**IST-615 Cloud Management**

Team #3

Krister Samuelson

Ninad Bhikaje

Sai Muggulla

Spoorthi Nagareddy

Final Project Report

Due: 4/29/24

Submitted: 4/29/24

Summary

The inspiration for this project comes from the real life experience of one of our members when they worked for an IT department that was struggling to actively track the environmental status of their server room. During a tumultuous period of HVAC work, the department resorted to having personnel manually check thermometers at regular intervals. This resulted in an unnecessary drain on time and morale that we believe could have been mitigated.

Our project will utilize cloud services to track and log the temperature and humidity of a fictional server room as a proof-of-concept solution to these problems. Whenever the temperature or humidity exceeds set thresholds, an alert will be created and sent out via email with details on the current status to notify personnel of the dangerous conditions. The data will be simulated since we don’t have access to a live server room and to allow for straightforward testing. The data will be logged and saved in the cloud permanently for reference and possible long-term trend analysis, although that is currently out of the scope of this project. We will be utilizing Microsoft Azure services for this project. By actively monitoring the status of a server room and utilizing the versatility, power, and scalability of the cloud, this project will serve as a proof of concept solution for organizations struggling to monitor the environmental status of their critical infrastructure.

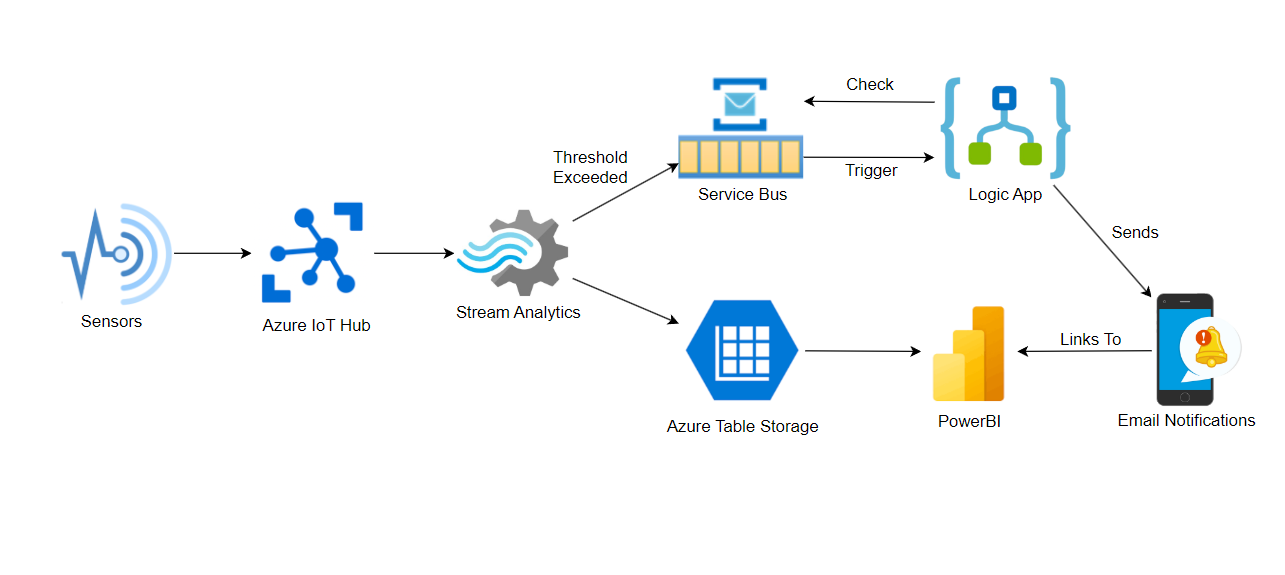
Services Used

1. Azure IoT Hub - The Internet of Things Hub is Azure’s IoT device management application. This service will interface with the temperature and humidity sensors to manage these devices and provide the live data stream to the cloud. The “sensors” will be sending simulated data for testing and proof of concept. Microsoft provides free IoT Hub device simulators which will be configured as the temperature and humidity sensors (<https://apps.microsoft.com/detail/9pcqwq17mxm0?hl=en-US&gl=US>).
2. Azure Stream Analytics -Stream Analytics is Azure’s real-time analytics service. A Stream Analytics job will be created to read the incoming real-time data from the IoT Hub and trigger a Logic Apps workflow when the data exceeds setpoints (<https://azure.microsoft.com/en-us/products/stream-analytics>).
3. Azure Logic Apps - Logic Apps is Azure’s low-code automated workflow service. This service will be triggered by Stream Analytics to run a workflow that will send out the alert to relevant personnel through an email. The alert will include basic information such as the date, time, and current status of the server room from Stream Analytics (<https://learn.microsoft.com/en-us/azure/logic-apps/logic-apps-overview>).
4. Azure Storage - Storage is Azure’s cloud storage service. Data from Stream Analytics will be logged and stored in a table for easy access and possible future trend analysis (<https://learn.microsoft.com/en-us/azure/storage/common/storage-introduction>).
5. Azure Service Bus - The Service Bus functions as a message queuing service (<https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-messaging-overview>)
6. PowerBI - Azure includes access to its PowerBI embedded service. However, due to the high cost of the Azure version, we will be utilizing the desktop version instead.

Goals

1. **Develop a Proof-of-Concept Solution:** Create a system using Microsoft Azure to actively monitor and record the temperature and humidity levels in a server room.
2. **Address Manual Monitoring Issues**: Implement an automated system to eliminate the need for manual temperature checks, and set up alerts for when predefined thresholds are exceeded.
3. **Utilize Cloud Services:** Utilize Azure IoT Hub for managing devices, Azure Stream Analytics for real-time data analysis, Azure Logic Apps for automated workflows, and Azure Storage for data logging. Showcase the adaptability, power, and scalability of cloud services for monitoring critical infrastructure.
4. **Use Simulated Data for Testing:** Use simulated data for testing purposes, as live access to the server room is not available. This allows for straightforward testing and demonstration of the proof of concept. Ensure that data is stored in the cloud for long-term reference and potential trend analysis.
5. **Ensure Alerting Mechanism:** Implement a system that sends alerts, such as emails, to notify personnel when the temperature and humidity levels reach dangerous levels.
6. Demonstrate how the solution can help organizations struggling to monitor their critical infrastructure's environmental status, highlighting the advantages of cloud-based monitoring solutions.

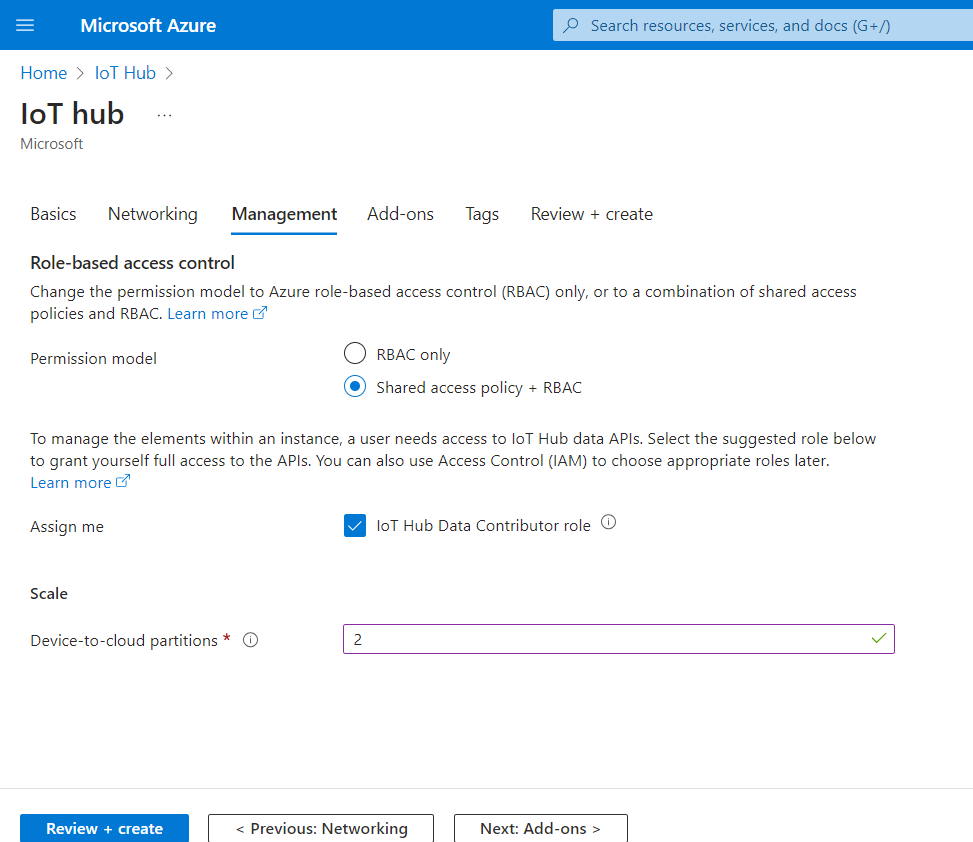
Diagram:



*Figure 1: Architecture Diagram for Proposed Cloud Solution*

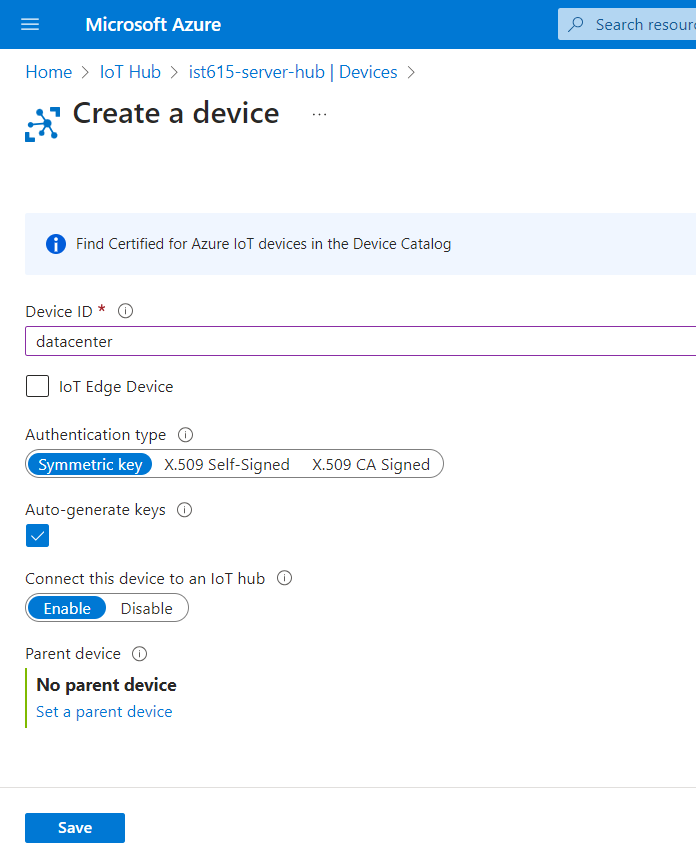
Azure IoT Hub Configuration

We began with configuring the Azure IoT Hub. All values were set to default except for the management tab, which was set as shown in Figure 2 below. We had to ensure that the Data Contributor role was checked so the owner could connect the endpoint devices. Also, the permission model had to be set to “Shared access policy + RBAC” in order for it to connect to the Azure Data Stream. Partitions were set to 2 as this is a small proof of concept project, and limiting partitions will simplify management and data streams.



*Figure 2: Azure IoT Hub Management Setup*

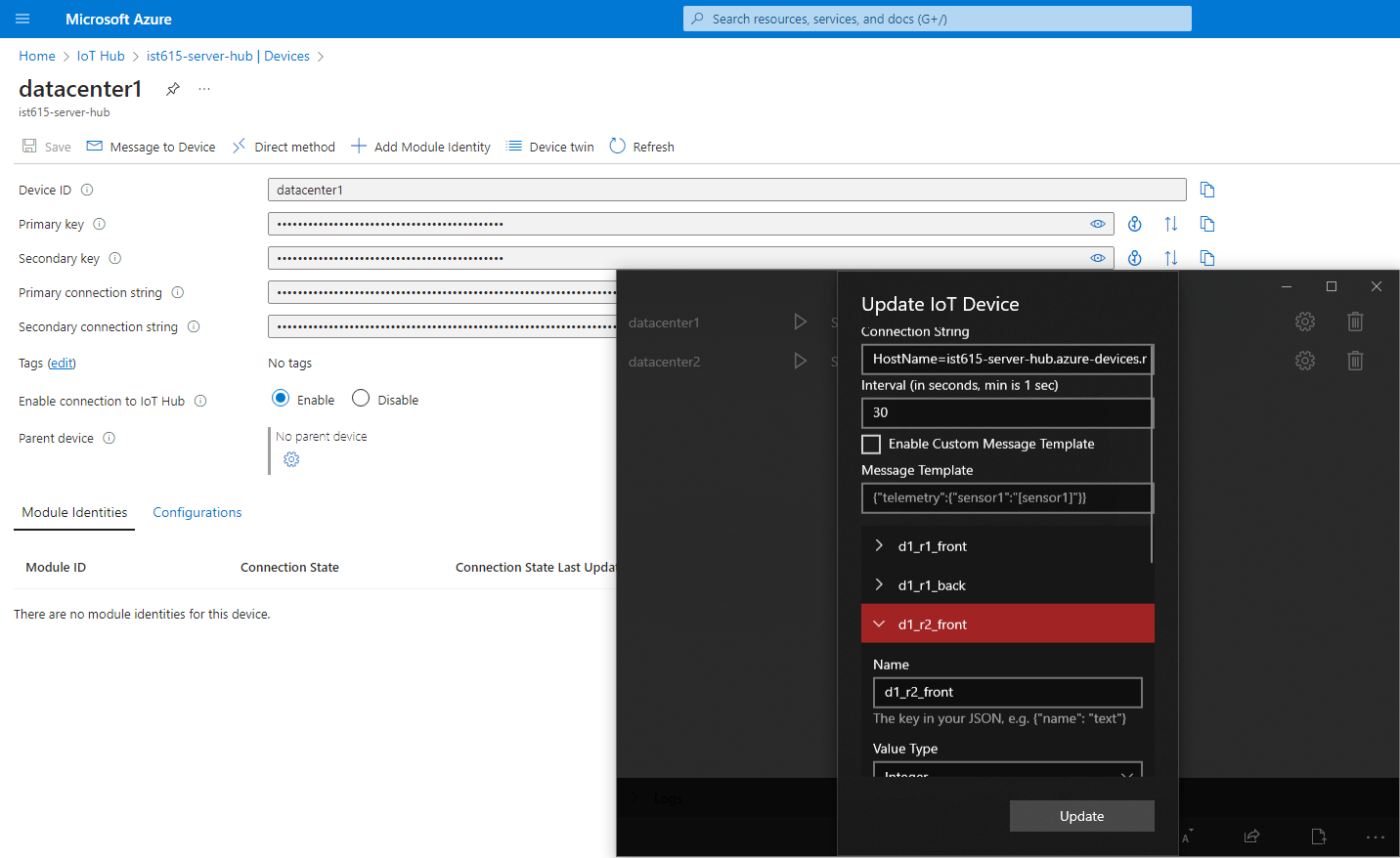
Next, we added devices to the IoT Hub which would be used to connect to the simulated sensors. Figure 3 below shows the setup for one of these devices. All values were kept as default. Two devices were created, datacenter1 and datacenter2, which will act as the main devices that our various sensors will connect to. Each datacenter will contain two server racks, with temperature sensors that read the air intake temperature into the front of the server racks as well as the back exhaust of the server racks. Each datacenter will also have a central humidity sensor reading the overall humidity in the room.



*Figure 3: Azure IoT Hub Device Configuration*

Microsoft Device Simulator Configuration

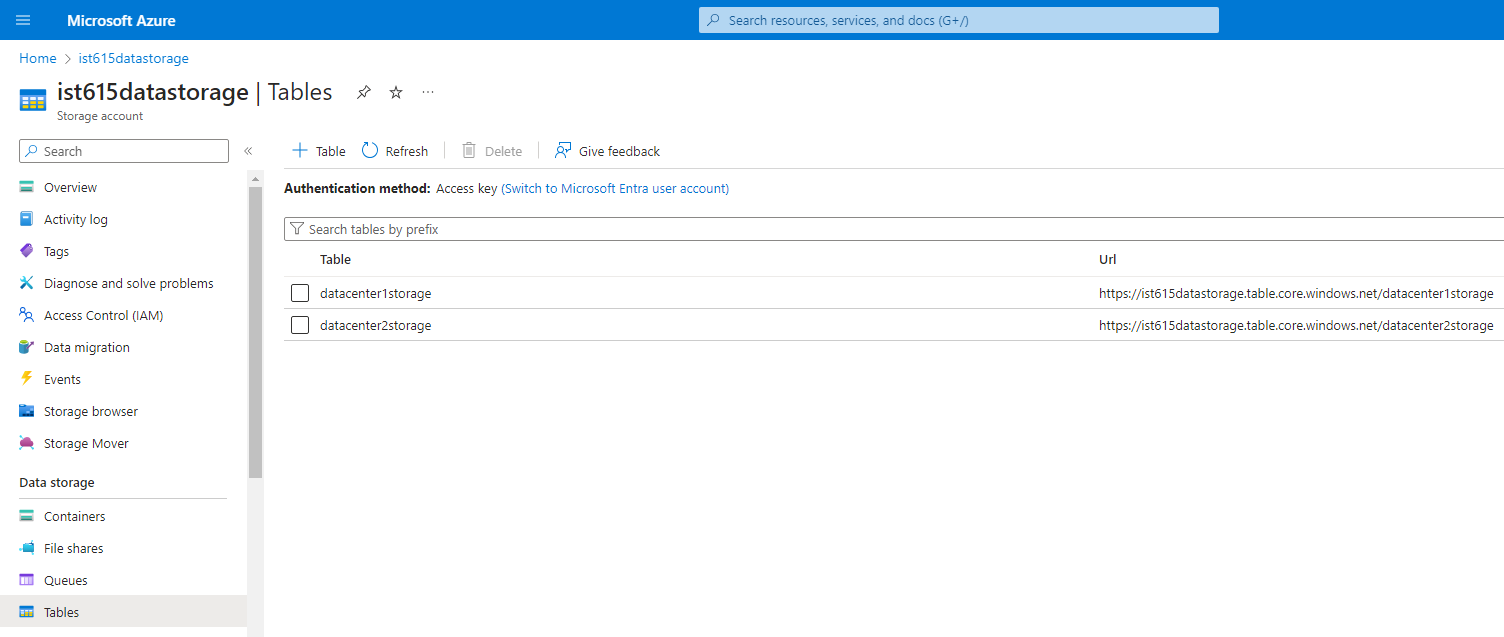
Now that the devices have been set up in the IoT Hub, we could create the simulated sensors with Microsoft's Device Simulator. On set up, the sensors can be directly connected to the IoT Hub devices with a connection string generated by Azure, as shown in Figure 4 below. Each required sensor is contained within the datacenter device connection. The naming convention followed a format where, for instance, “d1\_r1\_front\_temp” stood for datacenter 1, server rack 1, front intake temperature. The sensors were each configured to send random integers within a set range every 30 seconds.



*Figure 4: Device Simulator Sensor Configuration Alongside Azure IoT Hub Device Page*

Azure Storage

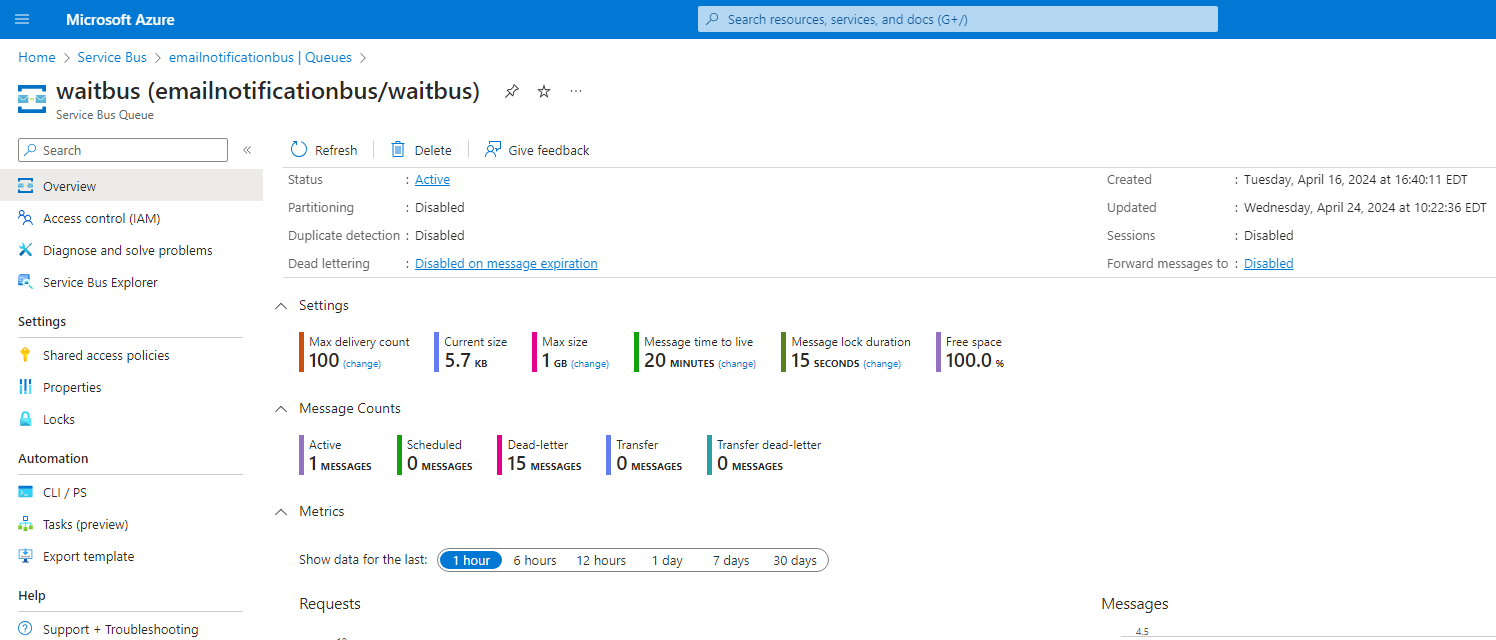
Following the setup of the devices, we can now create the Azure storage service that will house the data stream for long term analysis and dashboard creation. This was done by utilizing the table storage service as shown in Figure 5.



*Figure 5: Azure Storage Container for our Environmental Data*

Azure Service Bus

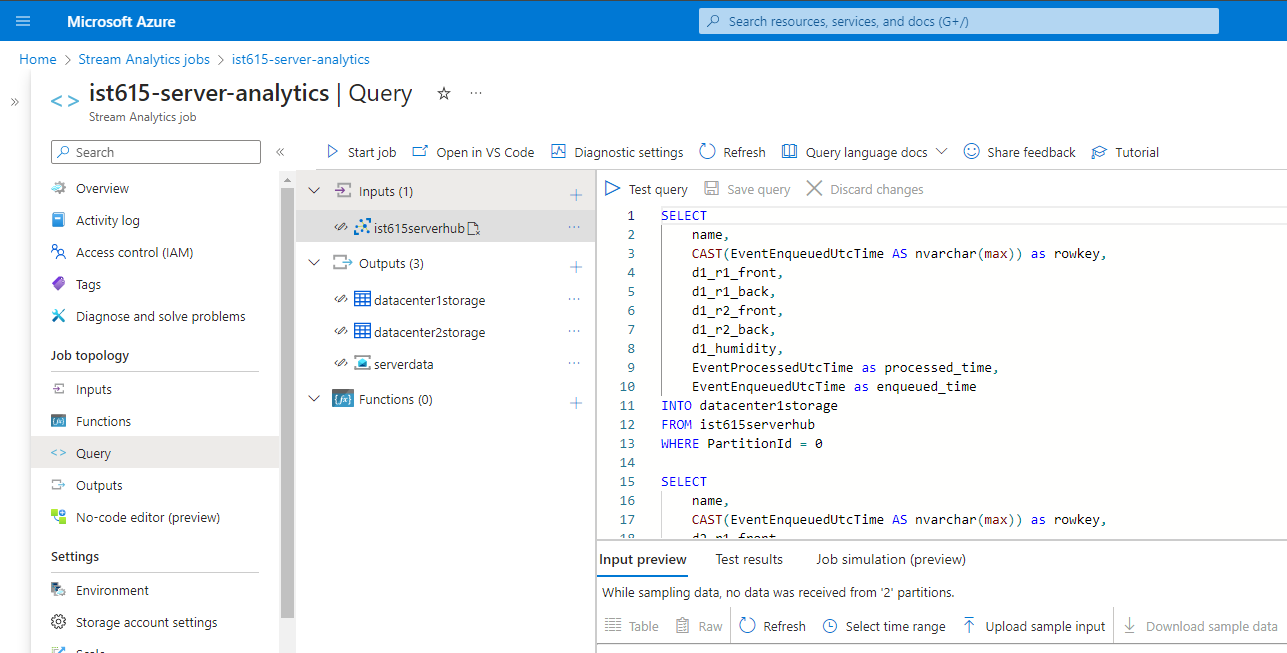
In order for our Azure Logic App to work, we first needed to create Service Bus Queues which would take in messages from Stream Analytics upon the breaching of a threshold and track the status of the alert. Upon the breaching of a threshold, a message will be sent to a queue called “serverdata” which will trigger the requisite Logic App to send an email notification. However, with a constant stream of readings coming in, this will cause the email notification to spam out continuously until the readings return to normal. In order to prevent this, we created a second queue, called “waitbus” which will receive a message from the Logic App upon the initial sending of a notification. The queue was configured such that this message will have a Time-to-Live (TTL) of 20 minutes, and will be consulted first by the Logic App at every trigger to see if it has been triggered in the previous 20 minutes. Thus, users will only see emails every 20 minutes. This value can be easily changed if desired. The Max Delivery Count was also set to a higher number, such that any queued message could be passed back and forth from the Logic App workflow for that 20 minutes. We utilized the following tutorial for basic setup and configuration of both the Service Bus and Logic App: <https://learn.microsoft.com/en-us/azure/iot-hub/iot-hub-monitoring-notifications-with-azure-logic-apps>



*Figure 6: Waitbus Service Bus Configuration*

Azure Stream Analytics

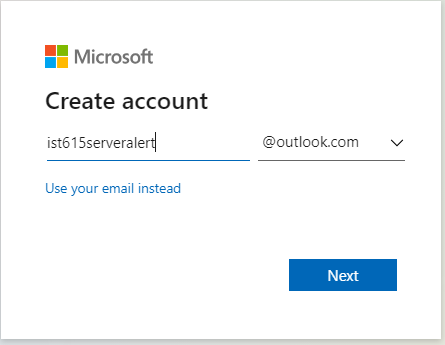
The next service to be configured is the Stream Analytics Job to take in the data stream from the IoT Hub. The job was configured with default settings, and then a query was created written in SQL to organize the incoming data stream as shown in Figure 7 below. The input was set as the IoT Hub we created, and the first output as the storage tables. The output appends the incoming data stream into two separate tables, one for each datacenter. The serverdata Service Bus queue was also made as the second output. Code was written in SQL to send messages to this queue when any readings breach our setpoint thresholds. The code used can be found in the Appendix.

**

*Figure 7: Stream Analytics Job Query*

Microsoft Outlook

Now with a working data stream, we could prepare the notification function. First, we created a Microsoft Outlook account specifically for our datacenter alert. While not an Azure service, we had to create our own account in order to bypass the 2FA requirements on our own personal .syr email account.

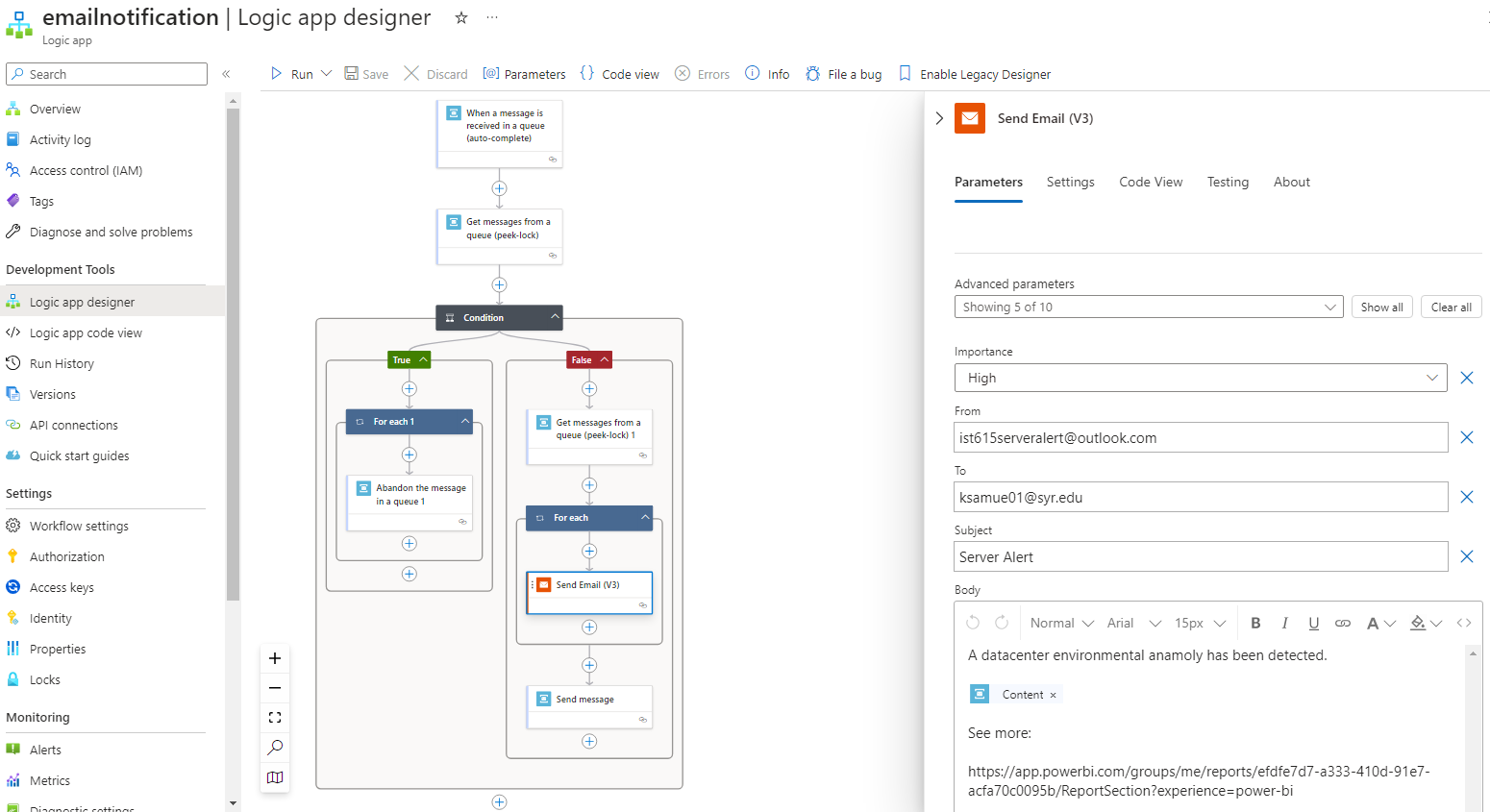


*Figure 8: New Microsoft Account for sending Outlook email Alerts*

Azure Logic App

While our Logic App was mostly similar to the tutorial referenced in our Azure Service Bus section, we differed in our initial trigger for the workflow. Instead of the device message containing the alert which triggered a message to enter the queue, we designed our stream analytics logic to send a message into the queue upon the passing of our set threshold. As described in that section, our Logic App was setup such that it would be triggered by the detection of a message entering the serverdata queue. The logic would then check to see if there were any messages present in our waitbus queue. A condition was set such that if the waitbus was empty, then the logic would go on to send an email notification using our newly created Outlook account because this would be its first time seeing a threshold breach. The email includes data from the message that initially triggered the workflow and a link to PowerBI analytics. Following the email, it would then send a message to our waitbus to sit in its queue for the next 20 minutes. Upon subsequent messages, the condition would fail as there would now be an active message in the waitbus queue, until the message was dropped after 20 minutes.

Importantly, the waitbus message must be abandoned following the completion of the condition check. This is because when the logic checks waitbus for messages, it uses a peek-lock, which grabs the message from the queue. Abandoning the message returns the message to the queue so that it can be picked up again. Additionally, the logic was set with a parallelism of 1, meaning only one instance of the logic can run at a time. This is because if the data steam enters the logic faster then the waitbus message can be returned, it may see the queue as “empty” and spam out further email notifications. Figure 9 below shows the codeless logic design and parallelism setting in our trigger. The actual code used can be found in the Appendix.

**

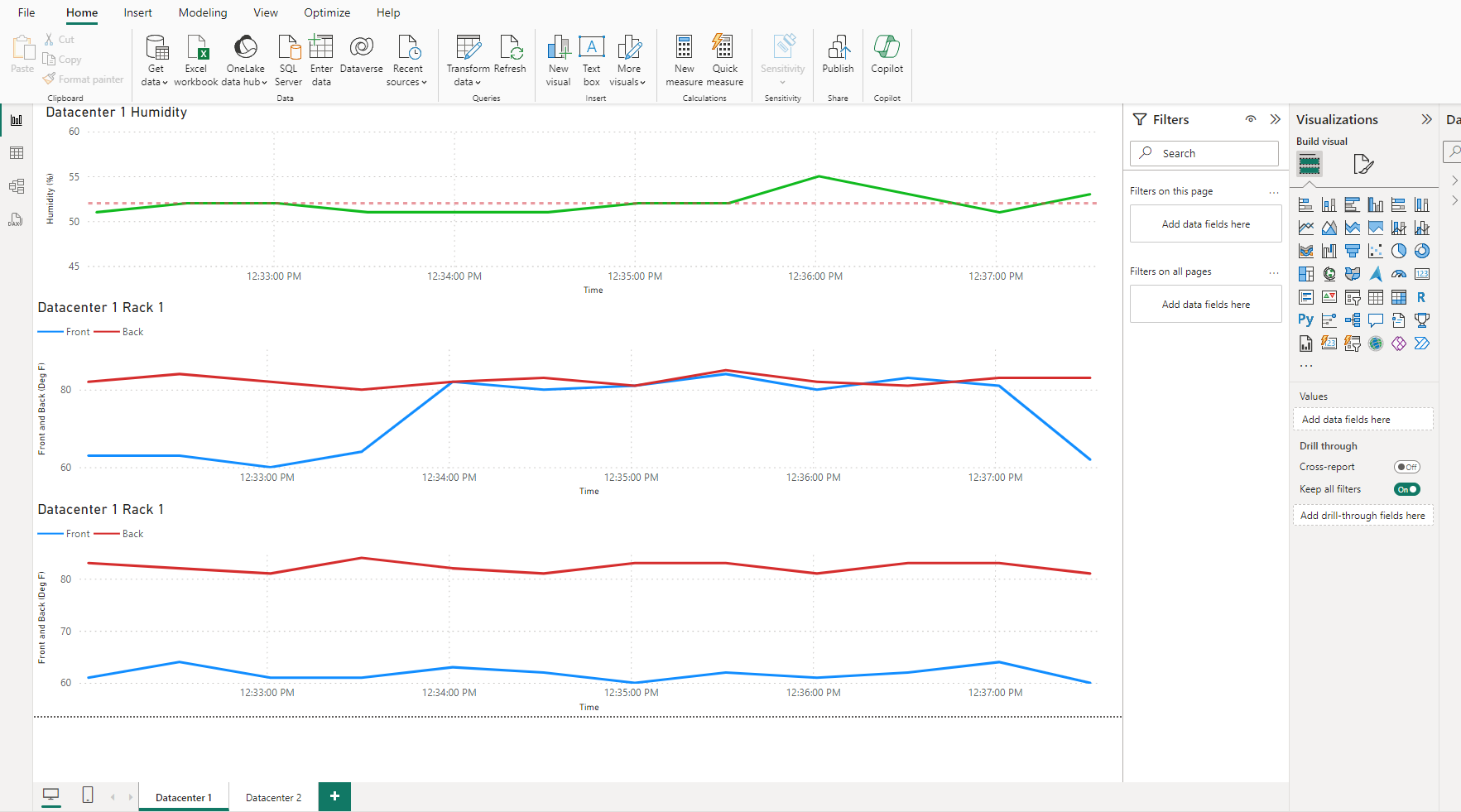
*Figure 9: Azure Logic App Workflow Configuration*

PowerBI

The final part of our project involved creating dashboards for analysis based on our stored data center environmental data in Azure Storage. Due to the high cost of running the Azure PowerBI embedded service online, we used a local desktop app which connected to the live Storage Container.

For each datacenter, line graphs were configured showing the overall server room humidity and rack temperatures. These values were plotted over the device’s enqueued time for the past 24 hours. The times were converted from UTC to EST by creating a new time column. Figure 10 below shows the graphs for one such datacenter.

The graphs could then be published online under Microsoft’s app.powerbi.com site. Frustratingly, we weren't able to find a way for the data pull to be auto-refreshed on the online app version. The graphs would first be updated on the desktop version and then pushed up to the online site. In theory, if we had utilized the Azure PowerBI, we believe we could have written an additional step in our Logic App workflow that would have refreshed the charts upon notification of a datacenter event. This would allow responding personnel to immediately access the most up to date trend data directly from the notification email.



*Figure 10: PowerBI Desktop Visuals*

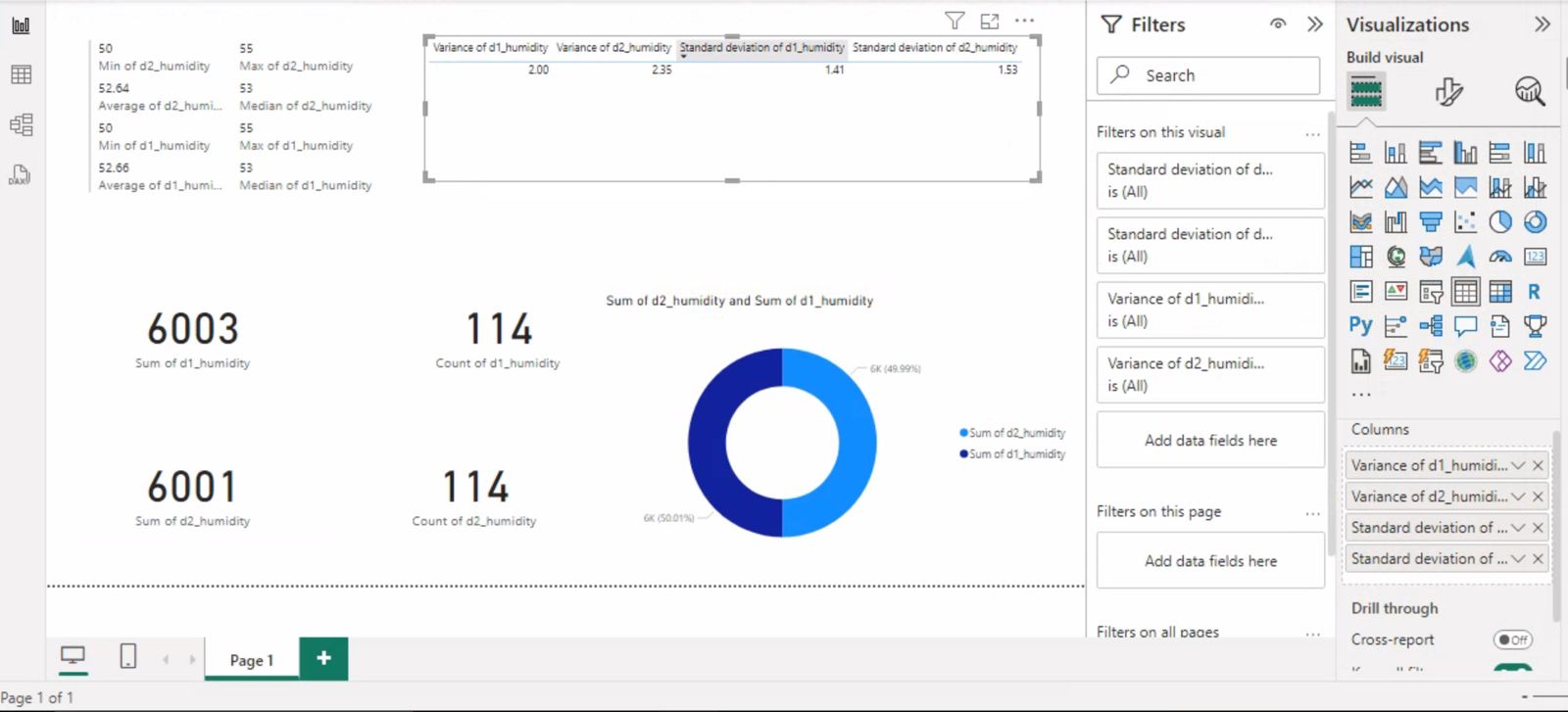
Figure 11 below shows the long-term analysis of the temperature data of our datacenters which is pulled from the storage. These are some insights to incorporate based on the Power BI analysis,

Humidity Levels: The study presents a number of statistical measurements pertaining to the levels of humidity particularly for the fields d1\_humidity and d2\_humidity. Humidity information from two distinct sources or locations is probably represented by these fields. The variable d1\_humidity has a minimum value of 52.86 and a maximum value of 55. Similar to this, d2\_humidity has a minimum value of 50 and a maximum value of 55. These ranges help to clarify the general variation in humidity levels observed in the data. The data for d1\_humidity exhibits a symmetric distribution with an average (mean) value of 53 and a median value of 53. There may be a distributional skew for d2\_humidity as the average is 50 and the median is 55.

Variance and Standard Deviation: The measurements of variance and standard deviation provide information about how evenly distributed the humidity data are. In contrast to d2\_humidity which has a variance of 2.35 and a standard deviation of 1.53, d1\_humidity has a variance of 2 and a standard deviation of 1.41. Values that are higher signify increased data variability.

Donut Chart: The donut chart shows the counts (both 114) for d1\_humidity (6003) and d2\_humidity (6001) as well as the sum of their values (6001). For each field the total humidity levels and the number of data points are quickly compared in this chart.

Data Distribution: The presence of metrics like median variance and standard deviation implies that the humidity data may follow a normal or roughly normal distribution even though the analysis does not go into great detail about this. With the help of these realizations the central tendency dispersion and overall range of the humidity data can be discussed. Based on the data and visualizations supplied you can also compare the humidity levels in the two fields (d1\_humidity and d2\_humidity).



*Figure 11: PowerBI Desktop Visuals*

Issues / Lessons Learned

There were some difficulties encountered with limiting the rate of email notifications. Without any sort of mechanism to slow the logic workflow, the rate at which data was coming in would trigger far more emails than acceptable. However, we also wanted to ensure that a notification would be sent as soon as an alert was detected. We briefly considered trying to write some sort of grouping in our SQL code in Stream Analytics, but this was abandoned because of this immediacy issue.

Utilizing the second service bus with a set message TTL was found to be a better solution, but without knowing exactly how the service bus queues functioned, we were surprised when email notifications were still being sent out with every out of spec datapoint. It was only after some reading and experimentation that we realized our logic app was firstly, not returning messages to the waitbus’s active queue, and secondly, running instances in parallel and mistaking the queue as empty. Once we knew to return (abandon) messages when the logic condition failed, and to limit the number of parallel functions, we were able to get the notification feature to run as desired.

Demo Video Link: <https://video.syr.edu/media/t/1_1l5tcfix>

Appendix

Azure Logic App Code

{

"definition": {

"$schema": "https://schema.management.azure.com/providers/Microsoft.Logic/schemas/2016-06-01/workflowdefinition.json#",

"actions": {

"Condition": {

"actions": {

"For\_each\_1": {

"actions": {

"Abandon\_the\_message\_in\_a\_queue\_1": {

"inputs": {

"host": {

"connection": {

"name": "@parameters('$connections')['servicebus-1']['connectionId']"

}

},

"method": "post",

"path": "/@{encodeURIComponent(encodeURIComponent('waitbus'))}/messages/abandon",

"queries": {

"lockToken": "@items('For\_each\_1')?['LockToken']",

"queueType": "Main",

"sessionId": ""

}

},

"type": "ApiConnection"

}

},

"foreach": "@body('Get\_messages\_from\_a\_queue\_(peek-lock)')",

"type": "Foreach"

}

},

"else": {

"actions": {

"For\_each": {

"actions": {

"Send\_Email\_(V3)": {

"inputs": {

"body": {

"Body": "<p>A datacenter environmental anamoly has been detected.</p><br><p>@{base64ToString(items('For\_each')?['ContentData'])}</p><br><p>See more: </p><br><p>https://app.powerbi.com/groups/me/reports/efdfe7d7-a333-410d-91e7-acfa70c0095b/ReportSection?experience=power-bi</p><br><p>https://portal.azure.com/#@syr.edu/resource/subscriptions/e7f6069e-5a3f-4a46-a22e-7c441a9f6255/resourceGroups/IST615-FinalProject/providers/Microsoft.StreamAnalytics/streamingjobs/ist615-server-analytics/queryV2</p>",

"From": "ist615serveralert@outlook.com",

"Importance": "High",

"Subject": "Server Alert",

"To": "ksamue01@syr.edu"

},

"host": {

"connection": {

"name": "@parameters('$connections')['smtp']['connectionId']"

}

},

"method": "post",

"path": "/SendEmailV3"

},

"type": "ApiConnection"

}

},

"foreach": "@body('Get\_messages\_from\_a\_queue\_(peek-lock)\_1')",

"runAfter": {

"Get\_messages\_from\_a\_queue\_(peek-lock)\_1": [

"Succeeded"

]

},

"type": "Foreach"

},

"Get\_messages\_from\_a\_queue\_(peek-lock)\_1": {

"inputs": {

"host": {

"connection": {

"name": "@parameters('$connections')['servicebus-1']['connectionId']"

}

},

"method": "get",

"path": "/@{encodeURIComponent(encodeURIComponent('serverdata'))}/messages/batch/peek",

"queries": {

"maxMessageCount": 1,

"queueType": "Main"

}

},

"type": "ApiConnection"

},

"Send\_message": {

"inputs": {

"body": {

"SessionId": "@triggerBody()?['SessionId']"

},

"host": {

"connection": {

"name": "@parameters('$connections')['servicebus-1']['connectionId']"

}

},

"method": "post",

"path": "/@{encodeURIComponent(encodeURIComponent('waitbus'))}/messages",

"queries": {

"systemProperties": "None"

}

},

"runAfter": {

"For\_each": [

"Succeeded"

]

},

"type": "ApiConnection"

}

}

},

"expression": {

"and": [

{

"equals": [

"@empty(outputs('Get\_messages\_from\_a\_queue\_(peek-lock)').body)",

false

]

}

]

},

"runAfter": {

"Get\_messages\_from\_a\_queue\_(peek-lock)": [

"Succeeded"

]

},

"type": "If"

},

"Get\_messages\_from\_a\_queue\_(peek-lock)": {

"inputs": {

"host": {

"connection": {

"name": "@parameters('$connections')['servicebus-1']['connectionId']"

}

},

"method": "get",

"path": "/@{encodeURIComponent(encodeURIComponent('waitbus'))}/messages/batch/peek",

"queries": {

"maxMessageCount": 1,

"queueType": "Main",

"sessionId": ""

}

},

"runAfter": {},

"type": "ApiConnection"

}

},

"contentVersion": "1.0.0.0",

"outputs": {},

"parameters": {

"$connections": {

"defaultValue": {},

"type": "Object"

}

},

"triggers": {

"When\_a\_message\_is\_received\_in\_a\_queue\_(auto-complete)": {

"evaluatedRecurrence": {

"frequency": "Second",

"interval": 10

},

"inputs": {

"host": {

"connection": {

"name": "@parameters('$connections')['servicebus-1']['connectionId']"

}

},

"method": "get",

"path": "/@{encodeURIComponent(encodeURIComponent('serverdata'))}/messages/head",

"queries": {

"queueType": "Main"

}

},

"recurrence": {

"frequency": "Second",

"interval": 10

},

"runtimeConfiguration": {

"concurrency": {

"runs": 1

}

},

"type": "ApiConnection"

}

}

},

"parameters": {

"$connections": {

"value": {

"servicebus-1": {

"connectionId": "/subscriptions/e7f6069e-5a3f-4a46-a22e-7c441a9f6255/resourceGroups/IST615-FinalProject/providers/Microsoft.Web/connections/servicebus-1",

"connectionName": "servicebus-1",

"id": "/subscriptions/e7f6069e-5a3f-4a46-a22e-7c441a9f6255/providers/Microsoft.Web/locations/eastus/managedApis/servicebus"

},

"smtp": {

"connectionId": "/subscriptions/e7f6069e-5a3f-4a46-a22e-7c441a9f6255/resourceGroups/IST615-FinalProject/providers/Microsoft.Web/connections/smtp",

"connectionName": "smtp",

"id": "/subscriptions/e7f6069e-5a3f-4a46-a22e-7c441a9f6255/providers/Microsoft.Web/locations/eastus/managedApis/smtp"

}

}

}

}

}

Azure Stream Analytics Code

SELECT

name,

CAST(EventEnqueuedUtcTime AS nvarchar(max)) as rowkey,

d1\_r1\_front,

d1\_r1\_back,

d1\_r2\_front,

d1\_r2\_back,

d1\_humidity,

EventProcessedUtcTime as processed\_time,

EventEnqueuedUtcTime as enqueued\_time

INTO datacenter1storage

FROM ist615serverhub

WHERE PartitionId = 0

SELECT

name,

CAST(EventEnqueuedUtcTime AS nvarchar(max)) as rowkey,

d2\_r1\_front,

d2\_r1\_back,

d2\_r2\_front,

d2\_r2\_back,

d2\_humidity,

EventProcessedUtcTime as processed\_time,

EventEnqueuedUtcTime as enqueued\_time

INTO datacenter2storage

FROM ist615serverhub

WHERE PartitionId = 1

SELECT

d1\_r1\_front,

d1\_r1\_back,

d1\_r2\_front,

d1\_r2\_back,

d2\_r1\_front,

d2\_r1\_back,

d2\_r2\_front,

d2\_r2\_back,

d1\_humidity,

d2\_humidity

INTO serverdata

FROM ist615serverhub

WHERE

d1\_humidity > 60 OR d2\_humidity > 60 OR

d1\_r1\_front > 68 OR d1\_r2\_front > 68 OR

d2\_r1\_front > 68 OR d2\_r2\_front > 68 OR

d1\_r1\_back > 88 OR d1\_r2\_back > 88 OR

d2\_r1\_back > 88 OR d2\_r2\_back > 88