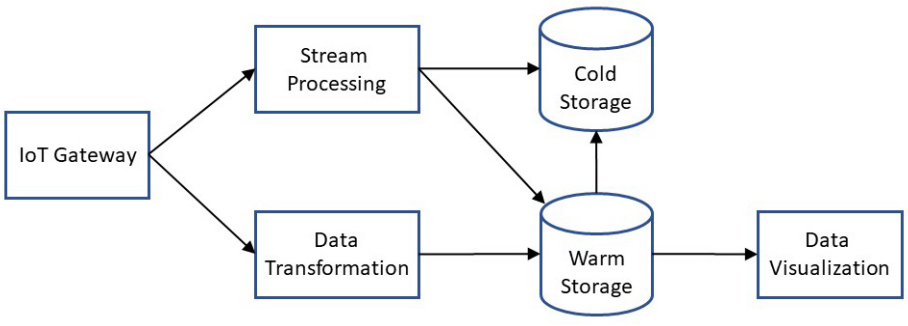

Figure 3.5: Dataflow between IoT gateway, data transformation, stream processing, cold storage, warm storage, and data visualization modules.

in cold storage. Data used in real-time processing components such as *Stream Analytics* and data visualization should be kept in warm storage [71]. A straightforward way of managing data for different storage is to store recently-transmitted data in warm storage first and then gradually migrate old data into cold storage after a predetermined amount of time has passed.

There are a few options for warm and cold storage on Azure IoT platform as shown in Figure 3.6. For warm storage, low latency is required. Azure provides *Cosmos DB*, which is a NoSQL database that supports five query methods including traditional SQL, MongoDB, graph, table, and Cassandra Query Language (CQL). Data in *Cosmos DB* can be set with an expiration date, after which the data is automatically deleted. Another option is *Azure SQL Database* [79], which provides relational database model with transactional support, which may be necessary for applications with strong integrity constraints. Both databases have flexible query interfaces but they also have higher cost than that of cold storage such as *Blob storage* and *Azure Data Lake Storage* [74], which stores data as files or objects and uses

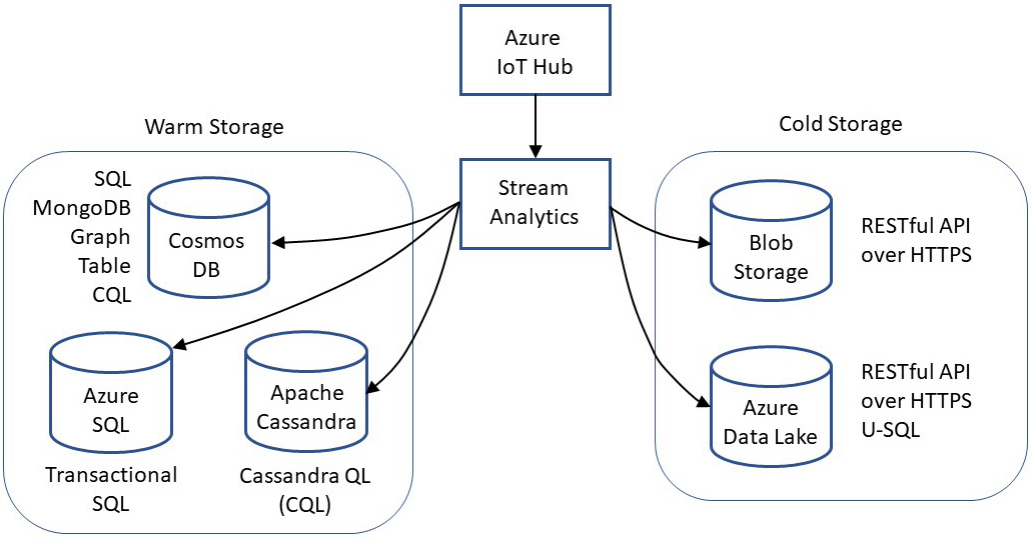

Figure 3.6: Options of warm and cold storage with different query methods.

RESTful API for query access. Query interface to cold storage is much more restrictive than that of warm storage. Thus, for computation such as visualizing power usage of a freezer over the past 30 days, warm storage should be used since the computation needs to issue queries that can return precise results with low latency. For computation such as analyzing the average power consumption of all freezers over the past year, cold storage can be used since the analysis can be computed offline with aggregate data set as input.

3.3 Data Analysis

The main purpose of IoT applications is to learn insight from data emitted from IoT devices and use the insight in making logical decisions towards business objectives [126] such as reducing operating cost of fleet vehicles or preventative maintenance of industrial equipment. Data analysis is the crucial step in realizing this goal. Depending on applications, IoT data analysis can be online processing of data streams or off-line processing of data batches. For online processing, the computation can also be divided as stateless or stateful. Figure 3.7 shows the possible dataflow paths for the different types of IoT data processing on Azure platform.

For online processing of IoT data, latency is an important consideration [125]. Latency in this context refers to the amount of time it takes to analyze the IoT data collected over a time window (such as 5 seconds) to generate the input for a backend application such as a data visualization tool. For example, latency in the scale of seconds is not suitable for applications such as connected vehicles, which may have real-time constraints in milli-seconds. Throughput is another consideration for online processing. Throughput refers to the amount of data that can be processed within a time period. Applications with high data ingestion rates require sufficient throughput to handle incoming telemetry data to avoid data

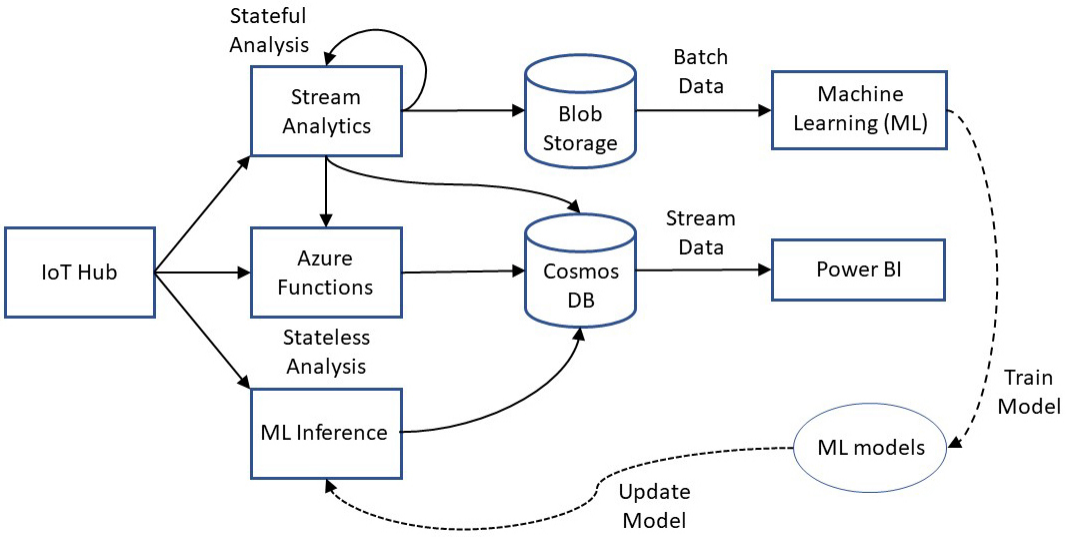

Figure 3.7: Possible dataflow paths for IoT analytics.

loss or excessive latency. Insufficient throughput can increase latency since unprocessed data has to be buffered and if the buffer overflows, unhandled telemetry data is dropped.

An application can reduce the requirement of throughput by slowing down data ingestion rates using methods such as sampling or aggregation. Cloud-based IoT platforms such as Azure can usually satisfy high throughput requirements of IoT applications since users can allocate more resources to process data in parallel. However, latencies of these platforms are more difficult to reduce. For applications that demand near real-time response time, on-premise data processing is more appropriate. For example, field gateways can be used to perform preliminary data integration, data analysis, and aggregation before sending the lower frequency data to the cloud gateway [29] such as Azure *IoT Hub*. This also can reduce the computation cost associated with cloud platform usage and the network bandwidth.

Online data processing includes data transformation, data filtering, event detection, alert generation, data aggregation, and other streaming data analysis such as Key Performance Indicator (KPI) calculation, classification, and anomaly detection. Online data processing may be stateless or stateful computation. Stateless computation such as data transformation only needs its input data at time t to obtain its output at time t. Stateful computation needs to use both its input at time t and its states at previous time stamps such as $t-1$ to compute its output and states at time t.

Azure includes two types of data processing systems: *Azure Functions* and Azure *Stream Analytics*. Both systems can take streaming inputs from *IoT/Event Hubs* and output results to Azure storage and reporting tools such as *Power BI*. *Stream Analytics* implements query computation on data streams while *Azure Functions* provides general-purpose computation triggered by events. A common design pattern is to use *Stream Analytics* to preprocess data streams from *IoT Hub*s and then forward the results to *Azure Functions* for advanced processing.

Azure *Stream Analytics* provides a domain-specific way for processing IoT data streams using a SQL-like (stream analytics) query language [46] that treats data streams as tables, stream events as records, and stream event fields as record fields. For example, `select V, I from Power where V > 10` returns a stream of values that are the fields `(V, I)` from the stream `Power` where value of `V` is greater than 10. Nested queries can be made using *with* clause to define result sets of inner queries to be used in the outer query. In this query language, streams can be joined with the addition of an *on* operator that specifies the time bound of the joined stream events. The time bound is calculated with the *datediff* function using the timestamps of the stream events. Selected stream events can be *group*ed *by* field names with the addition of a *window* function or system timestamp. A

window function specifies the grouping of stream events based on the event timestamps so that aggregate function can be applied to the events in each time window to output a single event. Azure *window* functions include *tumbling window* (fixed-sized and non-overlapping), *hopping window* (with fixed overlap), *sliding window* (all distinct windows with fixed size), and *session window* (events grouped by similar timestamps) [75]. This query language also provides a list of builtin functions to support computation on scalars, aggregates, records, spatial values, and temporal values (e.g. lagged events, last event, and first event in an interval).

Azure *Stream Analytics* can invoke user-defined function (UDF) [56] written in JavaScript in its queries. UDF allows users to perform operations such as math, array, regexp, and remote data access, which is not supported by builtin functions of the query language. *Stream Analytics* can also utilize user-defined aggregate (UDA) [49] that specifies how values are (de-)accumulated over event sequences grouped by window functions in a query. UDA is defined as a function that constructs a JavaScript object with fixed-named methods. *Stream Analytics* can also use *Azure Functions* to perform additional processing by sending query results via HTTP request to an *Azure Function* in batches [70]. The batch size is limited to 256KB while each batch is limited to 100 events by default. Note that Azure *Stream Analytics* can perform both stateless and stateful data processing. Through *window* and *analytic* functions, *Stream Analytics* can perform stateful computation on event streams subject to time-based constraints.

Azure *Stream Analytics* jobs can be launched through Azure's Web interface by filling in the query statements in a textbox and specifying the input and output sinks of the analytics jobs. Launching a *Stream Analytics* job takes a few minutes while error messages are few and uninformative, which makes it difficult to perform testing since mistakes in query statement such as typo of field names may simply yield empty results. Azure *Stream Analytics* does,

however, allow faster tests by executing the queries on sampled datasets though this kind of testing does not reveal all possible faults with the queries. Also, the UDF and UDA associated with each *Stream Analytics* query are in separate scopes, which prevents sharing across multiple queries.

Azure Functions provide *serverless* computation triggered by events. *Azure Functions* can be implemented in languages such as Java, C#, and JavaScript to provide stateless computation in response to events from sources such as HTTP, *Blob storage*, *Event Hubs*. *Cosmos DB*, *Storage Queue*, and *Service Bus* [47]. *Azure Functions* allows the use of general-purpose programming languages to define complex logic to process event data and redirect the result to a destination such as *Blob storage*.

For stateful data processing, users can use durable *Azure Functions* [57], which is an extension of *Azure Functions* that implicitly uses Azure storage to manage stateful computation. As shown in Figure 3.8, durable *Azure Functions* includes *orchestration* function that can compose multiple *activity* functions to run in sequence or in parallel. Durable *Azure Functions* also includes *orchestration* client that can start a long-running process that can be monitored for progress via HTTP endpoints.

As an evaluation of the throughput and latency of high-volume event processing with *Azure Functions*, Microsoft conducted a performance test [50] to demonstrate some of the optimization strategies. This test partitioned a stream of weather and seismic event data into two streams to be processed independently. The stream processing used Azure *Event Hubs* to act as buffers, *Azure Functions* to separate the two types of event data and push them into their respective *Event Hubs* for further processing. In this test, a stream of 100,000 events per second was processed for 9 days. During the test, half of the events were processed with about 1 second of latency while 90% of the events were processed with less than 3

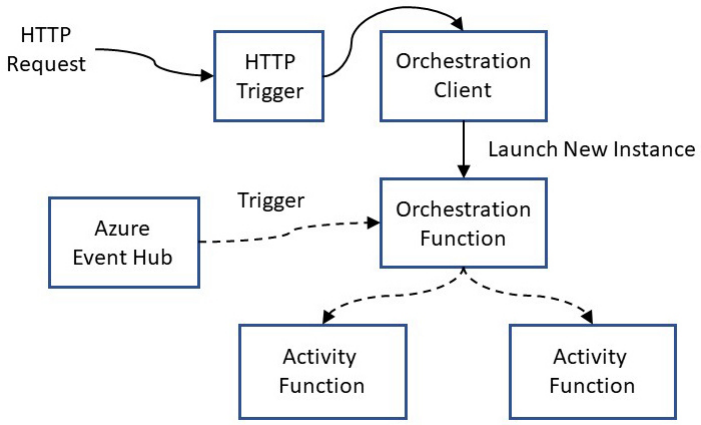

Figure 3.8: Stateful data analysis using durable *Azure Functions*.

seconds of latency. However, max latency was about 4 minutes. The test achieved this level of throughput by configuring each *Event Hub* with 100 partitions, which translates to 100 virtual machines that execute the *Azure Functions* in parallel. This test also processed the streaming events in batches instead of individual events in *Azure Functions*, which increases throughput by reducing the overhead of launching *Azure Functions*.

Offline data processing includes KPI calculation using historical data, clustering, regression, and classification using unsupervised or supervised machine learning. Many of these offline data analysis tasks can be completed in Azure *HDInsight* [72], which is a PaaS cluster platform for launching distributed computing services. Apache Spark, which is a distributed in-memory programming library, is available in *HDInsight* as a computation environment for big data analysis [69]. With Spark, users can develop parallel applications that can perform data analysis using data from storage facilities such as *Azure Data Lake Storage*, *Blob storage* and gateways such as *Event Hubs* (through Spark streaming). Spark utilizes a map-reduce programming model to support distributed computation and its memory-based resilient distributed dataset (RDD) abstraction provides better performance on big data than disk-based

distributed file system such as Hadoop does [122]. Spark supports offline analysis such as machine learning. For example, using Spark, an IoT application can train a regression model offline using historical data and use the model to make online predictions for streaming data as shown in Figure 3.7.

Note that Spark uses a cluster of machines to complete its tasks. When running Spark in *HDInsight*, a cluster of VMs of a minimum configuration such as a master and a slave must be created and the cluster will exist until the task completes. Keeping a cluster running is far more expensive than services such as *Azure Functions*. Thus, it is not cost effective to keep a long-running data analysis using Spark in *HDInsight*, which is more suitable for offline data analysis that is both computation intensive and has a finite duration.

3.4 User interface

Some IoT applications such as smart thermostats can use the results of data analysis to generate automatic control signals such as setting the thermostat temperature. However, for most applications, graphic user interface is needed to render data visualization and generate reports for business decision making. The interface can also be used to monitor the status of IoT devices and domain abstractions such as buildings that contain smart meters, to schedule an offline data analysis such as machine learning on historical data of certain time period, and to change an online data analysis such as the formula of a KPI. Azure IoT platform provides several PaaS and SaaS components for this purpose.

Visualization Azure *Power BI* is a versatile dashboard tool for data visualization and report generation of static or low frequency data. Using the graphic interface of *Power BI*, non-programmers can create dashboard with various data charts from a range of data sources. *Power BI* includes two versions: *Power BI Desktop* and *Power BI Service* (online).

Power BI Desktop has more functionalities in terms of visualization and report generation. However, only *Power BI Service* supports real-time streaming where its data ingest rate is limited to 5 requests per second for its streaming dataset and 1 request per second for push dataset [59]. Higher frequency data may be streamed or pushed to *Power BI* in batches with some added latency. For visualization and analysis on IoT data as time series, Azure *Time Series Insights* [51] may be used. *Time Series Insights* is also a type of warm storage that stores data in memory and solid state drives, though its data is only kept for a limited amount of time and its query latency is still too high for on-demand applications.

Web and mobile apps Azure *App Service* provides environment for building Web app, mobile backend, and RESTful APIs. Users can create a Web interface using Azure *App Service* to connect to IoT components such as Azure *IoT Hub* through WebSocket. For mobile devices with unreliable network and limited energy budget, push notification may be used to send IoT data such as alerts and event notification. To this end, Azure provides *Notification Hubs* [73], which allows applications to push messages to mobile devices of platforms such as iOS and Android. Azure *Notification Hubs* can also be integrated with Azure *App Service Mobile Apps*, which works by retrieving device PNS (platform notification system) handles, registering the devices with notification hubs, and sending notifications from app backend through notification hubs.

4 Experiences with Azure

The Azure Cloud offers a wide range of services, where each service belongs to a broader category to help with organization. As of this writing, there are 22 categories, where the number of services each category has ranges from as low as 1 to as high as 50. Many services are unique to each category, but some services do overlap and are placed in other categories as well. Information and examples of each service can be found at Microsoft's main Azure website. See Figure 4.1 for a view of the *Azure portal* and the many services offered. The wide range of services, categories, and amount of information online creates a steep learning curve to those unfamiliar with the Azure platform. To study the platform and ease of use, a set of tasks were selected to be performed on a data set. To illustrate the implementation of an IoT application using Azure platform, an experiment was conducted using Azure to

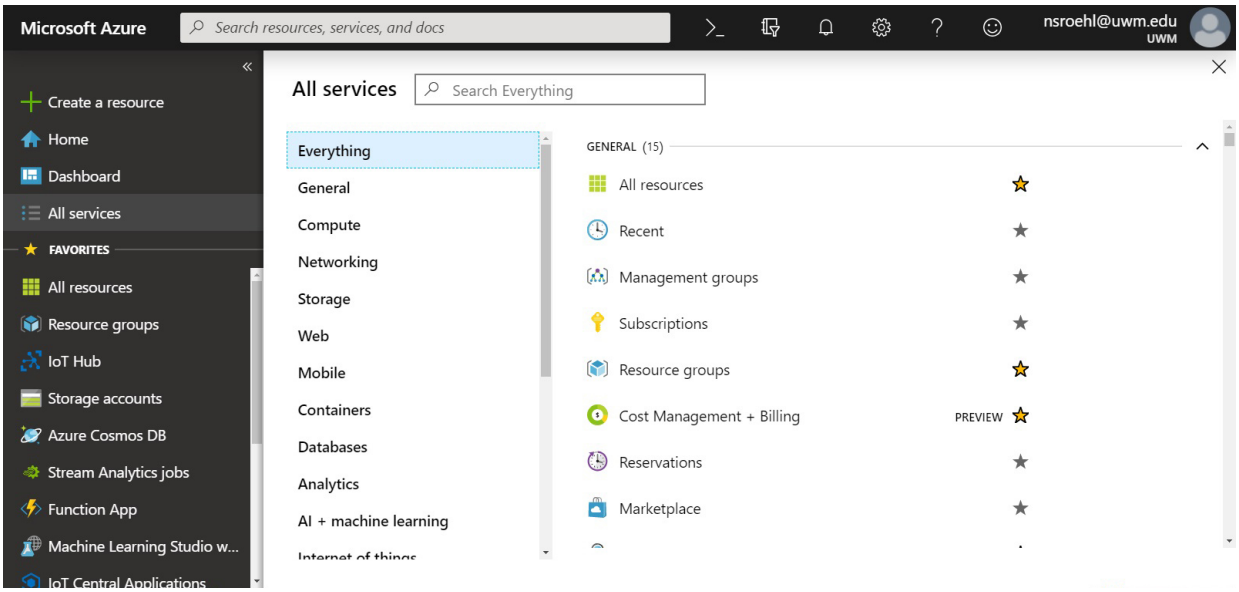

Figure 4.1: View of the *Azure portal*. The left most menu contains favorite services. When clicking on any of these tabs, a window opens to the right. These windows are called blades. The *All services* tab was selected, showing only a small portion of all available services.

analyze a set of time-series data of Li-ion batteries. The remainder of this chapter details the data set, the process, and our experiences using Azure. The experiences are based on the current iteration of Azure, which is always changing. Some features or services may not be available at the time this is read.

4.1 Data Set

The data set used details the operational use of a number of Li-ion batteries collected over an extended period of time by NASA [121]. This experiment used the data of battery number 5. The data was collected over many cycles, each of which includes a full charging phase followed by a full discharging phase while metrics such as current, voltage, temperature, sampling time, and the impedance were collected. As the number of cycles increased, the time to fully discharge the battery decreased. The data was originally in Matlab data files and was converted to CSV (Comma Separated Value) files before it is used for the simulation of IoT telemetries in the experiments.

4.2 Azure Set Up

The main goal of this experiment is to create a IoT system using Azure. The experiment simulated telemetry data from sensors connected to the battery and sent it to an Azure *IoT Hub*. The data is stored, processed, then analyzed during each cycle. The analysis will then reveal when the battery should be replaced as its capacity degrades over time. The list of different services used during the set up are *IoT Hub*, *Storage Accounts*, *Stream Analytics Jobs*, *Function App*, *Machine Learning Studio*, *Resource Groups*, and *Cost Management + Billing*.

To use any Azure service, a user must have an account along with a subscription. The

subscription keeps track of what services were used and how much to charge. A nice feature is that many of the services in Azure only charge a user when they are running. If a user is looking to cut costs, turning off a service during low-peak hours is a viable option. The user still must be careful as some services may incur charges just for creating a resource.

4.3 Resource Groups

Before going in to detail about many of the services and what was done, the importance of resources and Resource Groups must be stressed. When using any of these services, a resource from that service must first be created. For example, when using the *IoT Hub*, a user must create an *IoT Hub* resource first. Since multiple resources can be created from the same service, a unique name is given to each one to distinguish them. Once a resource is created, it must be assigned to a resource group. Resource group is a convenient way to store multiple resources together in one place. When creating some resources Azure may create other resources unaware to the user. The resources created automatically will appear in the same resource group, keeping related resources together. This makes it easier to locate, delete, and manage multiple resources. Creating a resource can be done via command line inputs using the *cloud shell* or through the *Azure portal*. Both work through a browser and allow a user to create and monitor resources, but the *cloud shell* requires the user to know all the commands where the *Azure portal* is an interactive graphic interface. *Azure portal* is used for the remainder of the chapter.

4.4 IoT Hub and Storage

After creating a resource group, an *IoT Hub* resource is needed as a location to receive messages [62]. An *IoT Hub* can also send messages, but this functionality is not required

for this experiment. When creating an *IoT Hub* resource, a tier must be selected to specify how many messages this hub is expected to receive. Tier selection helps Microsoft allocate physical hardware resources on their end to meet the users needs. The free tier was used in this experiment, as it provides sufficient throughput and is cost effective. See Figure 4.2 for the blade used to create an *IoT Hub* and Figure 4.3 for more details on *IoT Hub* tiers. IoT devices must be added to the *IoT Hub* after an *IoT Hub* resource is created. Each device requires its own device ID in the *IoT Hub* to distinguish one device from another. To add a device, select the *IoT Devices* tab under the newly created *IoT Hub* resource. Here a user can add devices where Azure will auto generate connection strings for each device. The connection string is critical in maintaining a secure connection between the device and the *IoT Hub*.

To send messages from a device, Microsoft provides many SDKs and example code to connect various types of IoT devices to the *IoT Hub* [100]. Various programming languages and

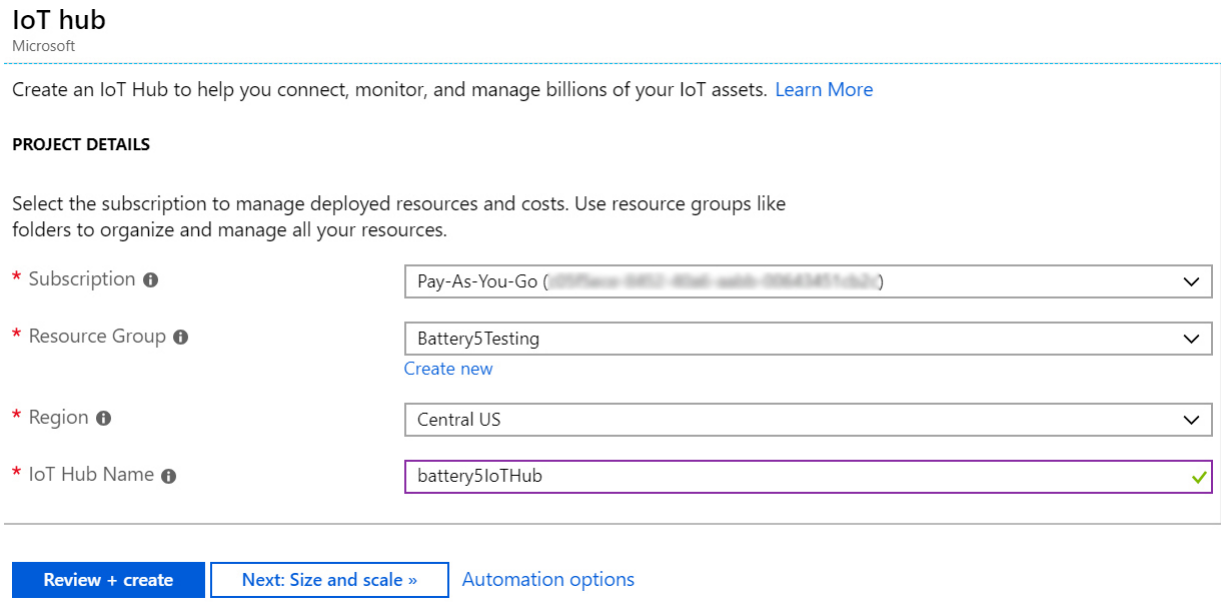

Figure 4.2: Creating an *IoT Hub*.

Tier Name	Messages per minute per unit	Message size per minute per unit	Number of units allowed	Cloud-to-device Messaging	IoT Edge Enabled	Device Management	Cost (USD) per month per unit
B1 : Basic Tier	278	1.111 MB	200	N	N	N	$10.00
B2: Basic Tier	4,167	16 MB	200	N	N	N	$50.00
B3: Basic Tier	208,333	814 MB	10	N	N	N	$500.00
S1: Standard Tier	278	1.111 MB	200	Y	Y	Y	$25.00
S2: Standard Tier	4,167	16 MB	200	Y	Y	Y	$250.00
S3: Standard Tier	208,333	814 MB	10	Y	Y	Y	$2,500.00
F1: Free Tier	8,000 (per day)	Unknown	1	Y	Y	Y	$0

Figure 4.3: Different tiers for *IoT Hub*s. IoT Edge enabled means this tier can send/receive messages from IoT Edge devices. Device Management means this tier can use device twin, query, direct method, and jobs [64].

platforms such as Java, C, .NET, Node.js, Python, and iOS frameworks are all supported. This experiment used Java, which needed Microsofts SDKs and Maven. It is recommended to use Maven as a build tool for Java-based IoT client, as Microsoft provides a dependency snippet that can be copied and pasted into the Maven build file (commonly named as pom.xml) which will then import all the appropriate library dependencies (namely jar files) to connect to the *IoT Hub*. Modifying one of the example codes provided by Microsoft was an easy way to start a project and send messages. The default connection string given in the example must be updated to match the devices ID in the *IoT Hub*, if it is not the messages will not send. To simulate telemetry messages, the battery data was first read from the CSV file, converted to strings in JSON format, and sent as MQTT messages to the *IoT Hub* using the libraries provided. A quick way to view whether the messages are being sent or not is to examine the real-time graphs that are shown on the overview page of the *IoT Hub* (Figure 4.4). There will be some latency, but if the messages are sent correctly the graphs will eventually update.

The type of storage account determines what services can be used to store data. StorageV2 allows storage in blobs, files, queues, and tables. StorageV1 is similar to StorageV2 except is used for legacy accounts. BlockBlobStorage accounts are blob only storage tailored

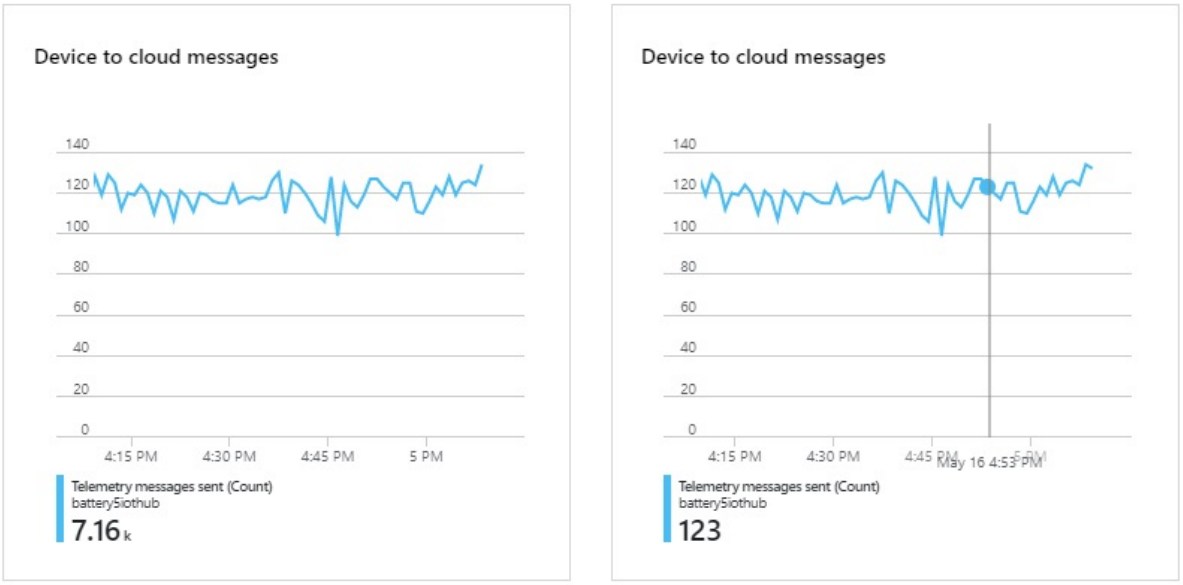

Figure 4.4: Real-time graphs showing incoming messages to the *IoT Hub*.

to blob performance. FileStorage accounts are file only storage tailored to file performance. BlobStorage accounts are blob only storage but for legacy accounts [98]. The replication option determines where duplicated data is stored geographically. Azure creates multiple copies of all data, the further away from the source the more expensive storage gets, and replicated data is updated synchronously.

First storage option is locally redundant storage (LRS). LRS maintains three replicas in the same storage scale unit, which is a collection of storage racks. LRS is the most cost-effective way but it provides the least durability in case of hardware failure or natural disasters. Zone-redundant storage (ZRS) maintains three replicas in the same data center but in different zones. If only the storage scale unit that contains source data malfunctions, replicated data will still be accessible with ZRS. This is the not case with LRS as the source and replicas are in the same storage scale unit. The last two options are geo-redundant storage (GRS) and read-access geo-redundant storage (RA-GRS). Both store three replicas in the same data center and three replicas at a different data center in a region hundreds

of miles away. RA-GRS is the only option that allows read only access to replicated data. Latency could be a problem with RA-GRS, as replicas at different data centers are updated asynchronously unlike updates done in the same data center. Lastly, cold or hot storage can be selected. Hot storage should be used for frequent reads and updates. Use cold storage otherwise.

After creating a storage account with the appropriate settings, see Figure 4.5 for example of this in *Azure portal*, messages sent to the *IoT Hub* can now be stored. If no other services are needed and storage is the only concern, the *IoT Hub*'s Message Routing options can be updated to output data into four locations. They are *Event Hubs*, *Service Bus queue*, *Service Bus topic*, and *Blob storage* [99]. Having a message reroute to an *Event Hub* seems redundant, since the *IoT Hub* is essentially an *Event Hub* with bidirectional communication, but if a user wanted to combine information from an *IoT Hub* and an existing *Event hub* that collects

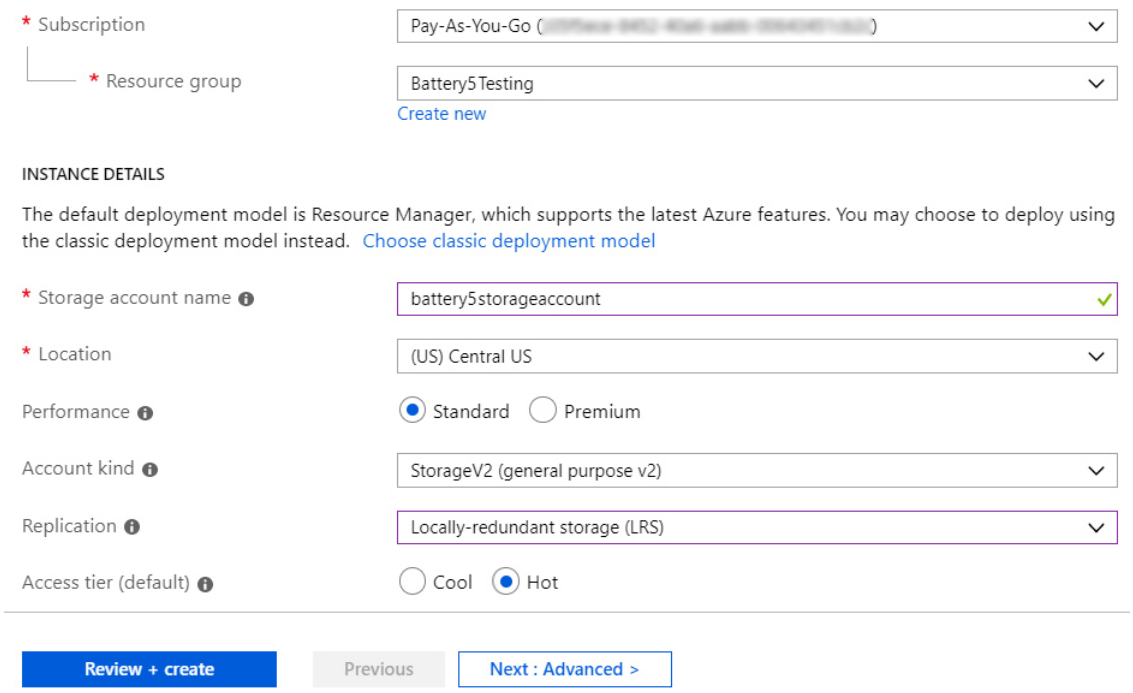

Figure 4.5: Creating a storage account.

data from multiple non IoT sources for data analysis this could be a useful option. A service bus is a message broker, where the *IoT Hub* would be the publisher and any number of Azure services could be a subscriber [65]. A *Service Bus queue* stores messages in a single queue where a subscriber service reads from the queue when ready. Typically, *Service Bus queues* are meant for point to point communication. A *Service Bus topic* is similar to a *Service Bus queue* except that multiple queues are used. This allows multiple subscriber services to read from one source and is more similar to the typical publisher/subscriber model. If the data isn't needed immediately, *Blob storage* can be used for long term storage or for analysis at a later time.

When using the *Azure portal*, certain storage options don't allow a user to view the contents of stored messages. It is recommended to download and install Microsoft Azure Storage Explorer. This program is free and links up with an existing storage account after providing the correct credentials. After installation a user can now access all stored messages in their storage account from their desktop with out having to go through the *Azure portal*.

4.5 Stream Analytics

If more destinations for storage or processing are needed, *Stream Analytics* is another service for routing messages sent to the *IoT Hub* [101]. The basic set up of a *Stream Analytics* job are the inputs, query, and outputs (Figure 4.6). Inputs are differentiated between data streams and reference data [104]. Data streams are unbounded streams of data, like *Event Hubs*, *Blob storage*, or an *IoT Hub*. Reference data is typically static data that is used for correlation and look ups. Currently only *SQL Databases* and *Blob Storage* are offered as reference data inputs. The query language used by *Stream Analytics* is similar to SQL, and considered a subset of Transact-SQL (also known as T-SQL) [46]. Queries can determine

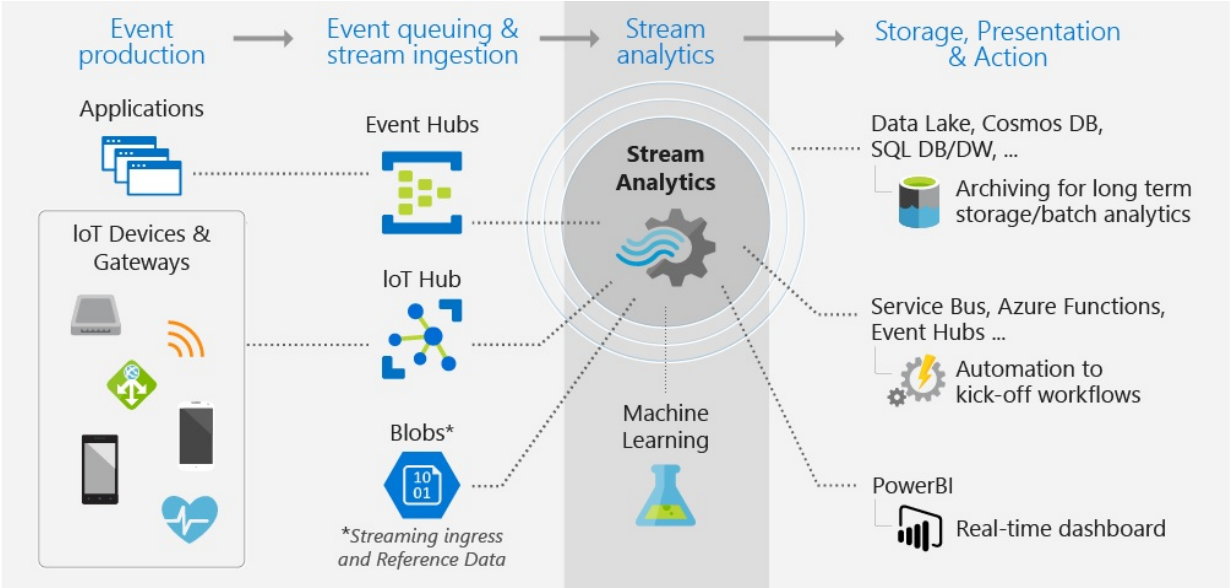

Figure 4.6: Overview of inputs and outputs for *Stream Analytics*. [101]

or limit what gets stored, and can be combined with windowing to perform analytics over designated periods of time. Output is where the data gets stored after being processed by the user defined query [105]. Options for output are *Event Hub*, *SQL Database*, *Blob Storage*, *Table Storage*, *Service Bus topic*, *Service Bus queue*, *Cosmos DB*, *Power BI*, and *Data Lake Storage Gen1*. *Azure Functions* is also an option for output, but these details will be discussed later. Unlike the *IoT Hub*, *Stream Analytics* jobs must be started to be used. A start button is visible after creating a *Stream Analytics* resource, click this to start a job after all inputs, queries, and outputs are added. It may take a few moments for the job to start, but it will inform the user when it has started. A user is only charged when a job is running, so remember to stop the job when it is not in use.

When adding inputs or outputs, an alias is used in a *Stream Analytics* query to refer to an actual resource. Multiple inputs and outputs can be used for a single *Stream Analytics* job, reducing the amount of services required. For the experiment, a *Stream Analytics* job was created that used a *IoT Hub* resource as input and a *Table Storage* resource as output.

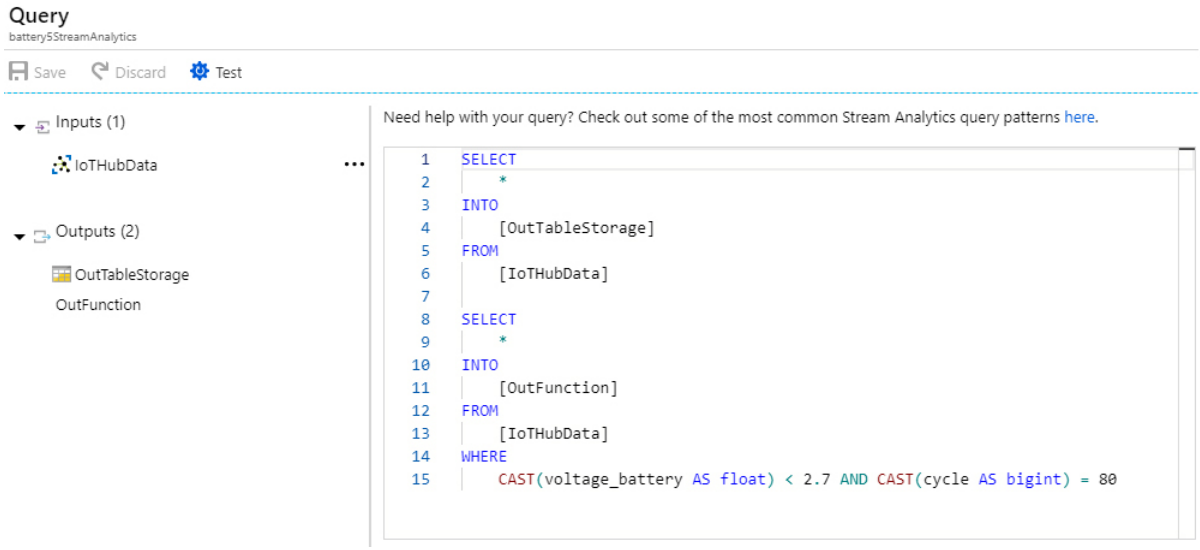

Figure 4.7: *Stream Analytics* query with multiple parts. First selection takes all data sent to an *IoT Hub* aliased as IoTHubData and outputs to *Table Storage* aliased as OutTableStorage. The second selection reads data from the same *IoT Hub*, except when the parameters of the where statement are true it executes an *Azure Function*.

See Figure 4.7 for the example. Several other storage options could have been selected, but *Table Storage* provides all the functionality required. Tables are a NoSQL data store that can store lots of data, can scale with the data, can be queried easily and quickly, and they are viable input and output sources for *Azure Functions* [102, 103]. When adding a table as output for a *Stream Analytics* job, it will ask for a Partition and Row key as field values which form the primary key for each row entry in the table. When combined they must be unique, if not it will over write the existing data with the matching primary key. Both keys must reference a named datatype in the streaming data. If not *Stream Analytics* will display a yellow warning symbol next to the output saying it is degraded. During testing no messages were updated when a *Stream Analytics* job became degraded.

Azure does offer query testing and some syntax error detection which helps save time and potential head aches as starting up a *Stream Analytics* job can be cumbersome [68]. Files to upload must be in JSON, CSV, or Avro format. Running the test after uploading, if done

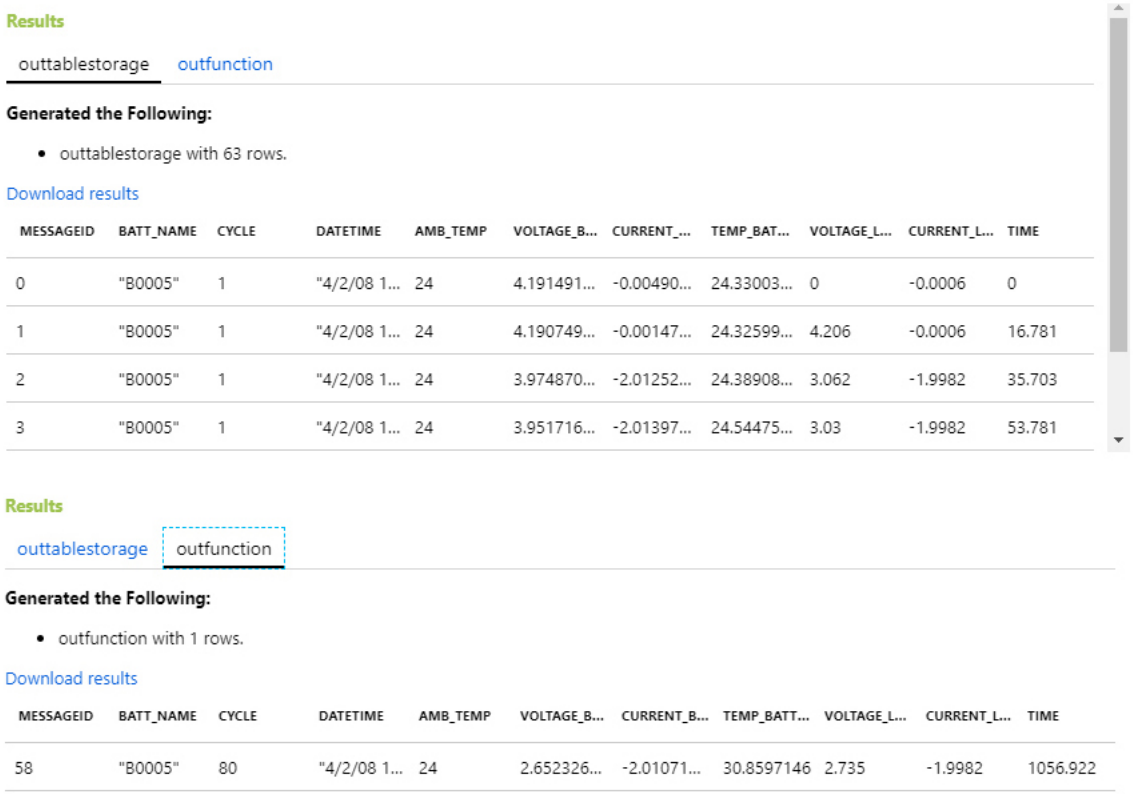

Figure 4.8: Results of testing query in Figure 4.7.

correctly, will give results to what is expected to happen during execution (Figure 4.8). The testing that is offered is helpful, but it is not fool proof. Queries can still be syntactically correct but produce no output. Azure does not provide any debugging tools to understand why something is not working. It is also unclear whether testing works with windowing, as windowing reads messages based on the time they were received and whether they reside inside a designated time slot.

4.6 Azure Functions

Once the data is stored, an *Azure Function* can be triggered to read and perform some computations on the data. The main parts of an *Azure Function* are the trigger, bindings,

and code implementation. The trigger causes a function to execute, and a function can only have one trigger [102]. Previously it was mentioned that *Azure Functions* can be used as output for a *Stream Analytics* job. The output is not a data stream, but an HTTPTrigger that is sent when a certain condition is met to execute the function. Bindings can be distinguished by input vs output. They are not required, but they provide an easier way to connect to other data sources for input or output in the function. See Figure 4.9 for a list of some of the triggers and bindings offered.

Azure Functions can be written in C#, JavaScript, Java, and Python [67]. All languages listed can be written in an integrated development environment (IDE) on a desktop and then uploaded to Azure. An IDE is provided in *Azure portal* through the browser, but it only supports C# scripts and JavaScript. Each option has its advantages and disadvantages. Using a desktop IDE gives the developer the choice to use a setup and programming language they are more familiar with. A desktop IDE also offers debugging and syntax error detection. Developing in the portal offers only two programming languages, and no debugging or syntax error detection. Mistakes are only found when the code is executed. A downside to using a desktop IDE is having to set up the programming environment to be able to connect to Azure. This means having to go through Microsoft's online documentation which can be

Type	Trigger	Input	Output
Blob Storage	X	X	X
CosmosDB	X	X	X
Table Storage		X	X
Queue Storage	X		X
HTTP & Webhooks	X		X
Service Bus	X		X
Event Hubs	X		X
Timer	X		

Figure 4.9: *Azure Functions* triggers, inputs, and output options [102]. Other services are available but were not included as they do not relate to IoT.

difficult to follow. Desktop IDE's also require a user to upload their functions to Azure, where anytime a function is updated it must be re-uploaded. The portal avoids both of these issues.

To create a function in the portal, use the *Function App* service to create a *Function App* resource. The resource creation stage is where the programming language is selected. All functions that reside in this resource will use that language. For the experiment, the .NET framework was chosen to create a C# script because it did not require much code, many of the better examples online were in C#, and ease of environment setup. The rest of the setup described is based on the user selecting the .NET framework in the portal to develop a function. The other programming language options were not tested and will not be discussed in great detail. Once the resource is created and opened in a new blade, a function can be made by clicking the plus symbol. After selecting the trigger to be used and providing a name, Azure will generate a C# script template, and a function.json file. For script files, a function.json file is required to list the trigger, and all input and output bindings [53, 102]. See Figure 4.10 for an example of the function.json file.

The process of updating the json file and connecting to sources outside the function are the most difficult aspects when creating an *Azure Function*. The idea was very simple for the experiment: have a function read data from a table, perform some calculations on the data, and then output the new information to another table. One would think having the storage and processing all done in Azure would be a simple process, but sadly it is not. Microsoft provides a lot of documentation and examples online for how to add input and output bindings, but the amount of information and lack of explanations for each example given make it more difficult than it needs to be.

To update the bindings, select the *integrate* option underneath the selected function. A new blade will open up which allows a user to update the trigger and add any bindings.

function.json

```
{
  "bindings": [
    {
      "authLevel": "function",
      "name": "req",
      "type": "httpTrigger",
      "direction": "in",
      "methods": [
        "get",
        "post"
      ]
    },
    {
      "type": "table",
      "name": "inputTable",
      "tableName": "battery5TableStorage",
      "take": 50,
      "connection": "AzureWebJobsStorage",
      "direction": "in"
    },
    {
      "type": "table",
      "name": "outputTable",
      "tableName": "battery5CoulombTable",
      "connection": "AzureWebJobsStorage",
      "direction": "out"
    }
  ],
  "disabled": true
}
```

Figure 4.10: A functions trigger and bindings json file.

When using table storage, it needs the connection string to the appropriate storage account, the name of the table where the data is stored, and a parameter name. There are other fields to fill out but they are optional and not required for the experiment. The connection string can be updated in the storage account connection field option. Clicking the *new* option next to the field reveals all storage accounts associated with the current subscription. Selecting the correct storage account will automatically import the connection string. When inputting the table, only the table name is required. Unlike *Blob storage*, table storage does not allow sub containers so all tables are stored in the same location under one storage account [103].

The parameter name is the name used in the function when referencing the actual table. For example, the experiment's table name was battery5TableStorage and the parameter name was inputTable. Any time a read operation was performed on the battery5TableStorage table in the function, the inputTable parameter name would be used. The documentation online on how to perform this basic setup is lacking. While intuitive in hindsight, it was initially very confusing. After updating the bindings through the portal, the existing json file should update automatically to reflect the changes made.

Once the bindings are updated, modifications can be made to the function code. The parameter names used in the json file must be used as actual parameter names for the function. The datatype for these parameters depends on which function runtime is being used. The two versions are 1.x, for legacy runtimes, and 2.x, for newer runtimes [66]. If using 1.x, the datatype used is called IQueryable. If using 2.x, the datatype used is called CloudTable. Developing functions in the portal automatically uses version 2.x, so each parameter representing a table should be a CloudTable datatype. Continuing the example from above, the full function parameter name would be CloudTable inputTable.

After the function parameters are declared, it would be nice if the process was over and a user could simply use the CloudTable parameters directly. But there is still more setup required. To get information from the table, it can be queried using TableQuery and TableQuerySegment datatypes [54]. A query is provided to a TableQuery, which in turn is passed to a TableQuerySegment to get information from the table. After the results are returned from the query, one final step remains. A class must be declared whose instance variables match the names of the columns in the table to be read. The user may select which columns to use as instance variables for the class, but the partition key and row key are included by default. Take note of the datatypes of each column, the datatype of each instance variable must match the datatype of the corresponding column. If unsure of the

data types, use the Microsoft Azure Storage Explorer to view specific details about each column. Once the class is created, data can finally be processed and analyzed by iterating over each row in the results sequentially. Each rows data is stored in an object of the classes instance variables allowing for easy access and modification of the data.

To store data to a table from a function, the output binding must be added to the json file similar to how input bindings were added. The chosen parameter name must be added to the function parameter list as well. For example, if an output table was declared in the bindings called outputTable, the full function parameter name would be CloudTable outputTable. No querying is necessary to write to the output table, so the TableQuery and TableQuerySegment datatypes are not needed. Instead, a TableOperation datatype is used [54]. An object must be created which stores any relevant data for a row to be written. This object is passed to the TableOperation which then writes the data to storage. Online documentation shows other ways to read and write from table storage, but the examples are limited and don't provide much detail if an error occurs. See Figures 4.11 and 4.12 for function used.

```
75 //Class to store input data from table storage.
76 public class ReadEntity : TableEntity
77 {
78     public string Batt_name { get; set;}
79     public string dateTime { get; set;}
80     public long MessageID { get; set;}
81     public long cycle { get; set;}
82     public double time { get; set;}
83     public double voltage_battery { get; set;}
84     public double voltage_load { get; set;}
85     public double current_battery { get; set;}
86     public double current_load{ get; set;}
87     public double temp_battery { get; set;}
88     //Variable not part of input table.
89     //Added as new compute variable to help prediction.
90     public double capacity { get; set; }
91 }
92
93 //Class to sort all inputs by messageID.
94 class TableSorter : IComparer<ReadEntity>
95 {
96     public int Compare(ReadEntity x, ReadEntity y)
97     {
98         return (int)(x.MessageID - y.MessageID);
99     }
100 }
```

Figure 4.11: Classes used to help *Azure Function* read and write to table storage. See Figure 4.12 for main method of *Azure Function*.

```
#r "Newtonsoft.Json"
#r "Microsoft.WindowsAzure.Storage"

using Microsoft.WindowsAzure.Storage.Table;
using Microsoft.WindowsAzure.Storage;
using System.Threading.Tasks;
using Microsoft.Extensions.Logging;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Primitives;
using Newtonsoft.Json;
using System.Net;

public static async void Run(HttpRequest req, CloudTable inputTable, CloudTable outputTable, ILogger log)
{
    //TableQuery used to store query.
    TableQuery<ReadEntity> query = new TableQuery<ReadEntity>().
        Where(TableQuery.GenerateFilterCondition("Batt_name", QueryComparisons.Equal, "B0005"));
    TableContinuationToken token = null;
    List<ReadEntity> allEntities = new List<ReadEntity>();

    do
    {
        //TableQuerySegment uses TableQuery to search inputTable.
        TableQuerySegment<ReadEntity> resultSegment =
            await inputTable.ExecuteQuerySegmentedAsync(query, token);
        //Token used to continue searching large tables.
        //If not used, TableQuerySegment will only return 1000 rows.
        token = resultSegment.ContinuationToken;

        //InputTable may not be sorted.
        //Must store in list to sort before performing computations.
        foreach (ReadEntity entity in resultSegment.Results)
        {
            allEntities.Add(entity);
        }
    } while (token != null);

    allEntities.Sort(new TableSorter());

    double coulomb = 0;
    bool hasWritten = false;
    ReadEntity prev,cur;

    for(int i = 1; i < allEntities.Count; i++)
    {
        prev = allEntities[i-1];
        cur = allEntities[i];
            if(prev.cycle == cur.cycle)
            {
                if(prev.voltage_battery < 2.7)
                {
                    //Add coulomb data to current object and write to table.
                    prev.coulomb = coulomb/3600;
                    TableOperation insertOperation = TableOperation.Insert(prev);
                    await outputTable.ExecuteAsync(insertOperation);
                    hasWritten = true;
                }
                else if(!hasWritten)
                {
                    //Compute coulomb.
                    coulomb += (Convert.ToDouble(cur.time) - Convert.ToDouble(prev.time)) *
                        Convert.ToDouble(cur.current_battery);
                }
            }
            else
            {
                //New cycle has started. Reset Values.
                coulomb = 0;
                hasWritten = false;
            }
    }
    log.LogInformation("Ending process.");
}
```

Figure 4.12: *Azure Function* used to read and write to table storage. See Figure 4.11 for classes required to complete function.

5.2 Possible Solutions

5.2.1 Distributed Model

The first solution uses open source software to create a distributed IoT system than can run on cloud virtual machines. It is composed of several parts of the Apache Software Foundation, which provides many open source software solutions that can be used and combined in different ways. The proposed combination uses Kafka [24] for ingestion, Druid [23, 132] for processing and querying, Zookeeper [26] for cluster management, and Hadoop Distributed File System (HDFS) [25] for data storage.

Apache Kafka is a distributed streaming platform which follows the publish/subscribe model, where IoT devices can publish their telemetry data and subscribers can read from it. Kafka is well suited for communications inside a shared network, but if Kafka is running in a cloud it is not immediately usable for ingesting IoT data without some modifications [115]. A major concern is devices need to be able to directly connect to Kafka. Because the IoT devices exist outside the cloud and Kafka is in the cloud, Kafka would need to be exposed to the public internet to be visible to the devices. To keep it encapsulated in the cloud, a server running an MQTT broker is required to act as a secure bridge for telemetry data between devices and Kafka. The MQTT Broker can be custom built for the project, or it can be open sourced or purchased such as the different options offered from HiveMQ. If Kafka struggles with the high velocity data, the server running the MQTT Broker can perform batch uploads of multiple messages to increase throughput.

The subscriber to Kafka would be Druid, which is an online analytical processing (OLAP) distributed data store. It is designed for real-time analytics, is column orientated, and is optimized for low query latency on large data sets. IoT data can vary greatly between sensors, as each sensor is tailored to a unique scenario or environment. A common trait with

IoT data is it is time series based, meaning all IoT data will have a timestamp related to when the information was collected. Time series databases (TSDB) such as InfluxDB [37], TimescaleDB [128], and OpenTSDB [116] were designed to store and query large volumes of time series data. As stated previously parts of this design can be swapped out, such as replacing Druid and HDFS with OpenTSDB and HBase. Druid was selected because it pairs well with Kafka, and it performs well not only on time series data but multi field data. Currently Airbnb, Alibaba, Cisco, eBay, Hulu, Lyft, Metamarkets, Netflix, Paypal, Twitter, Walmart and many more use Druid to monitor large ingestion streams and perform analytics. The highest ingestion rates come from Metamarkets, which claims to ingest 200 billion events per day which is about 2.4 million events/s [22].

Druid's architecture is based around having a cluster of nodes where each node performs a unique operation based on its type. The four types of nodes used are real-time, historical, broker, and coordinator. Because it is a distributed system, all nodes share metadata information with Zookeeper, a coordination system for distributed file systems. Real-time nodes are responsible for ingesting and querying event streams. Kafka can be used as a single point of entry for telemetry data and act as a message bus which real-time nodes can read from. They are also responsible for writing the data to deep storage, where deep storage can be any compatible storage service with Druid. For this solution HDFS was chosen as it is part of the Apache Framework and is widely used and tested.

Historical nodes load and serve the data created by real-time nodes. All data that is stored is immutable. Immutable data offers efficient replication, read consistency, and parallelization across all historical nodes. Broker nodes read the metadata stored in Zookeeper to determine which data segments are queryable and route the queries to the correct real-time and historical nodes to be processed. Coordinator nodes monitor the data in historical nodes. They tell historical nodes when to load, remove, and replicate data based on a set

of rules loaded in to each coordinator node. They also monitor load balancing to distribute data for better query performance.

Being distributed makes the system horizontally scalable, making it well suited for ingestion, storing, and querying of high velocity and high volume data. Using open source software has many benefits as well. It reduces costs as open source is typically free, it can have a robust support network of active users, and it gives more flexibility as the software can be tailored to unique scenarios. Running this solution in the cloud alleviates having to manage all hardware, which includes networking, operating system updates, and fixing, adding, or replacing machines.

A few downsides are costs and expertise. The costs of running multiple virtual machines can add up quickly. Knowing how much data will be processed can help determine what type of virtual machines will be needed, but sometimes this data isn't readily available or it may change as the project continues making estimation difficult. To further complicate things, understanding Quality of Service (QoS), Service Level Agreements (SLA), and the pricing structures for different virtual machines can be very confusing as they vary between cloud providers [18]. For example, virtual machines charge differently for compute and storage services per virtual machine. A Kafka cluster ingesting from 1 MB/s to 32 MB/s can cost between a few hundred dollars to over $1500 per month just for storage for that cluster using *AWS EC2* [2]. Using many of the tools selected will require a solid knowledge base, as getting everything to work together as a cohesive unit will require skill and time. For some developers this may not be a feasible solution as the knowledge required and costs may be too high.

5.2.2 Cost Efficient Model

An alternative solution is to combine many of the existing services from a cloud provider with custom software to create a design that is cost efficient, scalable, and queryable at the expense of added latency. The design consists of edge devices, cloud provided SDK's, blob storage, storage queues, serverless functions, and a single virtual machine for processing queries. Edge devices, a stand alone server, will be responsible for collecting IoT telemetry data from devices, compressing batches of data based on a set time frame, and performing batch uploads directly to blob storage using cloud provided SDK's. Since it is time series data, each entry in the batch should be sorted by time making it easier to keep track of which messages belong to which batch. The batch is stored as a blob in compressed format, which helps reduce bandwidth and storage needed in the cloud. For example, a CSV file with 200,000 different entries with 13 fields was created spanning 10 seconds to mimic device telemetry. With out any compression the file size is about 36MB, when using basic windows compression it is only 3.5MB. The batch is only decompressed when being scanned by the serverless function or when returning a query, both which are discussed later. When the batch of data has been successfully uploaded as a blob, the blob URL can be obtained and uploaded to a storage queue in the cloud.

The storage queues sole responsibility is to alert a serverless function a new blob has been created. Combining serverless functions and queue storage take advantage of the pay-as-you-go billing that many cloud providers offer, as serverless functions are only charged when they are running and they will only run when the queue triggers them. The queue creates an alert which triggers the function to execute, which reads the blob URL at the head of the queue. The function uses the blob URL to download the actual blob, decompress it, and the read the telemetry data creating a minimum and maximum range for each field. It is also possible to calculate ranges for other fields that are not part of the initial data set, such as

key performance indicators (KPI) like power levels or total harmonic distortion (THD). If certain fields are used for error detection, a minimum or maximum threshold can be applied when scanning. If the either threshold is met, the blob URL and a warning message can be added to a warnings queue. The developer is then alerted to when a message is added and they can inspect further.

After reading the entire blob, a metadata tag is created which contains the device id, current year, current month, current day, start time of blob, end time of blob, all minimum and maximum values for each field and KPI, and the blob URL. To reduce cloud computing costs the edge device can perform many of the calculations of each batch of data. A downside is this will put a greater strain on the edge device, requiring a more powerful and expensive server to handle the increased load. It is up to the developer which avenue to choose, but having each part of the system perform one job allows for easier implementation and debugging.

When the metadata for a blob is complete, it is sent to a single virtual machine which is responsible for the storing metadata tags, and answering queries. The storage structure used takes ideas from Colmenares *et al.* [16] MultiDimensional DataStore (MDDS), which focused on creating a single node that can handle ingestion, indexing, storage, and querying of massive amounts of data. To store and index data, a two-level indexing structure of R*-trees with kd-trees was used to increase query performance. A R*-tree is a variation of the R-tree, which creates minimum bounding rectangles that are useful in storing multidimensional data. The difference between the two is the R*-tree will "self balance" itself, which involves a complicated process of removal and reinsertion of data to reduce overlap and increase query performance. A kd-tree is a data structure similar to a binary search tree, only at each node depth a different dimension is used to sort the data. Kd-trees are useful for storing multidimensional data, as well as performing range queries, point queries, and finding a

nearest neighbor. R*-trees and kd-trees can suffer from the curse of dimensionality, which is when the number of dimensions exceeds a certain threshold causing poor query performance. High dimensions usually means 10 to 16, but can be as a low as 5 or 6 in some cases [13].

The virtual machine will have a varying level indexing structure, where the number of levels is determined by how much metadata is going to be stored. Because each device will be creating different telemetry, the first level will always be sorted by device id to keep the device data separate. Since IoT data is time series based, the next levels of the structure will focus on year, month, day, and hour. Year, month, and day are the varying part of the structure, as the amount of metadata to be stored will determine which values to use. If the number of metadata tags to store is low, year, month, and day can all be used. If the number is very high, then the developer needs to determine which levels to include as the memory of the virtual machine is finite. If one of these is not included in the indexing structure it will still be used when storing the indexing structure to blob storage. For example, if month and day were used and year was not, every year would have its own indexing structure stored as a blob. When a query is looking for data spanning years, each individual indexing structure relating to a year in the range of the query is loaded from blob storage in to the virtual machine for scanning.

The second lowest level will always be hour, where each hour level is broken up in to 24 separate buckets for each hour of the day. Each bucket points to an R*-tree where all metadata tags are stored for that time period. Breaking the indexing structure in to a tree where each level focuses on a different unit of time helps reduce the amount of data being stored in each R*-tree. It can also be advantageous to know what fields will have very little variance, as these fields do not need to be indexed to help reduce the curse of dimensionality. If the number of metadata tags created per hour is low, a list can be used instead of an R*-tree to store the metadata tags.

When a query is entered, the virtual machine will work its way through the levels and store all metadata tags returned from the R*-trees. When all tags are retrieved, the blobs are downloaded to the virtual machine to display the results to the user. If more processing is required, say only specific values are needed and not entire chunks of data, the virtual machine can parse out the relevant information from each blob. If the number of devices and time frame used are constrained limiting the search scope, the amount of blobs returned will be manageable with most latency coming from downloading and decompressing the blobs. If a query searches many devices with larger time frames this can drastically increase the amount of blobs returned. After all metadata tags are retrieved, the virtual machine can perform a quick estimation based on the number of blobs and the size of each blob for how long it will take to download and further process. The user can then decide whether to proceed or not based on the result, or they can further trim the results by modifying their original query. This design does involve re-computing derived fields such as KPIs. If the query limits the scope, the amount of blobs returned should be manageable for most computers to be able to download and process with acceptable latency.

This proposal offers a more affordable solution to using a distributed system. Costs are saved by using blob storage, bypassing cloud gateways with cloud provided SDKs, taking advantage of serverless functions with pay-as-you-go billing, and running a single virtual machine to perform queries. The solution is not perfect, as every design will have trade offs. Latency is increased, making it difficult to use in time critical situations. Using existing cloud services can be very restraining and at times frustrating as documentation can be lacking and confusing at the same time. Fine tuning parameters such as time frame size, error detection thresholds, and which fields to index in the R*-tree for optimal query performance are trial and error based. If many devices are required, each device requires its own indexing structure limiting the number of devices the system can handle. The goal was to provide a way to

store, index, and query IoT high volume and velocity data efficiently and in a cost-effective manner in the cloud. If costs are no issue, using a distributed system will provide good performance. Otherwise the alternative is to combine many of the existing technologies into a custom solution.

6 AWS vs Azure vs GCP

According to Harvey et al. [34], the top three cloud providers ordered from first to third are Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP). AWS's maturity, continued ingenuity, broad distribution of data centers, and extensive set of tools are reasons why developers and companies continue to make it the number one cloud platform. It is not surprising then that these are the top three cloud-based IoT platforms as well. According to an IoT developer Survey from the Eclipse Foundation [27], 34% of respondents use AWS, 23% use Azure, and 20% use GCP. As each platform competes for more market share, the services provided become more similar. The similarities and differences are made clear as the remainder of this chapter will compare AWS, Azure, and GCP based on IoT related services. Pricing is not directly addressed even though it is a major factor when selecting a cloud provider. Each provider uses different metrics for compute resources, services can have different tiers with varying pricing, and prices are constantly changing as new services are added or updated making it difficult to capture the total costs.

6.1 IoT Related Services

IoT related services are specific to IoT use cases. These services include IoT cloud gateways, acceptable message formats, support for edge computing, and provided software development kits (SDKs). Looking at Figure 6.1, the similarities are evident as each category has a similar service for each provider. The main differences are gateway throughput limits, possible message formats, and provided device SDKs. Each provider offers different levels of throughput, where limitations and scaling are unique to each gateway. Azure accepts MQTT, HTTP, and AMQP, where AWS and GCP only accept MQTT, and HTTP. In terms

IoT Related Services					
Cloud Platform	Gateway (Bi-Directional)	Gateway (Ingestion Only)	Message Format	Edge Computing	Device SDKs
Amazon AWS	AWS IoT Core	Kinesis Data Firehose, Kinesis Data Streams	MQTT, HTTP	AWS Greengrass	Embedded C, C++, Java, JavaScript, Python, Android, Arduino Yun, iOS
Microsoft Azure	IoT Hub	Event Hubs	MQTT, HTTP, AMQP	IoT Edge	C, Java, Python, Node.js, .NET, iOS
Google Cloud Platform	IoT Core	Cloud Pub/Sub	MQTT, HTTP	Edge TPU	Embedded C

Figure 6.1: Comparison of cloud-based IoT services between AWS, Azure, and GCP.

of device SDKs, C or embedded C SDKs are offered by all platforms. AWS and Azure offer Java, Python, and iOS SDKs. Azure is the only provider for Node.js and .NET SDKs. AWS is the only platform to offer Arduino Yun, a popular microcontroller, and Android device SDKs.

While not listed, AWS and Azure offer multiple other services tailored to IoT scenarios. *AWS freeRTOS* [3] is an open source operating system specifically for microcontrollers, allowing easier connections to *AWS IoT Core* and *AWS IoT Greengrass*. *AWS IoT 1-Click* [4] allows for simple devices to trigger AWS Lambda functions. This makes it easier to alert staff, and monitor asset tracking. *Azure Time Series Insights* [108] is a managed service that is meant to store, visualize, and query large amounts of time series data, combining several services into one. *Azure IoT Central* [107] provides a fully managed interface for managing simple IoT projects.

6.2 IoT Storage Services

Each provider offers a multitude of storage options which can serve to store IoT data. Figure 6.2 lists the different types of Object storage, SQL databases, NoSQL databases, Data Warehouses, and Data Lake options for each provider. There is not much that differentiates

Cloud Storage Services					
Cloud Platform	**Object**	**NoSQL Database**	**SQL Database**	**Data Warehouse**	**Data Lake**
Amazon AWS	Amazon Simple Storage Service (S3)	Amazon DynamoDB, Amazon SimpleDB, Amazon EMR (Apache Hbase)	Amazon RDS (Relational Database Service)	Amazon Redshift	AWS Lake Formation, AWS Data Lake Solutions
Microsoft Azure	Blob Storage	CosmosDB, Table, HDInsight (Apache Hbase)	SQL Database, PostgreSQL	SQL Data Warehouse	Data Lake Storage
Google Cloud Platform	Cloud Storage	Cloud BigTable, Cloud Firestore	Cloud SQL, Cloud Spanner	BigQuery	Cloud Storage, BigQuery

Figure 6.2: Comparison of cloud-based storage services between AWS, Azure, and GCP.

the three platforms as many of the services are replicated to provide similar functionality. Each version of object storage offers warm, cold, and archival storage. SQL databases are available for structured data. Object, NoSQL databases, Data Warehouses, and Data Lakes can be used for big data storage.

One category that requires further explanation is Data Lake Storage, as the services offered vary more compared to other storage categories. *Azure Data Lake Storage* has two versions, Gen1 [106] and Gen2 [74]. *ADLS Gen1* is an Apache Hadoop file system, running on an *HDInsight* cluster, for big data storage. *ADLS Gen2* combines several features from Gen1 but uses Azure *Blob storage* instead of an *HDInsight* cluster, reducing overall costs while still offering good performance.

AWS Lake Formation [5] is a service that can help set up a data lake using *Amazon S3* for storage. AWS also offers many guides and solutions for how to create a data lake, where ingestion, storage, and analytics are all integrated into one unit and the design can be customized to the problem [7]. One example uses a provided template file which is read by *AWS CloudFormation* service to create all services used by the data lake [6]. The process creates a solution using *AWS Lambda*, *Amazon Elasticsearch*, *Amazon Cognito*, *AWS Glue*, *Amazon Athena*, *Amazon S3* and *Amazon DynamoDB*. An odd thing about AWS naming is if a service starts with "Amazon" it means it is a stand alone service, where if it starts with "AWS" it is a utility service that typically uses other services.

GCP provides a design architecture for implementing data lake storage using *Cloud Storage* service [31]. Several techniques are discussed for ingestion and analysis, but from our research no services are provided to automatically create a data lake such as AWS and Azure. Another possibility is to use *BigQuery*, which will act as both Azure *Data Lake Storage* and *Data Lake Analytics*, reducing the amount of services to monitor [32].

6.3 Streaming, Analytics, and Visualization Services

The remaining services do not fit in the IoT specific or storage related categories from the previous sections, which is why they are compared in this final section. Real time stream processing, analytics, serverless computing, machine learning, and visualization services are offered by all three platforms as seen in Figure 6.3. Similar to gateways, real time streaming will vary in terms of throughput limits per platform. All three provide their own unique solutions for analytics, such as *AWS IoT Anaytics*, *Azure Data Lake Analytics*, and *BigQuery*. *Amazon EMR*, *Azure HDInsight*, and GCP *Cloud Dataproc* repackages the Apache Framework, such as Spark, Hive, HBase, and Kafka, to more easily integrate with many of their other services such as gateways, storage, and visualization for more analytical options.

Streaming / Analytics / Visualization					
Cloud Platform	**Real Time Stream Processing**	**Analytics**	**Serverless Compute**	**Machine Learning Services**	**Visualization**
Amazon AWS	Amazon Kinesis Data Analytics	AWS IoT Analytics, Amazon Athena, Amazon EMR(Apache Framework Tools)	AWS Lambda	SageMaker	Amazon Quicksight
Microsoft Azure	Stream Analytics	Data Lake Analytics, HDInsight (Apache Framework Tools)	Functions	Machine Learning Services	Power BI Desktop, Power BI Service
Google Cloud Platform	Cloud Dataflow	BigQuery, Cloud Dataproc (Apache Framework Tools)	Cloud Functions	Cloud Machine Learning Engine	Cloud Datalab, Data Studio

Figure 6.3: Comparison of cloud-based streaming, analytics, and visualization services between AWS, Azure, and GCP.

McGrath et al [44] performed two tests on all three serverless functions, *AWS Lambda*, *Azure Functions*, and *GCP Cloud Functions*. The first test focused on concurrency and scaling, with *AWS Lambda* almost scaling linearly and having the highest throughput. *GCP Cloud Functions* performed the worst, with performance appearing to decrease after a certain threshold. *Azure Functions* had good performance but does not exhibit obvious scaling with concurrency. The second test focused on cold start times and expiration behaviors for each function. *AWS Lambda* and *GCP Cloud Functions* had similar performance with minimal latency, and *Azure Functions* performed the worst.

Machine learning has become another hot topic in computer science, and each platform offers different yet similar services. *AWS SageMaker*, *Azure Machine Learning*, and *GCP Cloud Machine Learning Engine* offer multiple options and algorithms for creating a trained model to integrate into their IoT designs. Similar to machine learning and many of the categories before it, visualization services are offered for each platform. Microsoft's *Power BI* may have the upper hand with respect to the other services, as *Power BI* integrates with many of the programs from the Microsoft's Office Suite that many people are familiar with already.

7 Conclusion

Developing an IoT application on a cloud platform requires thoroughly understanding the programming API, configuration, performance, limitation, and design patterns of the tools of the platform. In this thesis, we reviewed the different aspects required to create a cloud-based IoT system, such as device clients, gateways, IoT communication protocols, storage, analysis, and visualization using many of the tools from Azure as examples. Following the requirements and design patterns for a cloud-based IoT system, we provided our experiences using Microsoft Azure to create a custom IoT solution. Our insights and experiences can be used by many as an introduction to the various services Azure provides, or as a starting point to implement their IoT solution.

New services for IoT platforms are emerging every year, with performance and API of existing tools rapidly improving as well. While cloud-based IoT platforms are already scalable, extensible, and comprehensive development environments, it still has many limitations. One limitation discussed focused on ingestion and storage of extremely high velocity and high volume data, where current gateways cannot handle the high traffic without adding great costs to the user. Two solutions were proposed, one implementing a distributed IoT system running on cloud virtual machines, and another custom solution combining existing cloud services, on-premise edge devices, and a unique indexing structure used for querying. The distributed model reduces latency, but costs more. The custom solution is less expensive, but has higher latency. There are many possible solutions that can be implemented, but there will always be a trade-off between latency and costs which need to be factored into design choices.

This thesis also provides a comparison of the current IoT related services from the top

three cloud platforms, AWS, Azure, and GCP. All provide similar services for many of the requirements of implementing an IoT system. For example, each platform offers a bi-directional gateway with similar functionality that varies in terms of throughput and costs. This is a trend in most categories, where services provided are similar with minor nuances. Some platforms offer unique solutions to different problems, such as *Azure Time Series Insights* and *AWS IoT 1-Click*. Each platform also provides unique services tailored to their platform, such as *AWS IoT Analytics*, *Data Lake Analytics*, and *BigQuery*.

There are several areas in this thesis that can benefit from further work and research. In Chapter 4, the IoT system created used only Microsoft Azure. It would be beneficial to replicate the same IoT system in AWS and GCP, explaining the different steps and comparing the overall process done to implement the system. Even though many of the services are similar, implementing and maintaining each service may vary greatly between platforms. If each platform is running the same IoT system using platform specific services, the performance of each IoT system can also be compared to create more distinctions between the providers. Another issue is the IoT system created in Azure was not designed to put a lot of stress on Azure. A more robust system could be designed where performance can be measured across all services used, comparing the results to the performance claims of Azure. This can also be done on AWS and GCP, not only comparing the results against each cloud providers claims, but against each other as well.

Two solutions were proposed in Chapter 5 for handling extremely high velocity and high volume data, but neither were implemented. Detailing the experiences of creating each solution can offer more insights into the many services offered by a platform. Analysis and performance metrics can be compared to show the feasibility, strengths, and weaknesses of each design. Costs can also be compared revealing potential savings and trade-offs with performance.

Lastly, further analysis can be gained when comparing the many different services between the cloud providers in Chapter 6. Costs were not discussed at all, even though this could be the single determining factor between each provider. One possible idea is to create many different IoT scenarios, each having varying ingestion, storage, analytical, and visualization requirements. Using the pricing calculators for each platform, overall costs can be estimated and compared. More information can also be provided between the many services offered, but navigating and reading through each platforms online documentation can be very cumbersome and confusing. Key details are often not discussed or buried in other pages, links to other pages can lead you in circles, and examples are limited or outdated. Beyond these shortcomings, this thesis provides a wide depth of knowledge and information with respect to cloud-based IoT systems that all skill levels can use to gain insights and better understanding of the evolving world of IoT.