Building a Low-Cost Micro Weather Station for Plant Disease Forecasting

Problem Statement

Herbivorous insects destroy one-fifth of the world's total crop production annually. This destruction results in loss for the poor and marginal farmers. There are multiple pest attack prediction models which can provide guidance to the farmers so that they can take some preventive measures.

All these models depend heavily on the local weather information. Simple weather parameters, such as temperature, humidity, pressure, wind speed and direction are needed for these models to work properly.

However, the density of weather station in India is not enough to provide predictions at village levels. This prevents micro level predictions for the villages to take preventive actions. This problem can be solved by deploying low-cost weather stations at panchayat level. This blog shows how to make a simple low-cost weather station using Raspberry Pi and how Azure technologies can help in managing these weather stations and gather data from these stations.

Solution

The advancement of electronics has enabled us to build a simple micro-weather station using Raspberry Pi along with sensors to fetch weather data at a low cost. While Azure can be used to manage the weather station and accept weather data for further analysis.

In this blog, we will create a simple micro-weather station using Raspberry Pi 2B based board and a DHT11 temperature and humidity sensor. We will also leverage Azure IoT Hub and Azure IoT Edge to manage the devices and accept data into cloud. In order to extract and store data, we will use Azure Stream Analytics Job which will store in Azure Table Storage.

The device shown here can be easily extended to gather additional weather parameters. Likewise, the Stream Analytics job can also be enriched to do further analysis. The data stored in Azure Table Storage, will allow it to be used for machine learning models to do predictive analysis.

Architecture

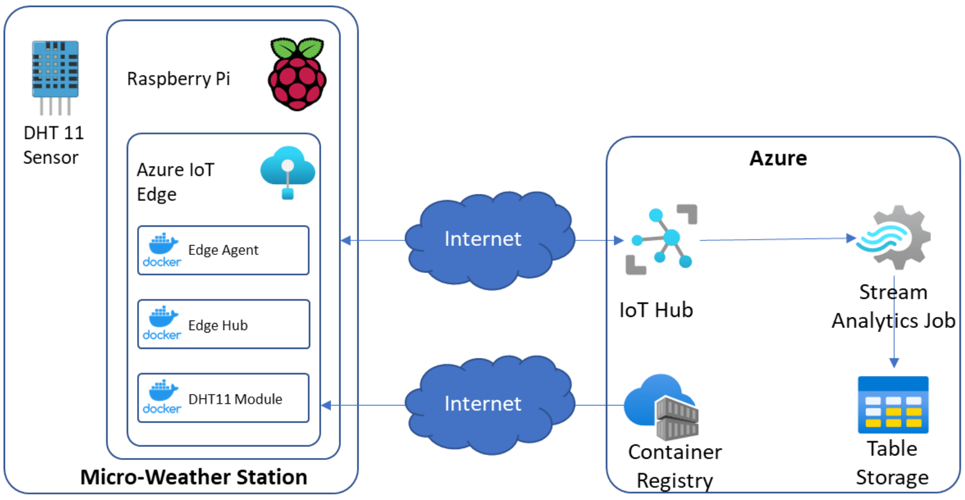


Figure 1: Micro-Weather Station Architecture

Figure 1 provides a high-level view of the setup. The heart of the micro-weather station is the Raspberry Pi which has a DHT11 based temperature and humidity sensor wired to it. Azure IoT Edge is installed on Raspberry Pi, which hosts the DHT11 custom module. The IoT Edge is configured to communicate with IoT Hub on Azure. The IoT Edge pulls the Docker image for the module from Azure Container Registry.

On the Azure side, a Stream Analytics Job pulls the data from IoT Hub, does some simple transformations, and stores the data in Azure Table Storage.

Implementation

In this section I will describe the steps I took for implementing the micro-weather station and storing the data in Azure Table Storage. I will not cover the basics of Azure subscriptions, Azure CLI, Docker, and Raspberry Pi as there are plenty of articles available in internet.

Configuring IoT Hub

Refer to the script below which I executed:

```
 1. #!/bin/bash
 2.
 3. # Add IoT extension if not present
 4. az extension add --name azure-iot
 5.
 6. # Create resource group
 7. az group create --name agri-iot --location centralindia
 8.
 9. # Create a standard IoT Hub
10. az iot hub create --resource-group agri-iot --name agri-hub001 --sku S1 \
11.           --location centralindia
12.
13. # Provision device
14. az iot hub device-identity create --hub-name agri-hub001 --device-id edge01 \
15.           --edge-enabled
16.
17. # List registered devices and their status
18. az iot hub device-identity list --hub-name agri-hub001 --output table
19.
20. # Retrieve Connection String
21. az iot hub device-identity connection-string show --device-id edge01 \
22.           --hub-name agri-hub001
23.
24. # Create Container Registry
25. az acr create --name acragri --location centralindia --resource-group agri-iot \
26.           --sku Basic --admin-enabled true
```

Figure 2: IoT Hub Provisioning Script

Line 4 adds the Azure IoT extension. Line 7 creates the resource group. Line 10 creates an IoT Hub named `agri-hub001`. Line 15 provisions an edge device. The edge device in this case is the micro-weather station which we will use to collect weather data. Note that the edge enabled switch is enabled. Line 21 retrieves the connection string for the device to connect to this hub. Note down the connection string, as this will be used later while provisioning the IoT Edge on Raspberry Pi. Line 21 creates the container registry where the image for the DHT11 module will be kept.

Wiring Raspberry Pi Board

DHT11 is a simple sensor consisting of 3 pins. 2 Pins are for 5.5V and GND, and the one pin is for the data transfer.

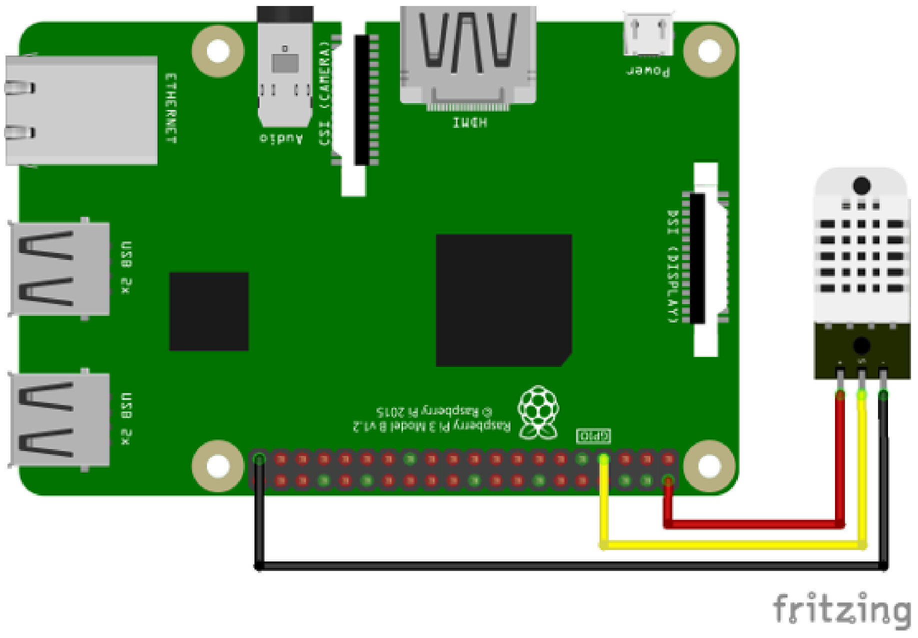


Figure 3: Wiring Diagram for Raspberry Pi and DHT11

Figure 3 shows the wiring diagram which I followed. Left pin is connected to +5.5V on Raspberry Pi GPIO, this is shown in red. Right pin is connected to GND pin, which is shown in black. Middle pin is the data pin, which is connected to the GPIO 4 pin on Raspberry Pi.

Installing & Configuring IoT Edge on Raspberry Pi

I flashed Raspbian image on a micro-SD card and started Raspberry Pi. The Raspberry Pi is connected to the internet through ethernet cable. Once booting completes, I logged into the system, updated it by issuing `apt update` and `apt upgrade` commands.

Once done I executed the following commands to add Microsoft PPA repository:

```
curl https://packages.microsoft.com/config/debian/11/packages-microsoft-prod.deb > ./packages-microsoft-prod.deb
sudo apt install ./packages-microsoft-prod.deb
```

Figure 4: Script to Add Microsoft PPA

Once added I installed moby engine by executing the following commands:

```
sudo apt update
sudo apt install moby-engine
```

Figure 5: Code to install moby engine

I set the logging for the moby engine to *local*, as suggested by the Microsoft documentation. I did this by updating the `/etc/docker/daemon.json` file. I had added the following JSON entry:

```
{
  "log-driver": "local"
}
```

Figure 6: JSON entry for Docker Daemon

Then I restarted the moby engine by issuing `sudo service docker restart` command to take the settings into effect.

Once this is done, I installed the IoT Edge runtime by executing the following command:

```
sudo apt install aziot-edge defender-iot-micro-agent-edge
```
Figure 7: Command to install IoT Edge runtime

After the runtime is installed, I set the device connection string, which I saved earlier, by executing the following commands:

```
sudo iotedge config mp --connection-string '<Device Connection String>'
sudo iotedge config apply -c '/etc/aziot/config.toml'
```
Figure 8: Code to configure IoT Edge

After this, the edge device started to appear online on Azure Portal.

Creating & Deploying the DHT11 Module

Before I started to code, I installed IoT Edge module development kit. For which I ran the following commands:

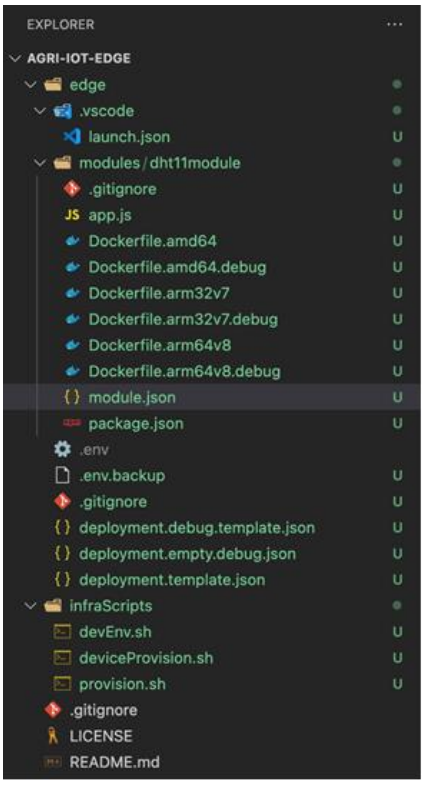
```
pip3 install iotedgedev
sudo npm install --global yo
sudo npm install --global generator-azure-iot-edge-module
```
Figure 9: Development Environment Creation Commands

The module helper `iotedgedev` uses yeoman to create node.js project. After installation, I ran `iotedgedev solution init --template nodejs` command to create the project. After the execution of the command, the project gets created, as shown in Figure 10, under *modules* folder.

I changed the name of the folder to `dht11module` to match the module name I preferred.

The *app.js* file contains the main code. The generator creates multiple Docker files for different types of processors. This is meant to create a module which can be used across multiple different types of boards' CPU architectures. Since I am using Raspberry Pi 2B, I used *arm32v7* files.

The generator also generates deployment templates for debug and production environments. The main difference between them is that in *debug* variants, a web socket port is opened for online debugging. I had to update these deployment files and used these for deploying the module to IoT Hub. This will be covered later in this section.



```
edge
  .vscode
    launch.json                      U
  modules / dht11module
    .gitignore                       U
    app.js                           U
    Dockerfile.amd64                 U
    Dockerfile.amd64.debug           U
    Dockerfile.arm32v7               U
    Dockerfile.arm32v7.debug         U
    Dockerfile.arm64v8               U
    Dockerfile.arm64v8.debug         U
    module.json                      U
    package.json                     U
  .env
  .env.backup                        U
  .gitignore                         U
  deployment.debug.template.json     U
  deployment.empty.debug.json        U
  deployment.template.json           U
infraScripts
  devEnv.sh                          U
  deviceProvision.sh                 U
  provision.sh                       U
.gitignore
LICENSE
README.md
```

Figure 10: Module Project Layout

Now let's look at the main code listing, i.e. *app.js* file:

```
'use strict';

var Transport = require('azure-iot-device-mqtt').Mqtt;
var Client = require('azure-iot-device').ModuleClient;
var Message = require('azure-iot-device').Message;
var sensor = require("node-dht-sensor");

Client.fromEnvironment(Transport, function (err, client) {
  if (err) {
    throw err;
  } else {
    client.on('error', function (err) {
```

```
13.       throw err;
14.     });
15.
16.     // connect to the Edge instance
17.     client.open(function (err) {
18.       if (err) {
19.         throw err;
20.       } else {
21.         console.log('IoT Hub module client initialized');
22.
23.         setInterval(() => {
24.           console.log("Fetching Temperature and Humidity...");
25.           sensor.read(11, 4, function(err, temperature, humidity) {
26.             if (!err) {
27.               console.log(`temp: ${temperature}°C, humidity: ${humidity}%`);
28.               var message = { temperature: temperature, humidity: humidity };
29.               var msgText = JSON.stringify(message);
30.               var outputMsg = new Message(msgText);
31.               client.sendOutputEvent('tempHumid', outputMsg, printResultFor('Sending
received message'));
32.             }
33.           });
34.         }, 10000);
35.       }
36.     });
37.   }
38. });
39.
40. // Helper function to print results in the console
41. function printResultFor(op) {
42.   return function printResult(err, res) {
43.     if (err) {
44.       console.log(op + ' error: ' + err.toString());
45.     }
46.     if (res) {
47.       console.log(op + ' status: ' + res.constructor.name);
48.     }
49.   };
50. }
51.
```

Figure 11: Listing of app.js

The code uses the `ModuleClient` object to interact with IoT Edge. Lines 8 thru 21 initializes the `client` object and gets it ready to run the code. Lines 23 thru 34 starts a timer which extracts the temperature and humidity data from the sensor every 10 seconds. Note how we called the sensor library to fetch the data in line 25. The call provides three parameters, the first one, "11", is to signify that the sensor is a DHT11 sensor, the second one, "4", signifies that the data pin of the sensor is connected to GPIO 4 pin on the board and the third parameter is a call back function which gets called after the data is retrieved. After retrieving the data, it forms a JSON message and sends the message as an output event to the IoT Hub. Lines 40 thru 50 is a simple call back function which prints the result of the message queueing to console log.

Now lets look at the Docker file:

```
FROM arm32v7/node:10-slim

RUN apt-get update && apt-get install -y --no-install-recommends \
  python3.5 \
  python3-pip \
  build-essential \
  make \
  && \
  apt-get clean && \
  rm -rf /var/lib/apt/lists/*

```

```
12. WORKDIR /app/
13.
14. COPY package*.json ./
15.
16. RUN npm install --production
17.
18. COPY app.js ./
19.
20. EXPOSE 9229
21.
22. USER node
23.
24. CMD ["node", "--inspect=0.0.0.0:9229", "app.js"]
25.
```

Figure 12: Listing of Dockerfile.arm32v7.debug file

The DHT11 library uses Python modules and C libraries to interact with the GPIO pins. Hence, you will notice that in lines 3 thru 10 we are installing a whole host of libraries. Since this is *debug* version, we are exposing port 9229 in line 20 and added `--inspect=0.0.0.0:9229` switch in line 24 to allow the debugger to connect to the container.

Now, we are ready to build the Docker image and push it to the registry. I ran the following command to build the Docker image locally. Of course, you will need to have Docker installed on your local machine.

```
docker build --rm -f "./modules/dht11module/Dockerfile.arm32v7.debug" -t acragri.azurecr.io/dht11module:0.0.6-arm32v7 "./modules/dht11module"
```

Figure 13: Docker command to build local image

Here we ran a standard *docker build* command and directly tagged the image to the Azure Container Registry which we created earlier. Once build completed, I ran the following commands to push the image to Azure Container Registry.

```
az acr login --name acragri
docker push acragri.azurecr.io/dht11module:0.0.6-arm32v7
```

Figure 14: Commands to push the local image to Azure Container Registry

Now I had to deploy this image to the Raspberry Pi using the IoT Edge and IoT Hub. In order to do so, I need to finalize the deployment JSON file.

```
{
  "$schema-template": "1.0.0",
  "modulesContent": {
    "$edgeAgent": {
      "properties.desired": {
        "schemaVersion": "1.0",
        "runtime": {
          "type": "docker",
          "settings": {
            "minDockerVersion": "v1.25",
            "loggingOptions": "",
            "registryCredentials": {
              "acragri": {
                "username": "acragri",
                "password": "rshVr...snip...snip",
                "address": "acragri.azurecr.io"
              }
            }
          }
        },
        "systemModules": {
          "edgeAgent": {
            "type": "docker",
24.             "settings": {
25.               "image": "mcr.microsoft.com/azureiotedge-agent:1.4",
26.               "createOptions": "{}"
27.             }
28.           },
29.           "edgeHub": {
30.             "type": "docker",
31.             "status": "running",
32.             "restartPolicy": "always",
33.             "settings": {
34.               "image": "mcr.microsoft.com/azureiotedge-hub:1.4",
35.               "createOptions":
"{\"HostConfig\":{\"PortBindings\":{\"5671/tcp\":[{\"HostPort\":\"5671\"}],\"8883/tcp\":[{\"Host
Port\":\"8883\"}],\"443/tcp\":[{\"HostPort\":\"443\"}]}}}"
36.             }
37.           }
38.         },
39.         "modules": {
40.           "dht11module": {
41.             "version": "1.0",
42.             "type": "docker",
43.             "status": "running",
44.             "restartPolicy": "always",
45.             "settings": {
46.               "image": "acragri.azurecr.io/dht11module:0.0.6-arm32v7",
47.               "createOptions": "{\"HostConfig\": {\"Privileged\": true, \"Devices\": [{
\"PathOnHost\": \"/dev/gpiomem\", \"PathInContainer\": \"/dev/gpiomem\", \"CgroupPermissions\":
\"mrw\" }]}}"
48.             }
49.           }
50.         }
51.       }
52.     },
53.     "$edgeHub": {
54.       "properties.desired": {
55.         "schemaVersion": "1.0",
56.         "routes": {
57.           "dht11moduleToIoTHub": "FROM /messages/modules/dht11module/* INTO $upstream"
58.         },
59.         "storeAndForwardConfiguration": {
60.           "timeToLiveSecs": 7200
61.         }
62.       }
63.     }
64.   }
65. }
66.
```

Figure 15: Listing of deployment.debug.template.json

Lines 12 thru 18 provides the credentials of the Azure Container Registry I created. This can be retrieved from the Azure Portal. Line 35 opens the ports so that my module (and any other module) can interact with the *Edge Hub Module*. Lines 40 thru 49 defines the *dht11module*, which needs to be deployed to the edge. Of most interest is Line 47, which provides the create options for the module. Note that this container runs in *privileged* mode and the device */dev/gpiomem* is replicated inside the container. This is used by the sensor library to talk to the sensor. Lines 56 thru 58 defines the route. Here we are routing all events from *dht11modules* to IoTHub.

In order deploy, I issued the following command:

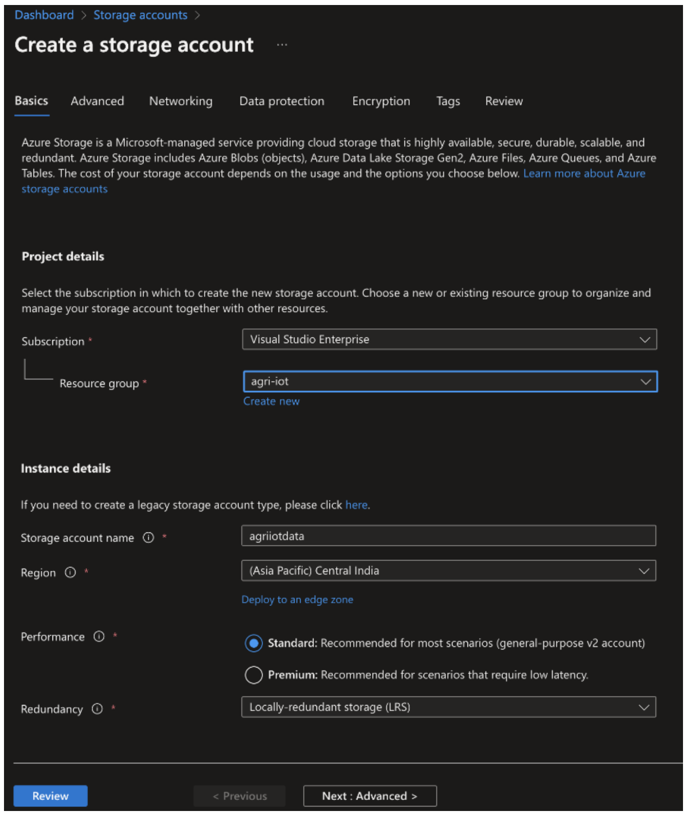
```
az iot edge set-modules --hub-name agri-hub001 --device-id edge01 --content
./deployment.debug.template.json –login "<IoT Hub Connection String>"
```

Figure 16: The command to deploy the module to edge

If everything goes well, you should see the *dht11module* container getting created in Raspberry Pi and the edge has started pushing the data to IoT Hub.

Creating Azure Stream Analytics Job

First I created a storage account in the portal as shown in Figure 17 below:



Dashboard > Storage accounts >

Create a storage account ...

Azure Storage is a Microsoft-managed service providing cloud storage that is highly available, secure, durable, scalable, and redundant. Azure Storage includes Azure Blobs (objects), Azure Data Lake Storage Gen2, Azure Files, Azure Queues, and Azure Tables. The cost of your storage account depends on the usage and the options you choose below. Learn more about Azure storage accounts

Project details

Select the subscription in which to create the new storage account. Choose a new or existing resource group to organize and manage your storage account together with other resources.

Subscription *: Visual Studio Enterprise

Resource group *: agri-iot

Create new

Instance details

If you need to create a legacy storage account type, please click here.

Storage account name *: agriiotdata

Region *: (Asia Pacific) Central India

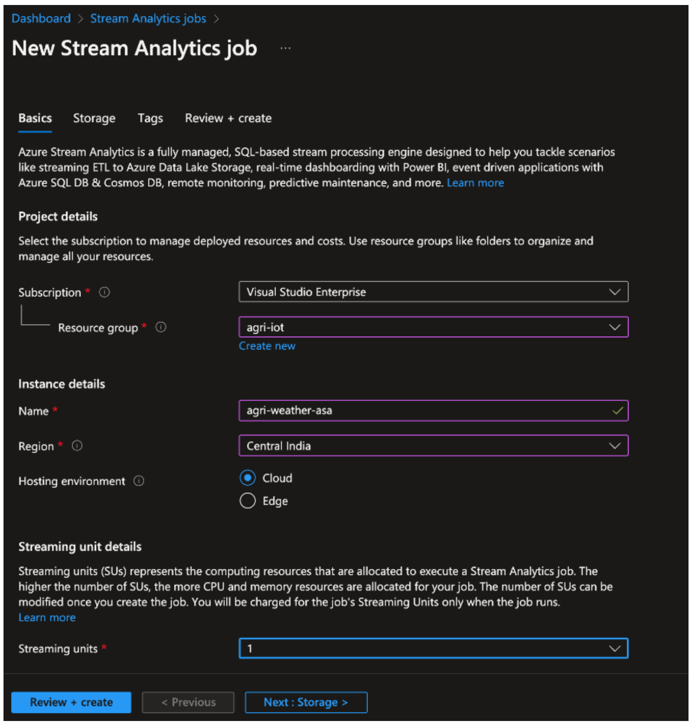
Deploy to an edge zone

Performance *: [✓ **Standard:** Recommended for most scenarios (general-purpose v2 account)] **Premium:** Recommended for scenarios that require low latency.

Redundancy *: Locally-redundant storage (LRS)

Figure 17: Create Storage Account Screenshot

Once the storage account got created, I went ahead and created the Stream Analytics Job as shown in Figure 18.



Dashboard > Stream Analytics jobs >

New Stream Analytics job ...

Azure Stream Analytics is a fully managed, SQL-based stream processing engine designed to help you tackle scenarios like streaming ETL to Azure Data Lake Storage, real-time dashboarding with Power BI, event driven applications with Azure SQL DB & Cosmos DB, remote monitoring, predictive maintenance, and more. Learn more

Project details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription *: Visual Studio Enterprise

Resource group *: agri-iot

Create new

Instance details

Name *: agri-weather-asa

Region *: Central India

Hosting environment: [✓ Cloud] Edge

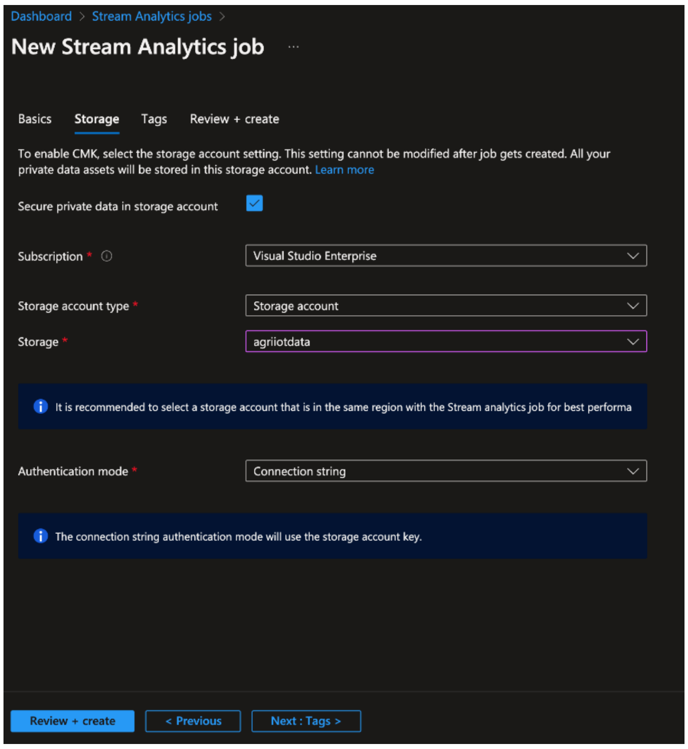
Streaming unit details

Streaming units (SUs) represents the computing resources that are allocated to execute a Stream Analytics job. The higher the number of SUs, the more CPU and memory resources are allocated for your job. The number of SUs can be modified once you create the job. You will be charged for the job's Streaming Units only when the job runs. Learn more

Streaming units *: 1

Figure 18: Create Stream Analytics Job Screenshot

In the storage section, I selected the storage account I created earlier as shown in Figure 18 and clicked "Review + Create".


Dashboard > Stream Analytics jobs >

New Stream Analytics job ...

Basics | Storage | Tags | Review + create

To enable CMK, select the storage account setting. This setting cannot be modified after job gets created. All your private data assets will be stored in this storage account. Learn more

Secure private data in storage account: ✓

Subscription *: Visual Studio Enterprise

Storage account type *: Storage account

Storage *: agriiotdata

It is recommended to select a storage account that is in the same region with the Stream analytics job for best performa

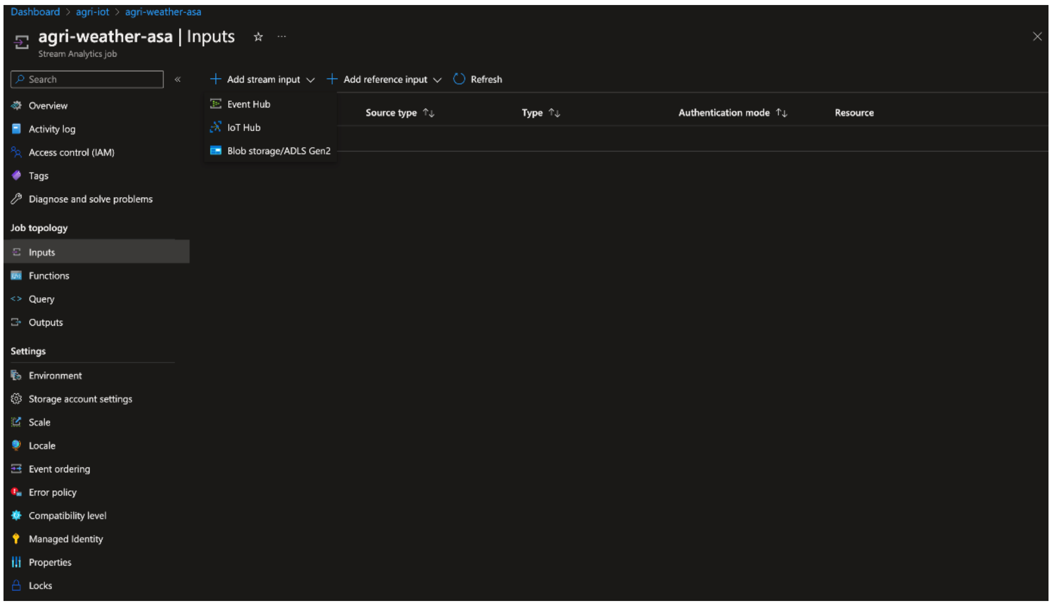
Authentication mode *: Connection string

The connection string authentication mode will use the storage account key.

Review + create | < Previous | Next : Tags >

Figure 19: Create Stream Analytics Job Storage Section Screenshot

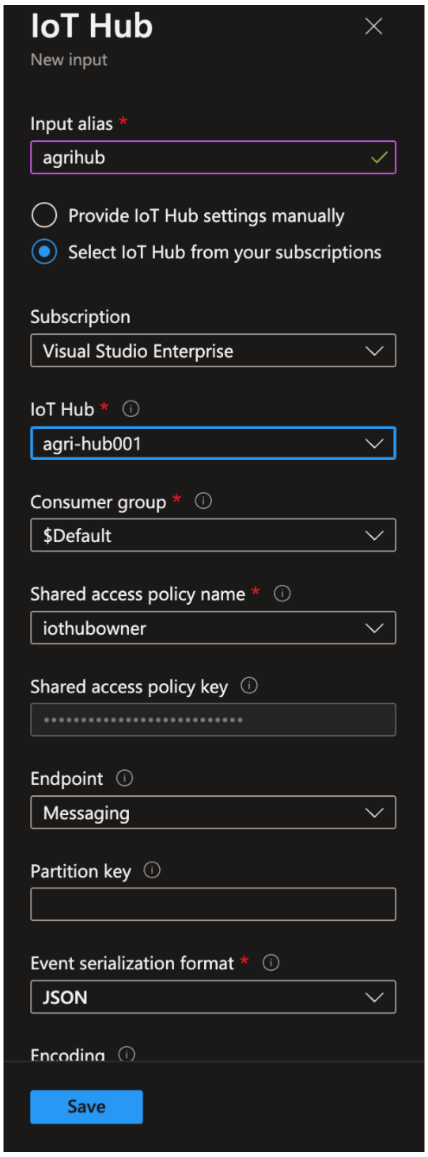
After the Stream Analytics job was created, I created the input source for the Stream Analytics job by selecting the source as IoT Hub as shown in Figure 20.


Dashboard > agri-iot > agri-weather-asa

agri-weather-asa | Inputs
Stream Analytics job

Figure 20: Screenshot of Input creation in Azure Stream Analytics

I filled in the details of the IoT hub as shown in Figure 21 and clicked "Save".



IoT Hub
New input

Input alias *
agrihub

( ) Provide IoT Hub settings manually
(●) Select IoT Hub from your subscriptions

Subscription
Visual Studio Enterprise

IoT Hub *
agri-hub001

Consumer group *
$Default

Shared access policy name *
iothubowner

Shared access policy key
••••••••••••••••••••••••••

Endpoint
Messaging

Partition key

Event serialization format *
JSON

Encoding

Figure 21: Screenshot of Add Input Blade

Then I went to "Outputs" blade and selected Table Storage as the output option as shown in Figure 22.

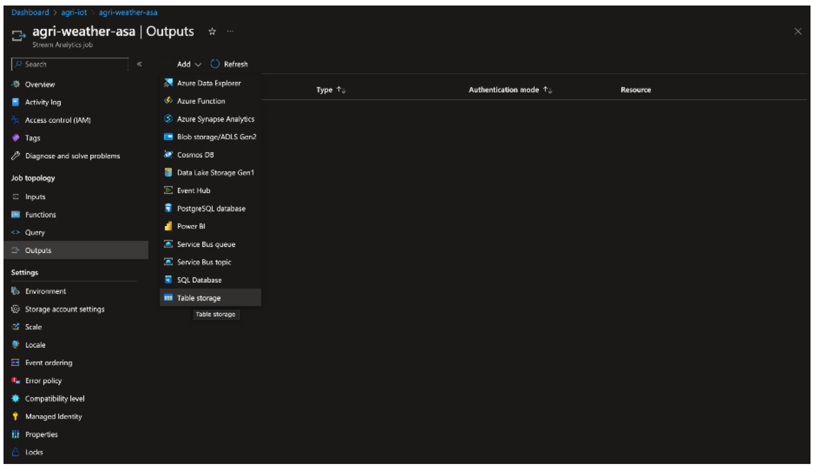
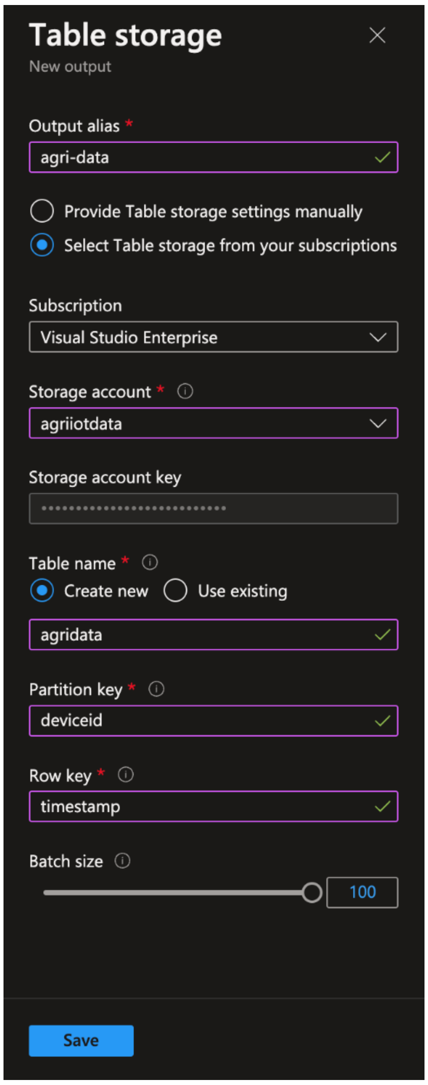


Figure 22: Screenshot of the Outputs Blade

In the table storage blade I filled the storage account and a table name where the details will be saved as shown in Figure 23.



Output alias * agri-data

Provide Table storage settings manually [✓ Select Table storage from your subscriptions]

Subscription: Visual Studio Enterprise

Storage account * agriiotdata

Storage account key ••••••••••••••••••••••••••••

Table name * [✓ Create new] Use existing

agridata

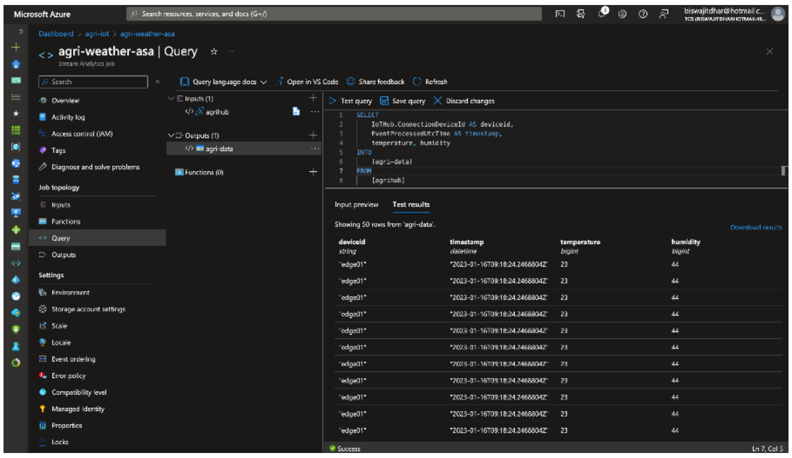
Partition key * deviceid

Row key * timestamp

Batch size 100

Figure 23: Screenshot of Add Table Storage Blade

Next, I selected the *Queries* blade and wrote a simple query to transform the data to a format suitable for table storage and tested the query. Figure 24 shows the query along with test output.



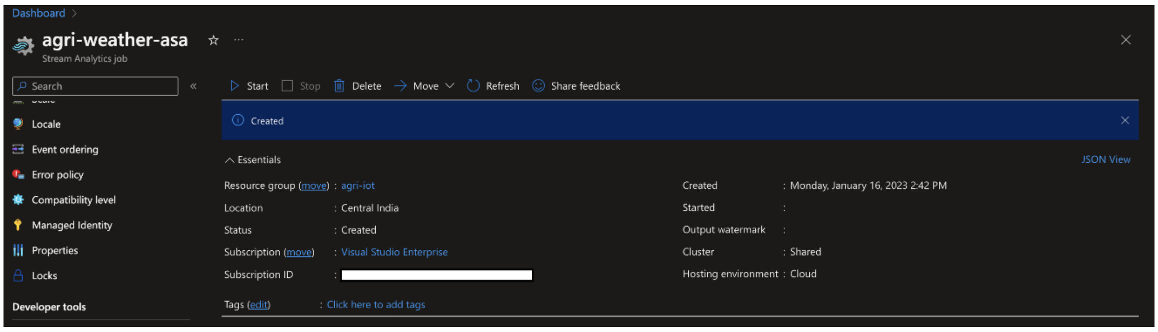
```
SELECT
    IoTHub.ConnectionDeviceId AS deviceid,
    EventProcessedUtcTime AS timestamp,
    temperature, humidity
INTO
    [agri-data]
FROM
    [agrihub]
```

Showing 50 rows from 'agri-data'.

deviceid *string*	timestamp *datetime*	temperature *bigint*	humidity *bigint*
"edge01"	"2023-01-16T09:18:24.2468804Z"	23	44
"edge01"	"2023-01-16T09:18:24.2468804Z"	23	44
"edge01"	"2023-01-16T09:18:24.2468804Z"	23	44
"edge01"	"2023-01-16T09:18:24.2468804Z"	23	44
"edge01"	"2023-01-16T09:18:24.2468804Z"	23	44
"edge01"	"2023-01-16T09:18:24.2468804Z"	23	44
"edge01"	"2023-01-16T09:18:24.2468804Z"	23	44
"edge01"	"2023-01-16T09:18:24.2468804Z"	23	44
"edge01"	"2023-01-16T09:18:24.2468804Z"	23	44
"edge01"	"2023-01-16T09:18:24.2468804Z"	23	44
"edge01"	"2023-01-16T09:18:24.2468804Z"	23	44

Figure 24: Screenshot of the Queries Blade along with the Query

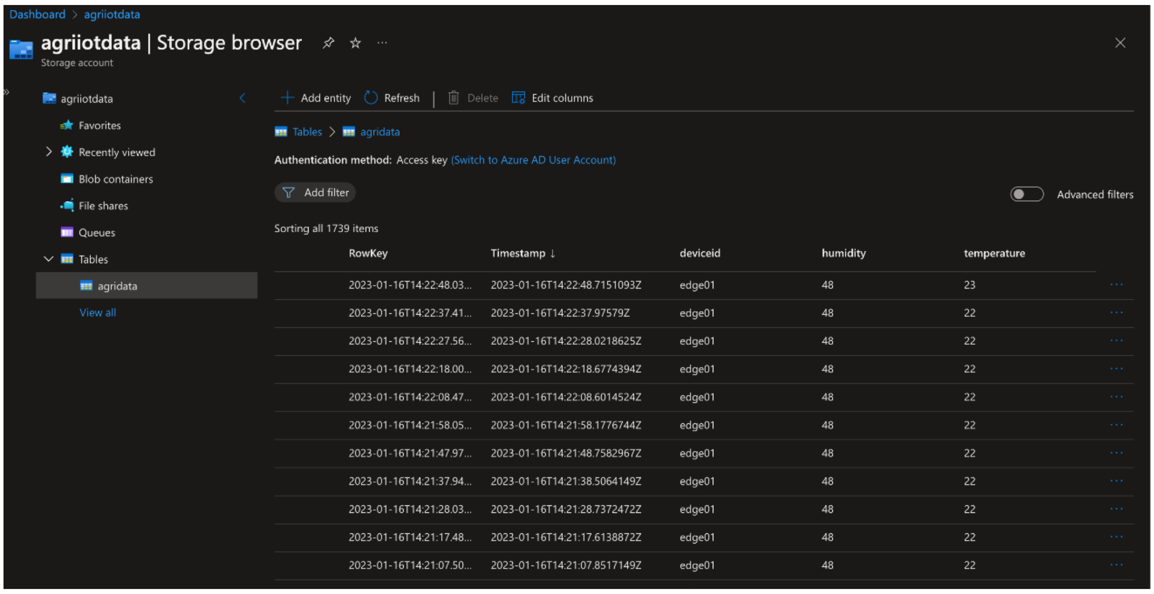
After this, I saved the query and started the job by clicking the “Start” button as shown in Figure 25.



Resource group (move) : agri-iot
Location : Central India
Status : Created
Subscription (move) : Visual Studio Enterprise
Subscription ID : [redacted]
Tags (edit) : Click here to add tags
Created : Monday, January 16, 2023 2:42 PM
Started :
Output watermark :
Cluster : Shared
Hosting environment : Cloud

Figure 25: Screenshot of the Stream Analytics Landing Page with Start Button

After some time, I opened the table storage in Storage Explorer and saw the data as shown in Figure 26.



Authentication method: Access key (Switch to Azure AD User Account)

Sorting all 1739 items

RowKey	Timestamp ↓	deviceid	humidity	temperature
2023-01-16T14:22:48.03...	2023-01-16T14:22:48.7151093Z	edge01	48	23
2023-01-16T14:22:37.41...	2023-01-16T14:22:37.97579Z	edge01	48	22
2023-01-16T14:22:27.56...	2023-01-16T14:22:28.0218625Z	edge01	48	22
2023-01-16T14:22:18.00...	2023-01-16T14:22:18.6774394Z	edge01	48	22
2023-01-16T14:22:08.47...	2023-01-16T14:22:08.6014524Z	edge01	48	22
2023-01-16T14:21:58.05...	2023-01-16T14:21:58.1776744Z	edge01	48	22
2023-01-16T14:21:47.97...	2023-01-16T14:21:48.7582967Z	edge01	48	22
2023-01-16T14:21:37.94...	2023-01-16T14:21:38.5064149Z	edge01	48	22
2023-01-16T14:21:28.03...	2023-01-16T14:21:28.7372472Z	edge01	48	22
2023-01-16T14:21:17.48...	2023-01-16T14:21:17.6138872Z	edge01	48	22
2023-01-16T14:21:07.50...	2023-01-16T14:21:07.8517149Z	edge01	48	22

Figure 26: Screenshot of the data in Table Storage

Challenges in Implementing the Solution

The biggest challenge I faced while implementing the solution was in procuring the sensors and boards from retail market. The next challenge was to access the GPIO pins from inside containers. This was resolved with the documentation from Microsoft and the discussions in the Microsoft Learn Portal forums and Stack Overflow.

Business Benefit

Crop protection is a major challenge for the farmers as they don’t get timely warnings so that they can prevent crop damage. Solutions like these allow government organizations to utilize the power of Azure cloud to manage and deploy micro-weather stations at community level and gather local weather data, perform analysis and warn the local community of probable pest attacks. This information can then be disseminated from panchayat level to the farmers so that they can take adequate protection.

Written By: **Biswajit Dhar (TATA Consultancy Services Ltd.)**

GitHub Link: https://github.com/bdhar1/agri-iot-edge