Cloud Temperature & Humidity Notification System

Lue Xiong

April 30, 2020

Contents

1 Context

The Cloud Temperature & Humidity Notification System is about an IoT system that gives the ability to notify users of temperature and humidity fluctuations within their living environment through the usage of a Simple Message Service, which is also known as SMS. The system will also notify a user if the Particle Argon is offline by checking that data is being sent and stored in the database. Though this is the main concern of the system, it also allows users to visualize their daily climate averages in an interactive graph. The graph is automatically updated for the users to view whenever they want to on the web. Behind the scenes, most computation is abstracted away from the user by using the Google Cloud Platform. The initiation of these functionalities start from the ambient temperature and relative humidity readings being published by the Particle Argon.

This system represents a single IoT device that performs the above-mentioned functionalities. One can imagine, however, being able to send in-home area location data and climate readings with a multiple of these IoT devices scattered across a home with multiple rooms and stories. A user would be notified where in the house and when the climate has reached configured threshold levels for each device. They would then be able to see data points for each specified area of the home. Ideally, aspects of the system such as climate data publishing intervals, setting climate threshold and maximum notification intervals, and timezones should be configurable. With that said, the project seeks a minimum viable product. That is, the system will notify users of climate thresholds being met and device status through SMS notification as well as graphing average climate per day for viewing averages over days. User configuration will be beyond the minimum viable product as it would require additional development of a mobile or web application.

Though the system is targeted for the home living environment, it can be used for environments that require careful monitoring. Take a fermented product like kombucha for instance; it needs to be fermented in an environment where the ambient temperature hovers in the range of 65 to 85 degrees Fahrenheit over a week to multiple weeks. Low temperatures will either stop or dramatically slow down the fermentation process. High temperatures will quicken the fermentation process but also increases the risk of unwanted bacteria or mold growth that would ruin the product. Striking a balance with temperature for optimal conditions is difficult without information. The usage is only limited to the imagination.

The Cloud Temperature & Humidity Notification System source code can be found in GitHub linked here:

`https://github.com/lxiong1/cloud-temperature-humidity-notification`.

2   Problem

The problem trying to be solved is giving the user climate data about their living environment. Certain ranges of temperature and humidity affect humans in ways that are detrimental to their health. For example, low humidity environments –characterized as 30% relative humidity and below – is a condition for being prone to respiratory infections, dry eyes, and itchy irritated skin. High humidity environments – characterized as 60% and above – is a condition for bacterial and mold growth, a catalyst for the decomposition of organic materials, and the attraction of bugs and insects to the home. A way to solve that problem is to give a renter or homeowner actionable data to understand that their living environment needs change. That is where the Cloud Temperature & Humidity Notification System comes in.

3   Market Research

From market researching, there are a couple of products that have similar functionalities:

- AcuRite 01166M 3-Sensor Indoor Monitoring

- Govee Temperature Humidity Monitor

- Proteus AMBIO

All three of these products measure temperature and humidity and have their value propositions. We'll take a look at what they do and how they differ from this system.

3.1   AcuRite 01166M 3-Sensor Indoor Monitoring

AcuRite 1166M is packaged with 3 sensor units along with what they call a *smartHUB*. The name is self-explanatory, it is the central communication piece for the 3 sensors. The max connection capacity for the smartHUB is 10 sensors. These sensors will read in the climate information of the environment and send it over to the smartHUB, which will then send information over the network to be processed and viewed online. There are some inherent flaws with this system as I will explain.

With the max capacity being 10 sensors connected to the smartHUB, it is only usable in a home environment and even then, a user may want more sensors if they have a larger home. The reason for the limitation is the design of having sensors centered around the smartHUB, and not standalone pieces that can interact with a network. The sensors are nothing without the smartHUB, therefore renders the system useless if it breaks. It also adds one more layer of complexity that is unnecessary for the user. Unnecessary because it is not

doing anything complex enough to justify a central hub.

Another issue is reliability. The range at which these sensors can communicate with the smartHUB is limited and depending on the structure, materials, and size of the home can make it more difficult to reliably transfer information from sensor to hub. This issue of reliability is further enforced by users of the system.

3.2   Govee Temperature Humidity Monitor

3.3   Proteus AMBIO

4   System Overview

This section will describe the minimum viable product requirements and the architectural decision made for achieving the desired results of the system.

4.1   System Requirements

To satisfy the need of giving a user useful information about temperature and humidity in their targeted environment, the following are the minimum viable product requirements:

- All notifications sent to the user shall be an SMS message

- Notifications sent to the user shall be based on the following criteria

– Temperature upper threshold shall be set to lesser than or equal to 75 degrees Fahrenheit
– Temperature lower threshold shall be set to greater than or equal to 65 degrees Fahrenheit
– Humidity upper threshold shall be set to lesser than or equal to 60 percent relative humidity
– Humidity lower threshold shall be set to greater than or equal to 30 percent relative humidity

- Minimum 1-hour gap in between each SMS message per climate type (AKA temperature and humidity) reaching above or below the threshold value

- The status of the device publishing temperature and humidity data must be checked every hour

– When the device is online, nothing shall happen
– When the device is offline, the user shall be notified

- Working hours shall be defined as 6 AM to 9 PM CDT

- Device shall publish every 15 minutes during the working hours

- Device shall publish every 30 minutes during the non-working hours

- System shall create a daily graph aggregating temperature and humidity averages over days at 9 PM CDT

4.2 Event-Driven Serverless Architecture

The approach taken for the system as mentioned before is heavy on the software side and manifested into an event-driven serverless architecture. An event-driven serverless architecture in the system context allows the climate data readings to be sent as events, processed based on its event type, and have actions performed without the need for standing up servers. The following are benefits of the said architecture:

- Loose coupling between software components

- Automatic scaling based on user demand

- All processing is driven by events

- No need for management of servers

- Overall cost reduction versus self-managed servers

- Inherent stateless nature

While these benefits are good to have, every architectural decision has its drawbacks. The following are drawbacks of the said architecture:

- Each serverless function must provide its own packaged dependencies

- Having a large number of serverless functions can become unwieldy

- Long-running processes are not fit for serverless functions

- Handling system state amongst stateless functions can be tricky

Using an event-driven serverless architecture made sense for this system, as the needs of the system are fulfilled automatically on publishing data without any manual intervention.

5 Process and Implementation

I will preface that the Cloud Temperature & Humidity Notification System is heavily software-based. The hardware components are nothing special and ultimately serve as a vessel for sending climate information to be processed in the cloud to create the closed-feedback loop. The Particle Argon referenced in Section 5.1 will contain firmware that is fundamental to the system requirements and architectural decision in Section 4. This particular section will describe the thought process behind the implementation, considerations taken during development, and how the system performs user notification and automatic graph updates.
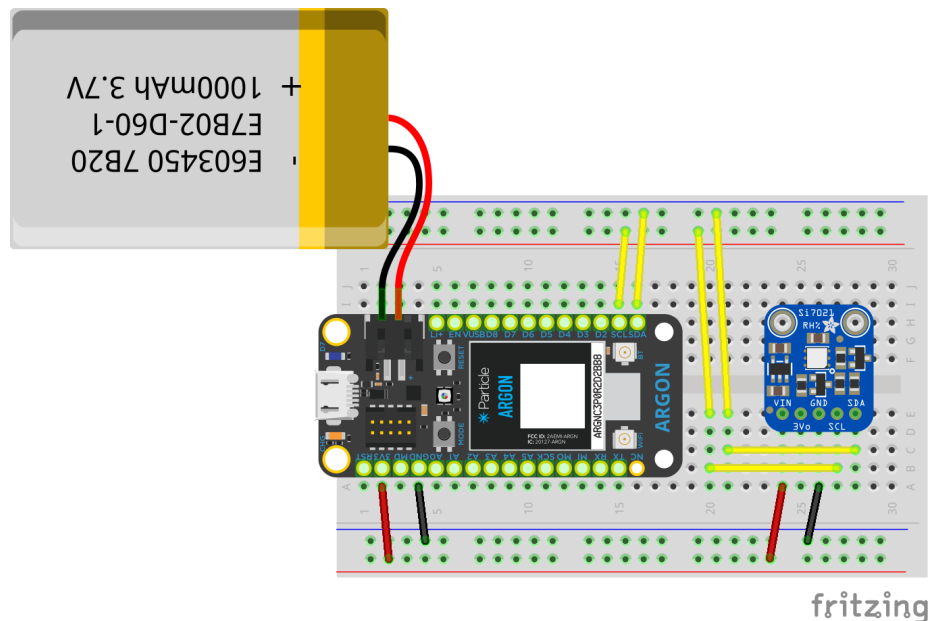
5.1 Hardware Components

Figure 1: Breadboard Schematic

Figure 1 is the bread schematic showing how all of the hardware components are connected. This enables the Particle Argon to read and publish climate data events. With the battery, it makes the setup standalone and charged only when needed. Below is a brief description of each component.

Particle Argon — An IoT development kit for connecting to and sending information over the network.

Adafruit Si7021 — A temperature and humidity sensor breakout board that provides the ability to read a given environment's climate.

Breadboard — A physical base that provides the ability to create circuits and prototype electronics.

Jumper Wires — Cables that can electrically connect components.

Lithium Ion Polymer Battery 1000 mAh — A power source for the connected components.

5.2 Particle Argon and Particle Cloud

Particle Argon is flashed with firmware written with C code. It is used with Particle's API and the Adafruit Si7021 helper library. Understanding the Particle ecosystem's role in the system will reveal the scope of its interactions in the event-driven serverless architectural decision.

```
void setup () {
  Time.zone(-5);
  sensor.begin ();
}
```

Figure 2: setup() function

The **setup()** as shown in Figure 2 is important here for two reasons. The first part is that the Particle Argon is in default timezone of Universal Time Coordinated (UTC) which differs from the timezone I live in is Central Daylight Time (CDT). To be able to represent the timezone difference, setting the default timezone with an offset of minus 5 hours is required. The second part is making sure that the Adafruit Si7021 is enabled after being initialized because it is the central piece for reading ambient temperature and relative humidity from a given environment.

```
void loop(void) {
  int degreesCelsius = sensor.readTemperature();
  int degreesFahrenheit = (degreesCelsius * 9 / 5) + 32;
  int relativeHumidity = sensor.readHumidity();

  Particle.publish("temperature", String(degreesFahrenheit),
      PRIVATE);
  Particle.publish("humidity", String(relativeHumidity), PRIVATE);

  if (isEndOfDay() == true) {
    Particle.publish("climateAverageUpdate", "Updating␣climate␣
        data␣file", PRIVATE);
    publishIntervalMilliseconds = 1800000;
  } else {
    publishIntervalMilliseconds = 900000;
  }

  delay(5000);

  System.sleep(systemSleepConfiguration
    .gpio(WKP, RISING)
    .network(NETWORK_INTERFACE_CELLULAR)
    .flag(SystemSleepFlag::WAIT_CLOUD)
    .mode(SystemSleepMode::STOP)
    .duration(publishIntervalMilliseconds));
}
```

Figure 3: loop() function

In Figure 3, the **loop()** function shows that temperature and humidity are being read with the sensor object. They are then used as values to publish to the Particle Cloud. From that point on they are passed to Google Cloud Platform. However, there will be a further explanation of how this integration works and what processes happen on the Google Cloud Platform in Section 5.3.

```
bool isEndOfDay() {
  int now = Time.now();
  int currentHour = Time.hour(now);
  int isMorning = Time.isAM(now);

  if (updated == false && currentHour >= 21) {
    updated = true;
    return true;
  }

  if (updated == true && currentHour >= 6 && isMorning) {
    updated = false;
    return false;
  }

  return false;
}
```

Figure 4: isEndOfDay() function

Beyond publishing the climate values, the **loop()** function checks whether the day has ended from the perspective of the system. The end of the day is defined as code in Figure 4. Before it was mentioned that a UTC offset of minus 5 hours was applied in the **setup()** function, and this is where it comes into context. The end of the day is 9 PM CDT, which is equivalent to 21 in a 24-hour format. Once the system understands that it is the end of the day, an event is published to update a file containing daily climate average aggregates for the day. To understand the update itself will require digging into the Google Cloud Platform integration. As written before, this will be further explained in Section 5.3.3.

Concerning the end of the day as well as a variable called **publishIntervalMilliseconds** will be set to 1800000, in other words, 30 minutes. This publishing interval will last for 9 hours starting from 9 PM CDT to 6 AM CDT. During the working day, the **publishIntervalMilliseconds** is set to 900000, or 15 minutes. This plays a role in conserving battery because, during the night, it is only publishing climate data two per hour as opposed to four. However, the bigger battery-saving impact comes from placing the Particle Argon in a sleeping state.

A sleeping state means that the Particle Argon turns off all unnecessary internal processes that consume energy and maintains a low-power state for a duration of a configured time. There are two main modes of sleep state that Particle Argon can go into: **STOP** and **HIBERNATE**. The decision was to use the **STOP** sleeping mode because it provides the opportunity to keep the network connection alive while it is in a low-power state. **HIBERNATE** does not allow this, and if it was used, it would require the Particle Argon to completely establish a new connection every time it wakes up from the specified duration of **publishIntervalMilliseconds**; defeating the point of using it.

5.3 Google Cloud Platform Services

Google Cloud Platform is the most important part of the whole system. That being said, it is also important to give a brief description of each service used within the platform and why.

5.3.1 Compute Engine

Compute Engine is a virtual machine service that provides user-configured instances. This is as opposed to App Engine that is a fully-managed by Google Cloud Platform. The instance will be used to host a web server. With that, the system will have the ability to deliver a static website containing an interactive graph. Figure 5 shows ambient temperature and relative humidity daily averages that are data points on the interactive graph. The user will be able to see the changes over time and adjust the climate in their environment according to the purpose of what they are trying to achieve. It can be accessed in the link here: `http://35.238.110.48:8080`.

Figure 5: Interactive Graph of Climate Data

Previously it was mentioned that a web server was used to deliver the interactive graph, but the specifics of how that was accomplished were not explained. On a high level, it is a combination of using Python and several software components including Plotly, Docker, NGINX, and a separate project repository. Below will explain what each of these software components is and the role they play in the automatic creation and delivery of the graph to the web.

Plotly — A Python library to help develop custom graphs. It offers many graph types and visual options. To show climate data in a visually simple way, a line graph was used to indicate changing ambient temperature and relative

humidity over time.

Docker — An OS-level containerization tool that can package software. Docker has a registry called Docker Hub full of container images created by the open-source community. These container images can be pulled down to a local machine and be used on-demand and/or developed on top of. These containers can be thought of as software building blocks to assist in software development.

NGINX — A web server that provides communication between client and server via HTTP. NGINX has container images stored in Docker Hub that is publicly accessible, which in this context, is used for standing up a web server in Compute Engine.

Climate Data Graph Updater — Updates the interactive graph on a daily basis when the **update_climate_data_graph_in_engine()** trigger function is initiated, referenced in Section 5.3.3. This sits in a different GitHub repository from the Cloud Temperature & Humidity Notification repository. The decision to isolate this portion of functionality was due to the lower operational cost of pulling it down from GitHub from the Compute Engine instance. It also separates the concern from the system perspective. The GitHub repository is linked here: `https://github.com/lxiong1/climate-data-graph-updater`.

5.3.2   Firestore

Firestore is a NoSQL document database that provides data persistence across the system. Specifically, this database is used for storing three major collections that assist in the state of the system. Figure 6, 7, and 8 shows the different root collections and their respective documents.

Figure 6: Contact Attempt History Document

Figure 7: Relative Humidity Document

Figure 8: Ambient Temperture Document

As you can see, the root collections contain multiple documents with fields that contain data values. Each time a climate data or contact attempt event comes through the system, a new document is created for each of those events to ensure that the system understands what historically has occurred.

5.3.3 Functions

Functions is an event-driven serverless compute service. This is the heart and soul of the Cloud Temperature & Humidity Notification System. This is where most of the computation happens. As outlined in Section 4.2, there are major advantages of using a service like Functions that align with being event-driven serverless. Figure 9 shows the existing serverless functions of the system.

Figure 9: Google Cloud Functions

A convenient feature about Functions is that it allows engineers to test all code written to be tested from anywhere. An engineer can deploy a serverless function to be deployed and then tested in a controlled environment or from a local

machine. There simply is no limitation to where these serverless functions run. From first-hand experience, being able to test locally helps reduce unnecessary deploy time, which can take up to a couple of minutes and even up to a minute afterward to properly register.

It's important to point out that Section 5.3.4 and this section are tightly related. Refer to Figure 15 to see the association of trigger functions to their respective PubSub topics. Topics will relay events (AKA messages) to these trigger functions to perform some action. What this action encompasses depends on the event.
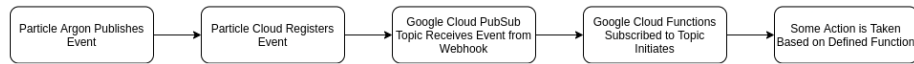

Figure 10: Generic System Event Flow

Figure 10 shows the general event flow starting from the Particle Argon to the serverless function. The specifics of what happens after an event type makes this process more complex. To better understand the specifics of the system, take a look at Figure 11. Though it is specifically referring to temperature events, humidity events have the same flow throughout the system.
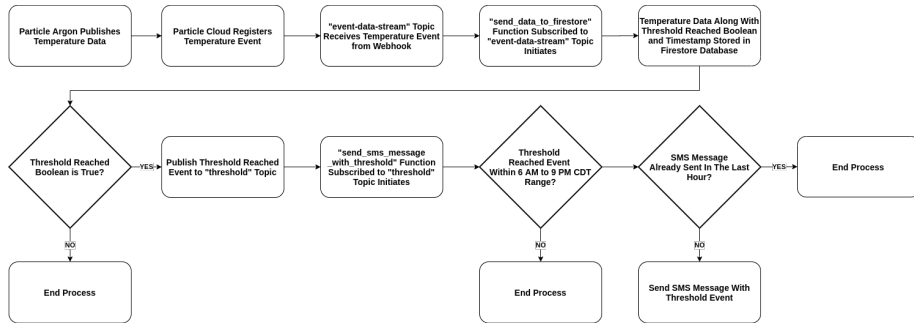

Figure 11: Temperature Event Flow Within System

Figure 11 demonstrates how an SMS message is determined to be sent to a user. In the event flow, there is what is called **Trigger Function Chaining**. A function can send a message to another topic which then initiates a different function, thus why it is referred to as chaining. The functions themselves hold absolutely no state of the system, it searches in the database to understand the state. Notice how the **Threshold Reached** boolean value was stored into the database to determine whether there should be a potential SMS message to be sent out. That is the database storing the state, not the function itself. This separation of concern is crucial to properly build out an event-driven serverless system.
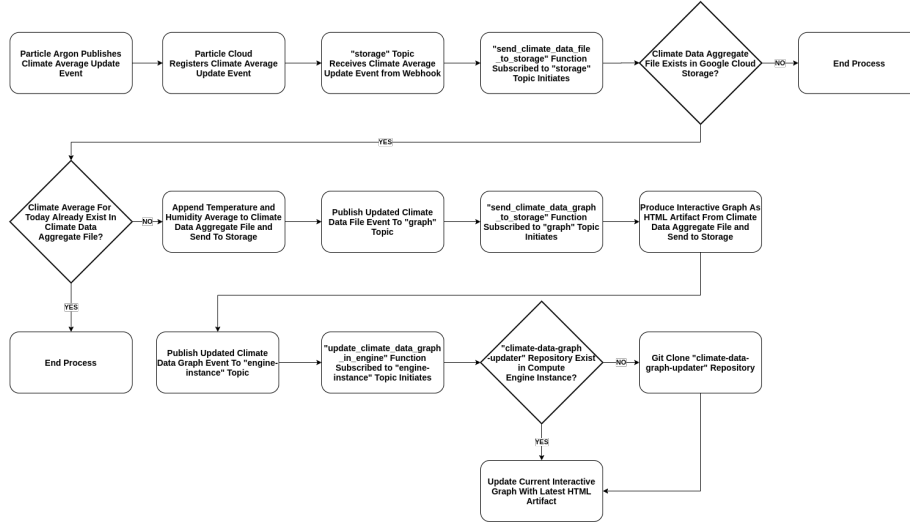
14

Figure 12: Climate Average Update Event Flow Within System

To update the interactive graph `http://35.238.110.48:8080`, Figure 12 demonstrates the processing that has to occur. It was mentioned before in Section 5.2 that the prerequisite for this update to begin, is when it is the end of the day, namely, 9 PM CDT. This process uses Trigger Function Chaining just as does when a temperature or humidity event is sent. The last functionality of the system is checking for the device status of the Particle Argon.
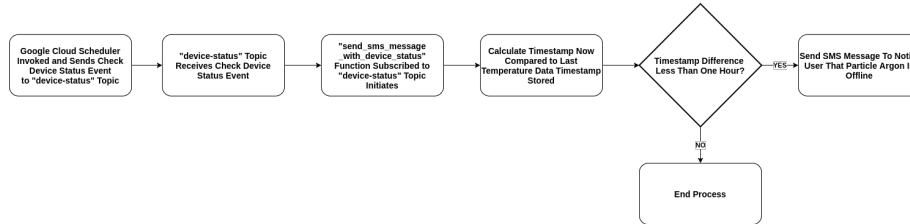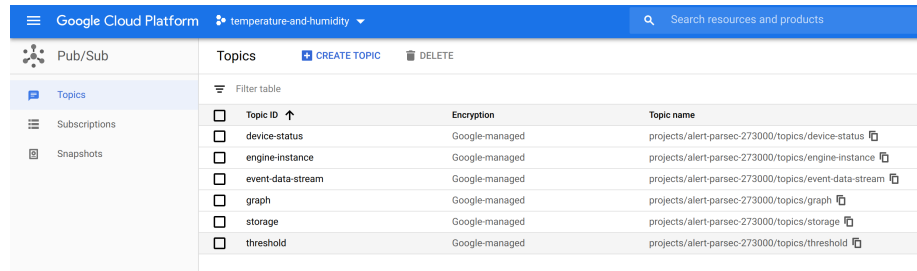

Figure 13: Device Status Event Flow Within System

This is where the Google Cloud Scheduler explained in Section 5.3.5 is used. Every hour of the working day – defined as 6 AM to 9 PM CDT, the Scheduler will send a message to the "device-status" topic and check whether the Particle Argon is offline or online. Figure 13 goes into detail of what that event flow looks like.

5.3.4 PubSub

PubSub is an asynchronous middleware messaging service. The base for this service revolves around three major concepts of topics, subscriptions, and messages. Topics are where messages are sent to when published. Subscriptions are the message retrievers when a message is published to the topic they are associated with. Messages are are the data published to topics and streamed to subscribers.

Figure 14: Google Cloud PubSub Topics

The advantage of using PubSub is that messages can be sent from anywhere and at any scale. There is no need to manage to scale as PubSub enacts the scaling needs automatically. This is a part of what makes the system event-driven serverless. It consumes messages and passes the information along as directed without standing up and configuring any server. It just works.

In the system context, messages are the events published by Particle Argon. Figure 14 shows all of the existing topics that support events processing. These topics are the first point of contact with the Particle ecosystem, starting from the Particle Argon publishing its events to the Particle Cloud and those events being webhooked into the PubSub topics. Every time an event is published to a topic, a subscribed trigger function initiates. Figure 15 is a mapping of topics and their subscribed trigger functions.

```
    TRIGGER_FUNCTION_NAME: send_data_to_firestore
    TOPIC_NAME: event-data-stream

    TRIGGER_FUNCTION_NAME: send_sms_message_with_threshold
    TOPIC_NAME: threshold

    TRIGGER_FUNCTION_NAME: send_climate_data_file_to_storage
    TOPIC_NAME: storage

    TRIGGER_FUNCTION_NAME: send_climate_data_graph_to_storage
    TOPIC_NAME: graph

    TRIGGER_FUNCTION_NAME: update_climate_data_graph_in_engine
    TOPIC_NAME: engine-instance

    TRIGGER_FUNCTION_NAME: send_sms_message_with_device_status
    TOPIC_NAME: device-status
```
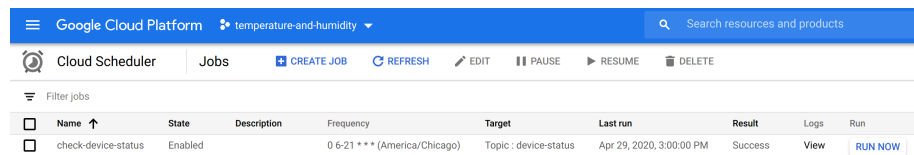
Figure 15: Mapping of Subscribed Trigger Functions to PubSub Topics

5.3.5 Scheduler

Scheduler is Cron job scheduler fully-managed by Google Cloud Platform. It would be wise to explain what Cron because it is not common knowledge. To put it simply, Cron is a Unix utility for scheduling arbitrary processes to run at arbitrary times. For example, Figure 16 shows that the system defines **0 6-21 * * * (America/Chicago)** under the frequency column. This is UNIX-Cron format for time-based scheduling and translates to, "every day every hour from 6 AM to 9 PM CDT."

Figure 16: Google Cloud Scheduler

What exact process is running for the system? As the name **check-device-status** from Figure 16 suggests, it sends an event to a PubSub topic called **device-status** that ultimately checks whether the device status is online or offline. For more information on the details of how that works, refer to Section 5.3.3.

5.3.6 Identity and Access Management

Identity and Access Management (IAM) is a service that provides the ability to manage roles and permissions on Google Cloud Platform resources. This is extremely important to manage correctly as giving people more access than

17

they need can potentially compromise a system entirely. Google Cloud Platform takes the approach of a zero-trust security model, so existing resources have to be given appropriate roles and permissions by the administrator.

5.3.7 Secret Manager

Secret Manager provides security around creating, storing, accessing, versioning, and destroying secrets. Many companies struggle with managing their secrets and often overlook the need for security for the sake of convenience. Although Secret Manager makes it easier to do this, it can be incredibly complex depending on the security needs of an organization.
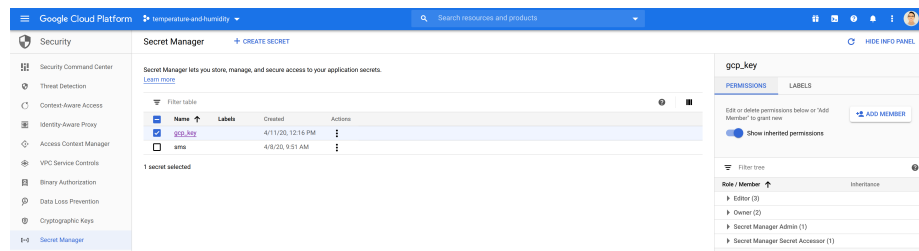
Figure 17: Google Cloud Secret Manager

There are currently two secrets stored as shown in Figure 17. They are automatically retrieved for usage when certain events come through the system. The **gcp_key** secret is used to Secure Shell (SSH) into the Compute Engine instance to update the interactive graph from a trigger function, explained in Section 5.3.3. The **sms** secret is used for authenticating to an email account used solely to send out SMS messages, also explained the Section 5.3.3.

5.3.8 Storage

Storage is a web file storage service. To store files in the service, users have to create what Google Cloud Platform calls a **Bucket**. A Bucket is just a way to categorize content, comparable to the way a Linux filesystem is structured with directory and files. For the system, one bucket was created and called **climate-data-files**. In it contains two files that are foundational to auto-updating the interactive graph referenced in Section 5.3.1.

Figure 18: Google Cloud Storage

The first file is called **climate_data_aggregate.csv** and its contents are comma-delimited average temperature and humidity data values as well as the date. Refer to Figure 19 for a 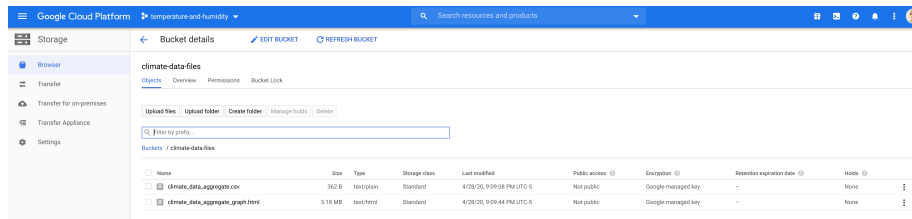visual. It is the base for updating the second file called **climate_data_aggregate_graph.html**. The HTML file ultimately ends up being used for updating the interactive graph.

```
date,temperatureAverage,humidityAverage
2020-04-07,70,35
2020-04-08,66,33
2020-04-09,67,41
2020-04-10,71,42
2020-04-11,67,40
2020-04-12,68,39
2020-04-13,64,42
2020-04-14,65,45
2020-04-15,69,40
2020-04-16,65,44
2020-04-18,66,33
2020-04-19,66,32
2020-04-20,67,38
2020-04-22,67,32
2020-04-23,67,36
2020-04-25,70,32
2020-04-26,69,30
2020-04-27,71,38
2020-04-28,72,37
```

Figure 19: Average Climate Data from Aggregate CSV file

5.4   GitHub Actions

With all of the development effort that has gone into building out the system, there was a need to simplify the deployment process. This process usually involved manually typing long commands in the terminal repetitively as the system iterations happened. The rule of thumb for software engineers is this: if you have to do it more than once, just automate it. This is where GitHub Actions comes into play.

Self-explanatory in its name of who the creator is, GitHub Actions is a platform

19

created to automate workflows. In essence, it is a Continuous Integration (CI) and Continuous Delivery/Deployment (CD) tool for software engineers that want to simplify and reduce repetitive tasks. GitHub Actions has helped remove the repetitive tasks that involve deploying the source code into production.

```
name: master
env:
  RUNTIME: python37
on:
  push:
    branches:
      - master
    # paths:
    #   - 'functions/**'
```

Figure 20: Part-one master-workflow.yml

Figure 20 shows the **master-workflow.yml** containing the instructions to run on a push to the Master branch and only when functions directory's contents have been changed. Otherwise, the automated pipeline does not run. The reason for this is because the serverless functions are the only concern in terms of deployment.

```
jobs:
  deploy:
    name: Deploy Trigger Functions to Google Cloud Platform
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v2
      - name: Setup Python
        uses: actions/setup-python@v1
        with:
          python-version: 3.8.2
          architecture: x64
      - name: Run Python Formatter
        run: |
          pip install black
          black .
      - name: Run Python Linter
        run: |
          pip install flake8
          flake8 --filename=./functions/*py --ignore E501
      - name: Autheticate for Google Cloud Platform Access
        uses: GoogleCloudPlatform/github-actions/setup-
            gcloud@master
        with:
          project_id: ${{ secrets.GCLOUD_PROJECT_ID }}
          service_account_email: ${{ secrets.GCLOUD_EMAIL }}
          service_account_key: ${{ secrets.GCLOUD_AUTH }}
```

Figure 21: Part-two master-workflow.yml

Figure 21 shows the setup portion of the workflow that runs after the workflow trigger criteria is met as a result of the workflow defined in Figure 20. This will check out the source code, setup Python and run the Python code formatter as well as the code linter, and then finish up by gaining API access to Google Cloud Platform. The secrets that are needed to gain this access is stored within GitHub, and will only be retrieved at runtime for security purposes. At any point that any of these steps fail, it will stop the whole pipeline and throw an error.

The actual deployment is shown in Figure 22 is the key reason for the automated pipeline. Each of the serverless functions that exist within the functions directory will be deployed and associated with each of their respective Google PubSub topics. This portion takes the longest but since it is automated, it makes deployment much faster and easier. In summary, no more manual setup and deployments in the terminal.

```yaml
      - name: Deploy send_data_to_firestore Trigger Function
        run: cd functions/$TOPIC_NAME && gcloud functions deploy
            $TRIGGER_FUNCTION_NAME --trigger-topic $TOPIC_NAME --
            runtime ${{env.RUNTIME}}
        env:
          TRIGGER_FUNCTION_NAME: send_data_to_firestore
          TOPIC_NAME: event-data-stream
      - name: Deploy send_sms_message_with_threshold Trigger
          Function
        run: cd functions/$TOPIC_NAME && gcloud functions deploy
            $TRIGGER_FUNCTION_NAME --trigger-topic $TOPIC_NAME --
            runtime ${{env.RUNTIME}}
        env:
          TRIGGER_FUNCTION_NAME: send_sms_message_with_threshold
          TOPIC_NAME: threshold
      - name: Deploy send_climate_data_file_to_storage Trigger
          Function
        run: cd functions/$TOPIC_NAME && gcloud functions deploy
            $TRIGGER_FUNCTION_NAME --trigger-topic $TOPIC_NAME --
            runtime ${{env.RUNTIME}}
        env:
          TRIGGER_FUNCTION_NAME: send_climate_data_file_to_storage
          TOPIC_NAME: storage
      - name: Deploy send_climate_data_graph_to_storage Trigger
          Function
        run: cd functions/$TOPIC_NAME && gcloud functions deploy
            $TRIGGER_FUNCTION_NAME --trigger-topic $TOPIC_NAME --
            runtime ${{env.RUNTIME}}
        env:
          TRIGGER_FUNCTION_NAME:
              send_climate_data_graph_to_storage
          TOPIC_NAME: graph
      - name: Deploy update_climate_data_graph_in_engine Trigger
          Function
        run: cd functions/$TOPIC_NAME && gcloud functions deploy
            $TRIGGER_FUNCTION_NAME --trigger-topic $TOPIC_NAME --
            runtime ${{env.RUNTIME}}
        env:
          TRIGGER_FUNCTION_NAME:
              update_climate_data_graph_in_engine
          TOPIC_NAME: engine-instance
      - name: Deploy send_sms_message_with_device_status Trigger
          Function
        run: cd functions/$TOPIC_NAME && gcloud functions deploy
            $TRIGGER_FUNCTION_NAME --trigger-topic $TOPIC_NAME --
            runtime ${{env.RUNTIME}}
        env:
          TRIGGER_FUNCTION_NAME:
              send_sms_message_with_device_status
          TOPIC_NAME: device-status
```

Figure 22: Part-three master-workflow.yml

It is crucial to point out the inherent benefit of building an automated workflow on such a platform, that is, idempotency. Machines only run the instructions they have been given and nothing more. It means that an engineer can expect the same results whenever their pipeline runs. Figure 23 is a visual for engineers to see the automated pipeline step by step that ran after a push to the Master branch.
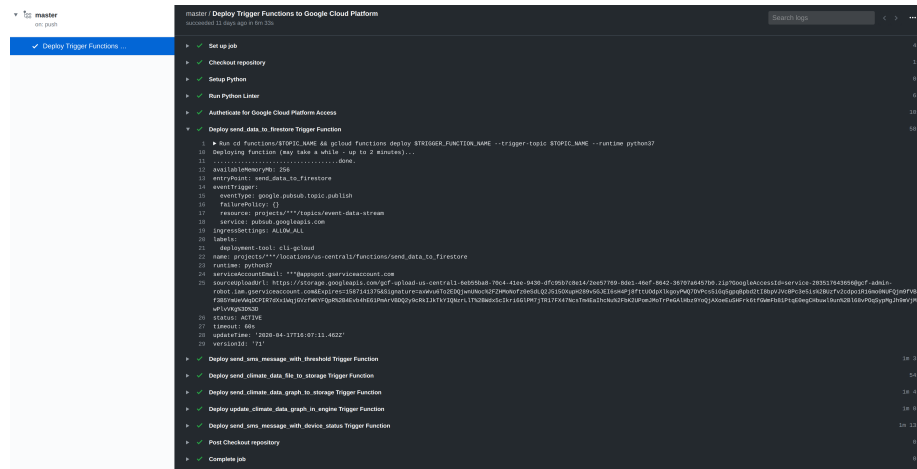

Figure 23: GitHub Actions Workflow Visual

6   Discussion

It was quite a difficult experience integrating all the hardware and technology stack used in the Cloud Temperature & Humidity Notification System. This was especially true of Google Cloud Platform because while there was extensive documentation on their services, Google is a company known for renaming, re-branding, and deploying new services often. As one can imagine, that makes it difficult to discern what was changed, what to look for, and what to use. Their documentation is not very well organized as concepts and subconcepts of a particular service are scattered throughout the website. Ironically enough, it was more efficient – though not exactly pleasant – to Google search for what I needed than navigating through the Google Cloud Platform documentation.

Aside from the obvious pitfall of the documentation, Google Cloud Platform is powerful. Google manages and reduces a lot of work that would otherwise be placed as responsibility for a software engineer. As engineers, it is only natural to selectively point out aspects of a system that are nuances but truly software is a collection of reused code that makes life easier for them. Take Google Cloud functions, for example, an engineer only has to write the code to handle

23

incoming events and Google manages the rest. The rest comprising of buying, maintaining and upgrading hardware to host servers for computational needs, developing security practices around the serverless functions, developing logging abilities on all existing functions, and the list just goes on.

As we are on the topic of serverless functions, I believe it is important to point out that they were the big driver behind much of the background processing necessary to create the system. A lot of the other cloud services played a role but I feel that Google Cloud Functions played the most prominent role. Serverless functions are relatively new to the software engineering field, which was popularized initially by Amazon Web Services when they released their Lambda functions in 2014. Since then it has gained traction and understandably so. Unlike the traditional approach of an always-on server – hence the word *serverless* in serverless functions – there is very little operational overhead to speak of. I believe that this is the way of the future for software and especially IoT.

The results of the Cloud Temperature & Humidity Notification System developed was possible due to the collective knowledge of many industries and intellectuals put together. That is something to be proud of and awe of when we realize how much knowledge was accumulated over time to get to this point. It is simply amazing.

7   Conclusion

Being able to read ambient temperature and relative humidity and send information about it can have beneficial implications when thoughtfully used. IoT – though as vague and broad as it is, possibly even misnomer – has large and impactful implications on humans whether; good or bad. In this case, mostly good because the system is simply reading the given environment's climate. The Cloud Temperature & Humidity Notification System only represents an incredibly small portion of the potential of IoT.