W205 Final Project
Title: Cloud-based IOT Server for Energy Savings

Team Members
Matthew Burke, Jan Forslow, Vyas Swaminathan, Xiao Wu

<u>Target Use Case</u>

The original idea for this project was to take the particular energy savings use case of Conservation Voltage Reduction (CVR) and show an architecture that can scale to hundreds of thousands of measuring points using AWS cloud as this is not possible today with current on-premise SQL-based RDBMS systems. The goal with CVR is to optimize the voltage level on the feeder line while not exposing customers at the end of the line to flickering (low voltage). This is achieved by using the SmartMeters as sensors and by sending control signals to regulators to adjust the voltage levels as shown in Figure 1.

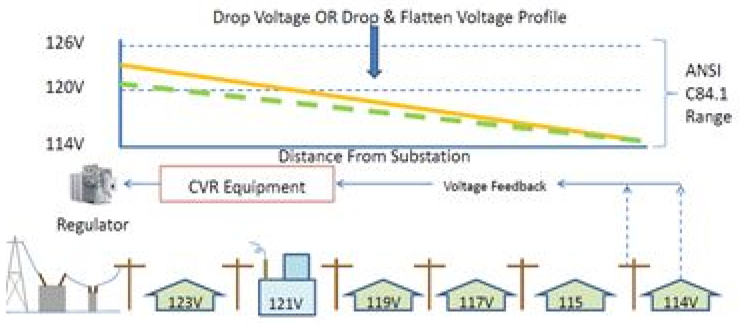


Figure 1. Example Feeder Line with CVR activated.

There are on average 300-600 households per feeder line of which ~ 40-50 SmartMeters will need to be used as sensors (also called bellwether meters). These bellwether meters will report back voltage levels every 5-10 minutes in normal setups. For this project, we are creating a simulator to simulate the streaming of this data. The simulator is sending messages containing voltage level of the individual bellwether meters on the Feeder Lines. We know from industry standard that the voltage on the feeder should not exceed 3% of 120V (123 – 117V). We limited this project to automatically correct the voltage level inside the simulator, but we have provided hooks for creating an automatic feedback loop to the device simulator from the IOT Server to readjust if the voltage level is below certain thresholds as well. Randomization is applied in the simulator to create voltage changes leading to adjustments.

<u>Architecture Review</u>

As a group, we discussed two possible architectures:

1. AWS IoT -> Kinesis -> ElasticSearch (and stateless AWS Lambda functions)

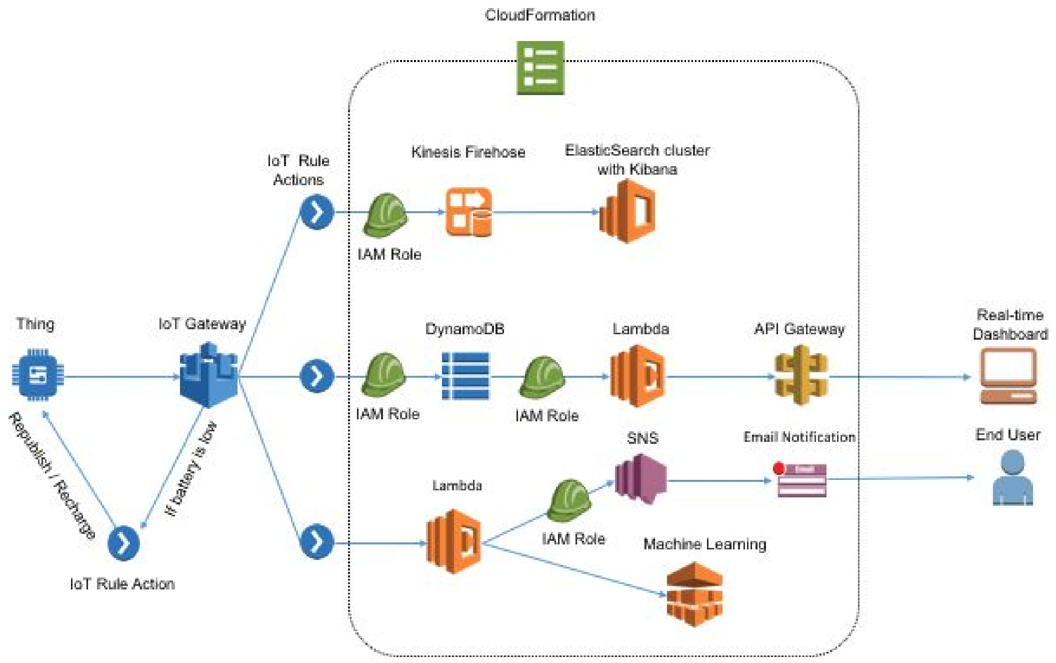


Figure 2. AWS IoT Reference Architecture.

As shown in Figure 2, the main components within the AWS IoT references architecture for near real-time analytics are:

- AWS IoT (Amazon Internet of Things) will collect and analyze data from internet-connected devices and sensors and connects that data to AWS cloud applications. It is an AWS service running independently of the EC2 instance we create.
- Amazon Kinesis Firehose is a platform for streaming data on AWS and offers services to make it easy to load and analyze streaming data. Firehose is optimized for write (load to destination) processing such as ElasticSearch.
- Amazon Elasticsearch is an open-source search and analytics engine for use cases such as data analytics, log analytics, real-time application monitoring, and clickstream analytics.
- Amazon DynamoDB is a flexible NoSQL database service for applications that need consistent, single-digit millisecond latency at any scale.
- Amazon API Gateway is a fully managed service to create, publish, maintain, monitor, and secure APIs at any scale.

- AWS Lambda allows to run code without provisioning or managing servers. It will run on any available AWS infrastructure and you pay as these code snippets are run only. Often used to connect different AWS building blocks together.
- Amazon Machine Learning is a service that provides visualization tools and wizards that guide through the process of creating machine learning (ML) models.
- Amazon Kinesis Streams enables custom applications that process or analyze streaming data for specialized needs. Kinesis Stream is more complex to configure than Kinesis Firehose, but Kinesis Stream allows for more customization.
- Amazon Simple Notification Service is a managed push notification service for sending individual messages or to fan-out messages to large numbers of recipients.

2. Kafka -> Storm -> MongoDB

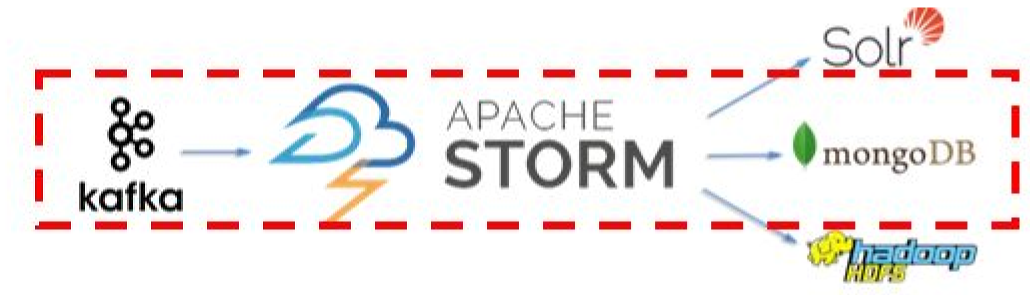


Figure 3. Apache Open-source Real-time Streaming Architecture.

- Kafka is a distributed streaming platform. It allows publishing, storing, and processing streams of records.
- Storm allows for reliably processing unbounded streams of data, doing real-time processing, and integrates well with Kafka.

For this project, we decided to use the first architecture, given the reasons for future commercialization:

1. With AWS IoT, applications can keep track of and communicate with all devices, all the time, even when they aren't connected.
2. Kinesis was modeled after Kafka, but in addition allows for zero-administration (not managing infrastructure).
3. Stateless Lambda statements automatically matches capacity to request rate, which allows purchasing compute capacity in 100ms increments.
4. Elasticsearch allows for flexible data visualization using in-built Kibana.

We use the top flow of Kinesis Firehose and Elasticsearch/Kibana to do real-time streaming analytics on the SmartMeter data coming in via the AWS IoT Gateway. We use a second data flow for creating a long-term data storage into DynamoDB. Finally, we use the lower Amazon ML data flow for prediction of anomalies and sending associated Email alerts.

<u>Design/Implementation</u>

1. Data Generation

For this project, we didn’t have access to publicly available SmartMeter data that was appropriate for us to create a CVR application with. The publicly available stream on data.sparkfun.com that we had found showed not to contain the voltage data that we needed. Instead we created a simulator in python.

The data generation itself is performed in two classes: `FeederLine` and `PowerGrid`.

`FeederLine` simulates a collection of numbered SmartMeters along a single power line, each with their own unique device ID and number along the `FeederLine.` An instance of a `FeederLine` has a few properties that allow the voltages for each device to be calculated and modified for CVR:

- `base_voltage` - average starting voltage at beginning of the line
- `base_variation` - standard deviation used in generation of true starting voltage
- `base_drop` - average drop in voltage between each SmartMeter in the line
- `bonus` - modifier used to modify starting voltage to perform CVR

As the `FeederLine` iterates through all of the SmartMeters, it calculates the voltage for the next SmartMeter by subtracting a value calculated based on the normal distribution of the `base_drop`. There is a small chance (0.1%) for the voltage calculation to produce an erroneous, high value that is a one-off anomaly ,and these are reported. When the `FeederLine` reaches the last device, it will begin recalculating the first device in the line on the next iteration using a `bonus` modifier. It contains a function `correct_bonus()` that, if the voltage is too high or low, will modify the `bonus` property by half the difference from expected values, so that it will normalize to the optimal value over time.

The `PowerGrid` class holds a user-defined number of `FeederLine` instances and stores the base parameters used during `FeederLine` creation. Each `FeederLine`’s parameters are initialized using `PowerGrid`’s parameters with an element of randomization. A `PowerGrid` is able to add and remove `FeederLines` in order to scale testing. Lastly, every `PowerGrid` instance has a function `update()` that iterates through all `FeederLines` and all devices within them, returning the following values each time: `FeederLine` number, device hops from `PowerGrid`, voltage, device ID, and current voltage status. Possible voltage statuses are as follows:

- `Normal` - voltage at device is within normally expected deviations
- `High` - voltage seems to be trending higher than necessary
- `Low` - voltage seems to be trending lower than acceptable values
- `Anomaly` - voltage does not correspond to meaningful value, must be error

2. Server Setup with AWS CloudFormation

We used AWS CloudFormation extensively to provision and configure the required AWS resources. The CloudFormation Template was written in JSON (`smartmeter_cloudformation.json`) and contains setup information for the EC2 instance as well as the different AWS Services.

AWS IOT

The AWS IoT and its different components were instantiated using the CloudFormation tempate:

- *Message Broker* — Provides a secure mechanism for things and AWS IoT applications to publish and receive messages from each other. We use the MQTT protocol to publish and subscribe.
- *Rules Engine* — Provides message processing and integration with other AWS services. We use a SQL-based language to select data from message payloads, process the data, and send the data to other services, such as Amazon S3, Amazon DynamoDB, and AWS Lambda.
- *Thing Registry* — Organizes the resources associated with each thing including associating attributes with each device. In our case we associate certificates and MQTT client IDs to improve ability to manage and troubleshoot.
- *Thing Shadows Service* — Provides persistent representations of things in the AWS cloud so that the thing does not always have to be connected. Our simulator publish its current state to the thing shadow for use by other applications.
- *Thing Shadow* — This is the JSON document used to store and retrieve current state information for a thing.
- *Device Gateway* — Enables the Simulator to securely and efficiently communicate with AWS IoT.

The things, policies and rules that are generated during the setup can be seen if signing in to the AWS IoT console.

Kibana Analytics

In this part of the system we are aggregating and analyzing the device data in near real time with Amazon Kinesis Firehose and Amazon Elasticsearch. Three key components are:

- *Amazon Kinesis Firehose*: Amazon Kinesis Firehose is the easiest way to load streaming data into AWS. Using Amazon Kinesis Firehose we load the streaming Voltage data to Amazon Elasticsearch Service and backup to Amazon S3. This is accomplished by defining an AWS IoT Firehose Rule Action.
- *Amazon Elasticsearch*: Our device data will be stored and indexed in Amazon Elasticsearch. Being a managed service it is easy to deploy, operate, scale and eliminates administration overheads like patch management, failure detection, node replacement, backups, and monitoring. As Amazon Elasticsearch Service includes

built-in integration with Kibana it eliminates the need for us to install and configure Kibana simplifying our process further.

- *Firehose Action*: A firehose rule action sends the device data from the MQTT message that triggered the rule to an Amazon Firehose stream.

The smartmeter CloudFormation template provisioned our Amazon Elasticsearch cluster, and created the Amazon Elasticsearch index for our device data mapping. This index was created through a simple bash script which we ran while bootstrapping our EC2 Instance.

```
"./smart-meter-device-data -d '{",
"\"mappings\" : {\n",
"\"device-message\" : {\n",
"\"properties\" : {\n",
"\"Line\": { \"index\": \"analyzed\", \"store\": \"yes\", \"type\":
\"long\"},\n",
"\"Hops\": { \"index\": \"analyzed\", \"store\": \"yes\", \"type\":
\"long\"},\n",
"\"Voltage\": { \"index\": \"analyzed\", \"store\": \"yes\", \"type\":
\"double\"},\n",
"\"Status\": { \"index\": \"analyzed\", \"store\": \"yes\", \"type\":
\"string\"},\n",
"\"Modifier\": { \"index\": \"analyzed\", \"store\": \"yes\", \"type\":
\"long\"},\n",
"\"DeviceID\": { \"index\": \"analyzed\", \"store\": \"yes\", \"type\":
\"string\"},\n",
"\"timeStampEpoch\": { \"index\": \"analyzed\", \"store\": \"yes\", \"type\":
\"long\"},\n",
"\"timeStampIso\": { \"index\": \"not_analyzed\", \"store\": \"yes\", \"type\":
\"date\" },\n",
"\"location\": { \"index\": \"not_analyzed\", \"store\": \"yes\", \"type\":
\"geo_point\" }}}}}'"
```

Anomaly Detection using AWS Machine Learning

This part of the system adds a simple test of Machine Learning, where we detect whether device messages are anomalous based on the voltage report rather than the message status value, i.e. we "predict" the voltage abnormality. If an anomaly is detected, an email is sent to the email provided as part of the CloudFormation setup. In addition to using AWS Lambda function, we are using the following services:

- Amazon Machine Learning: We trained a new ML model with a sample data set (generated from the Simulator).
- Amazon Kinesis Streams: Device messages are staged in a Kinesis Stream, where it is then retrieved by a Lambda function that compares the device measurements against the predicted value.
- Amazon Simple Notification Service: Amazon SNS is then responsible for sending push notifications to the email recipients if an anomaly is detected.

Test Run Results

Kibana Analytics

The following section contains an analysis of a test run of the upper data flow in the architecture for 25 minutes on December 10, 2016 between 07:50 and 08:15 am PST. The analysis of the streaming data is using AWS ElasticSearch 1.3 and Kibana 4.0.3.

The simulator were set to the following configuration settings:

- settings.py:
    - NUM_LINES = 20 # 20 feeder lines created by simulator
    - BASE_VOLTAGE = 123 # Starting voltage of 123V on each feeder line
    - BASE_DROP = 1 # Drop in voltage between hops on feeder line (randomized)
    - NUM_DEVICES = 40 # Number of bellwether meters reporting on a feeder line
- app.py:
    - time.sleep(0.5) # 0.5 sec between sending a message from simulator, i.e. 6.67 minute for each individual bellwether meter
- feederline.py
    - random.uniform(0, 1) <= 0.001: # 0.001% chance of anomalies (erroneous voltage data)

This test setup corresponds to a pilot system of 800 bellwether meters or about 12,000 households. Some existing, on-premise RDBMS data collection systems have processing capacity issues even at this level due to frequent read/write processing. To support a full scale system, our system would be expanded 100 times to a million of households.

The simulator is initiated by the following command from the ec2-user directory:

```
[ec2-user@ip-10-0-0-21 ~]$ python MIDS-W205_Project/Simulator/app.py
```

After starting the Simulator we go to the AWS Elasticsearch management console and select the SmartMeter Elasticsearch domain under the Dashboards column and then click on the Kibana endpoint to visualize the device data.

Inside Kibana, all we need to set up is our device-mapping index for us to visualize the SmartMeter device data.

1. Under Index name, type smart*
2. Click in the Time-field-name dialog box and select timeStampIso
3. Click Create.
4. Once the mapping has been created, selecting the Discover tab in the navigation bar starts to visualize the device data.

The data shown is for the last 15 minutes by default. For this run we altered to capture 25 minutes within the visualizations to get a better grasp on how the Simulator worked.

The first set of visualizations in Kibana are grouped in an AnomalyDashBoard and as the name suggests, this dashboard focuses in on anomalies as these are critical to detect and correct immediately. Input settings for the initial AnomaliesPerLine graph is:

- Search Filter: Status: Anomaly
- Y-axis:
    - Aggregation: Count
- Split Bars:
    - Aggregation: Histogram
    - Field: Hops
    - Interval: 1
    - X-Axis:
        - Sub Aggregation: Histogram
        - Field: Line
        - Interval: 1
- Time Filter: 2016-12-10 07:50:00.000 to 2016-12-10 08:15:00.000

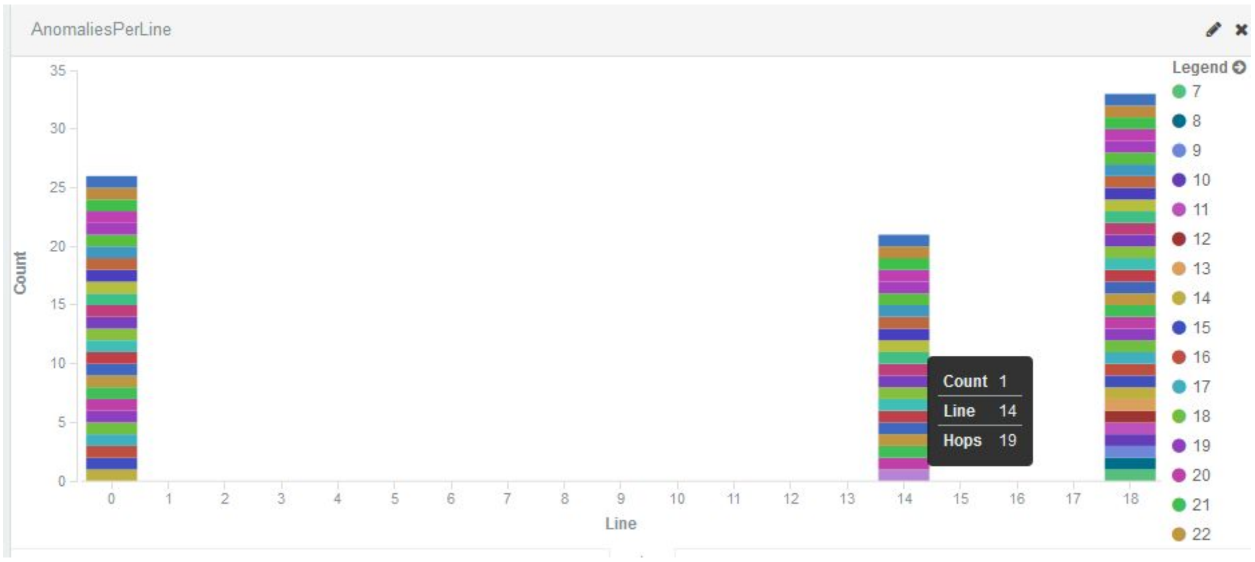
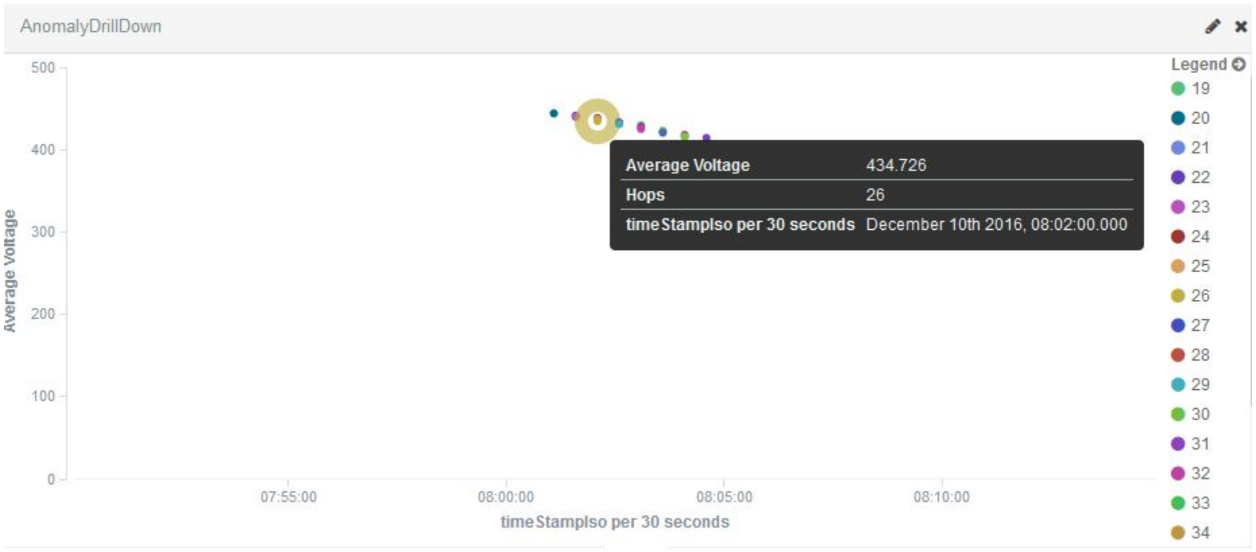


Figure 4. Anomalies Count per Feeder Line across time filter interval.

The graph shows each feeder line on the x-axis and the count of anomalies received on the y-axis. The legend helps color code the bellwether meters (hops) along the line that were affected. We can, e.g., see that the Feeder Line 14 experienced anomalies and that bellwether meters from hops 19 and on were affected.

We then make a drill-down on Feeder Line 14 using the AnomalyDrillDown Line Chart and can identify not only the individual bellwether meters (hops) but also the abnormal voltage readings and exact timestamp of the events. Input settings for this graph is below:

- Search Filter: Line: 14 AND Status: Anomaly
- Y-axis:
  - Aggregation: Average
  - Field: Voltage
- X-Axis:
  - Aggregation: Data Histogram
  - Field: timeStampIso
  - Interval: Auto
  - Split Lines:
    - Sub Aggregation: Histogram
    - Field: Hops
    - Interval: 1
- Time Filter: 2016-12-10 07:50:00.000 to 2016-12-10 08:15:00.000



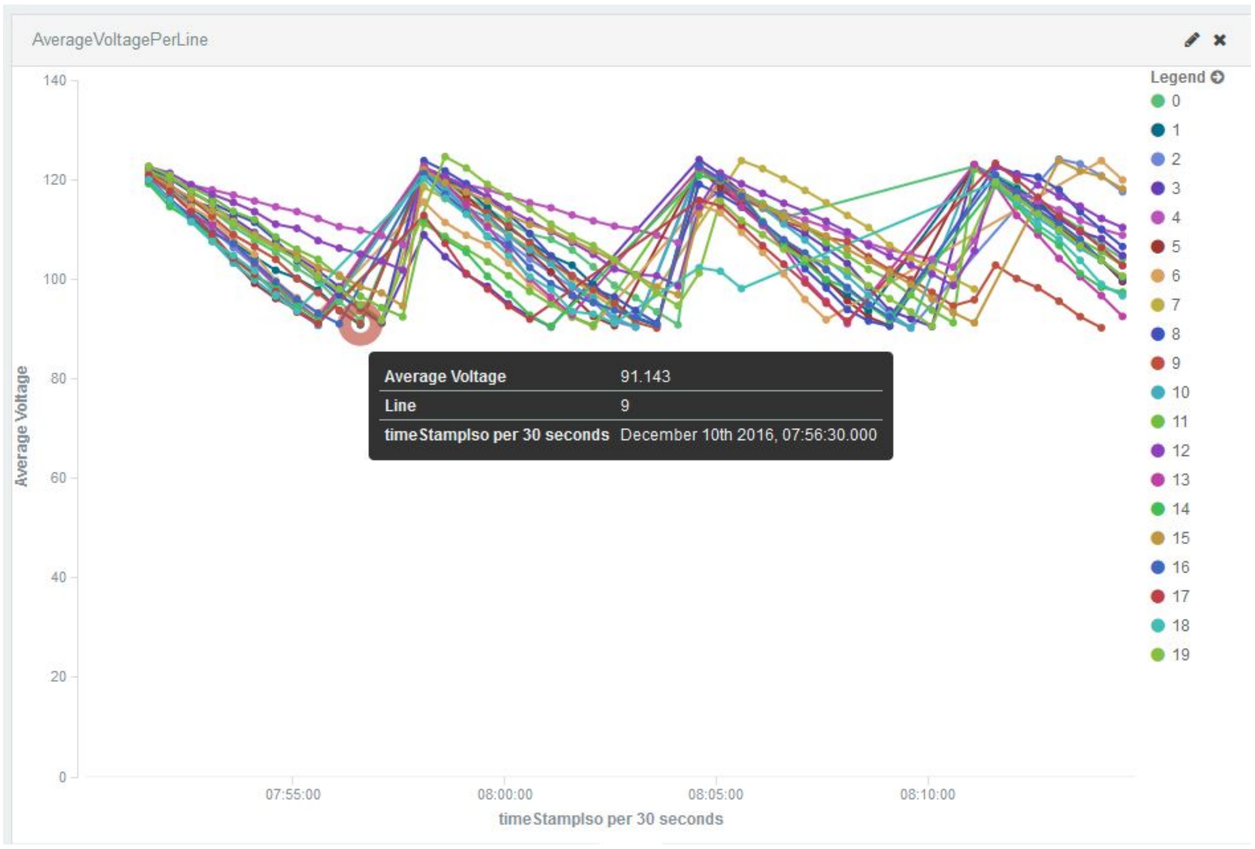
Average Voltage	434.726
Hops	26
timeStampIso per 30 seconds	December 10th 2016, 08:02:00.000

Figure 5. Anomaly Drill Down with Individual Plotting and Timestamp.

The Legend color marking helps indicate hop number and as such the bellwether meter affected during the anomaly. We can also in this graph see how an anomaly propagates through the Feeder Line from where it started. If you are interested in another Line, you can simply change the filter by clicking on the edit marker in the top right corner.

Now when the anomalies are analysed, we move forward to review how the voltage optimization of the system is working for the rest of the bellwether meters. For this analysis we use a PowerGridDashboard with three views. We start with seeing the simple measurement count of each voltage level between 90 and 125V using the AvergageVoltagePerLine Chart. The input data for this chart is:

- Search Filter: Voltage: {90 TO 125}
- Y-axis:
    - Aggregation: Average
    - Field: Voltage
- X-Axis
    - Aggregation: Date Histogram
    - Field: timeStampIso
    - Interval: Auto
- Time Filter: 2016-12-10 07:50:00.000 to 2016-12-10 08:15:00.000



Average Voltage	91.143
Line	9
timeStampIso per 30 seconds	December 10th 2016, 07:56:30.000

Figure 6. Average Voltage per Line across Time Filter.

The Legend color coding does in this case indicate the different Feeder Lines. The graph shows that the voltage optimization algorithm generally has a 5 min cyclic behavior for most Feeder Lines. A problem we can see is that the lower and upper voltage extremes are too high.

To get a feeling for the distribution of the measurement counts across our voltage spann, we use a Bar Chart called BarChartVoltageCount:

- Search Filter: Voltage: {90 TO 125}

- Y-axis:
  - Aggregation: Count
- X-Axis
  - Aggregation: Histogram
  - Field: Voltage
  - Interval: 1
- Time Filter: 2016-12-10 07:50:00.000 to 2016-12-10 08:15:00.000

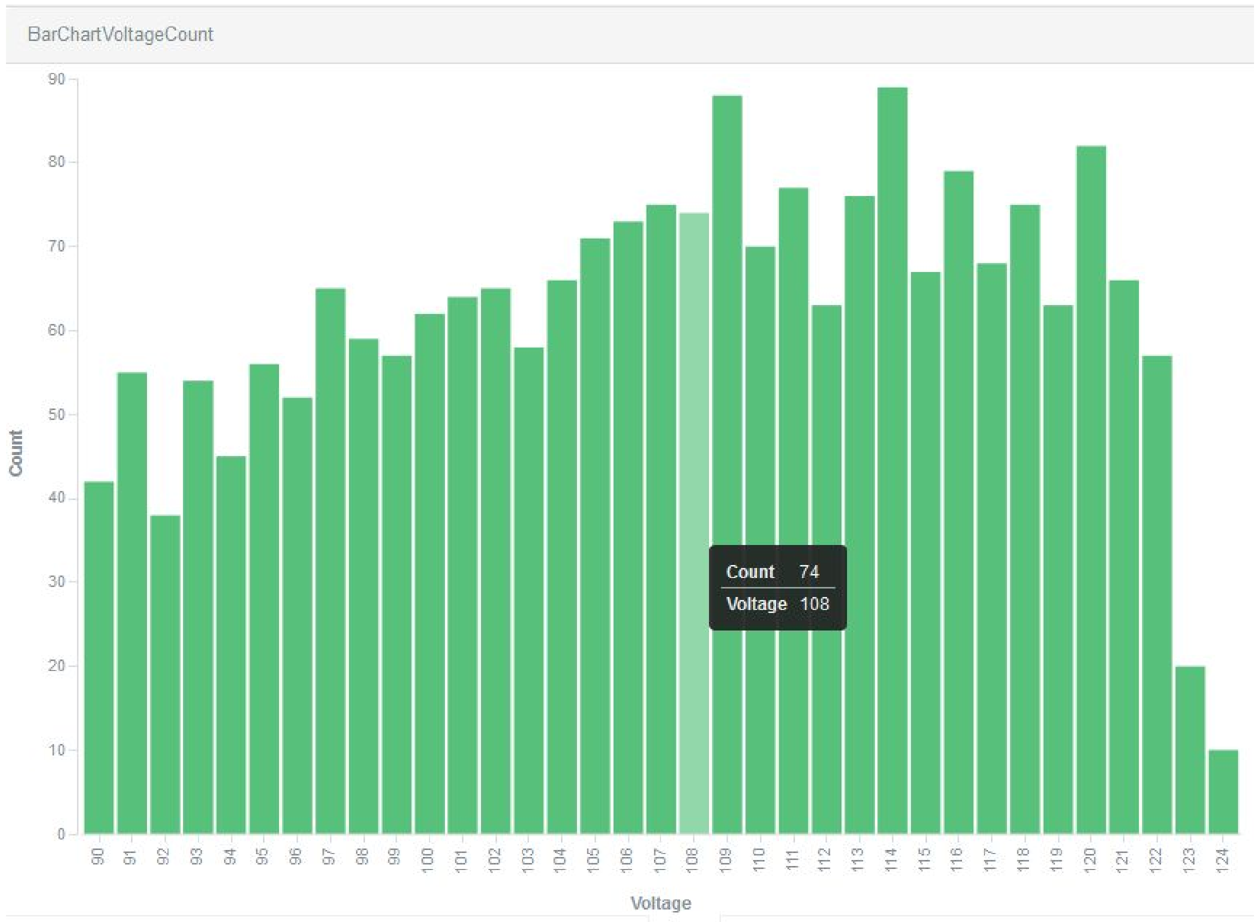


Figure 7. Measurement Count for different Voltage Levels across all Feeder Lines.

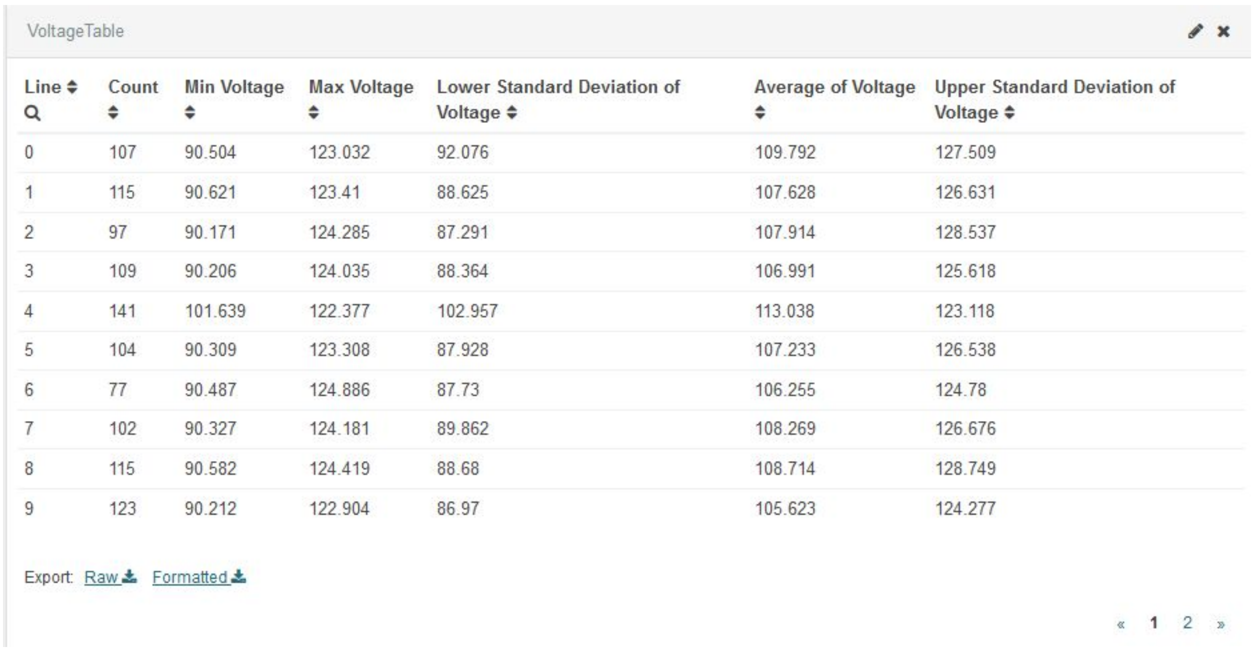
This shows that the algorithm for voltage optimization in the simulator is also not fully optimized in terms of centering the voltage to the sweet spot of 117-123V. Instead, we have a fairly even distribution across the voltage spann that the algorithm is spanning.

Finally, we also provide a tableview (VoltageTable) in the dashboard that allows us to drill down per Feeder Line the distribution of voltage readings through a set of statistical measures. This view is built with:

- Search Filter: Voltage: {90 TO 125}
- Metrics:

  - Aggregation: Count
- Metrics:
  - Aggregation: Standard Deviation
- Field: Voltage
- Metrics:
  - Aggregation: Voltage
  - Field: Min
- Metrics:
  - Aggregation: Voltage
  - Field: Max
- Metrics:
  - Aggregation: Percentiles
  - Field: Voltage
  - Percentiles: 23, 75
- Buckets
  - Split Rows:
    - Aggregation: Histogram
    - Field: Line
    - Interval: 1
- Time Filter: 2016-12-10 07:50:00.000 to 2016-12-10 08:15:00.000

These configuration settings create the following table.



VoltageTable

Line	Count	Min Voltage	Max Voltage	Lower Standard Deviation of Voltage	Average of Voltage	Upper Standard Deviation of Voltage
0	107	90.504	123.032	92.076	109.792	127.509
1	115	90.621	123.41	88.625	107.628	126.631
2	97	90.171	124.285	87.291	107.914	128.537
3	109	90.206	124.035	88.364	106.991	125.618
4	141	101.639	122.377	102.957	113.038	123.118
5	104	90.309	123.308	87.928	107.233	126.538
6	77	90.487	124.886	87.73	106.255	124.78
7	102	90.327	124.181	89.862	108.269	126.676
8	115	90.582	124.419	88.68	108.714	128.749
9	123	90.212	122.904	86.97	105.623	124.277

Export: Raw Formatted

« 1 2 »

Table 1. Voltage Statistics per Feeder Line.

We cannot only see that the min voltage is generally too low, but also that the standard deviation spann is wide across each Feeder Line so a systematic problem with the voltage optimization algorithm as it is designed now and something to work on in the future.

The static views of the two analytics dashboards we created as part of this analysis can also be accessed on the Internet at the following URLs:

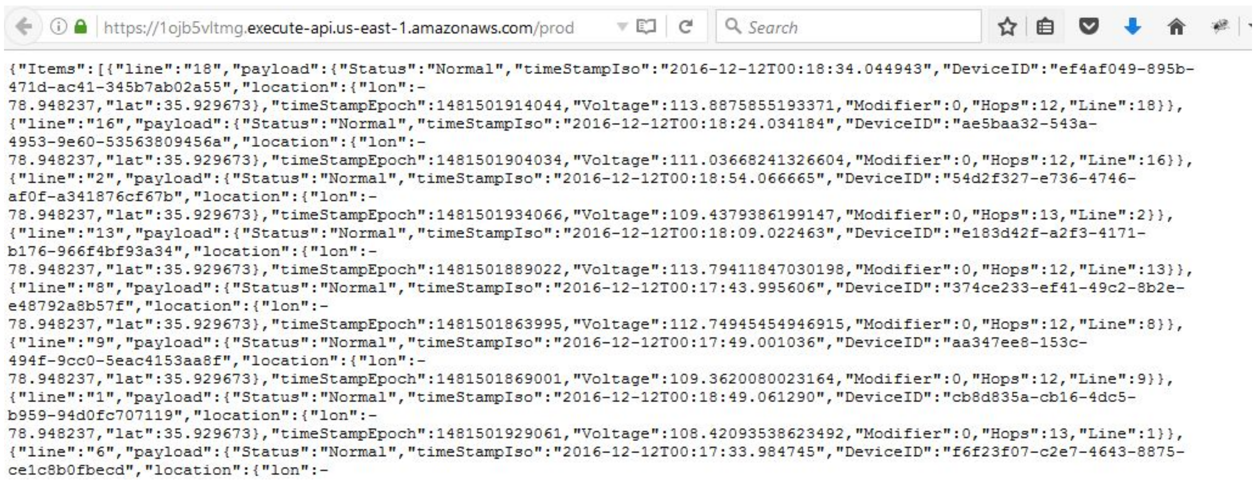
- AnomalyDashBoard
- PowerGridDashBoard

Storing Data in DynamoDB

In this section we describe a test run to see that the middle data flow in the IoT Server architecture works and that the API for the data stored in DynamoDB is accessible.

During the system setup, we created rules to write time series data from the Simulator into a DynamoDB table called SmartMeterDynamoTimeSeriesTable as well as to write the latest received messages into a table called SmartMeterDynamoDeviceStatusTable. The actual DynamoDB table names will be prefixed by the name chosen for the CloudFormation stack. (e.g. in our case SmartMeter-SmartMeterDynamoTimeSeriesTable)

In Amazon API Gateway, a stage defines the path through which an API deployment is accessible. The CloudFormation template already configured a production stage called 'prod'. In the API Gateway console we can click on Stages and then prod to review the current API configuration. When clicking on the link next to Invoke URL (https://1ojb5vltmg.execute-api.us-east-1.amazonaws.com/prod), we see SmartMeter data in JSON format. We can refresh the page every few seconds and notice that the data changes:



```
{"Items":[{"line":"18","payload":{"Status":"Normal","timeStampIso":"2016-12-12T00:18:34.044943","DeviceID":"ef4af049-895b-
471d-ac41-345b7ab02a55","location":{"lon":-
78.948237,"lat":35.929673},"timeStampEpoch":1481501914044,"Voltage":113.8875855193371,"Modifier":0,"Hops":12,"Line":18}},
{"line":"16","payload":{"Status":"Normal","timeStampIso":"2016-12-12T00:18:24.034184","DeviceID":"ae5baa32-543a-
4953-9e60-53563809456a","location":{"lon":-
78.948237,"lat":35.929673},"timeStampEpoch":1481501904034,"Voltage":111.03668241326604,"Modifier":0,"Hops":12,"Line":16}},
{"line":"2","payload":{"Status":"Normal","timeStampIso":"2016-12-12T00:18:54.066665","DeviceID":"54d2f327-e736-4746-
af0f-a341876cf67b","location":{"lon":-
78.948237,"lat":35.929673},"timeStampEpoch":1481501934066,"Voltage":109.4379386199147,"Modifier":0,"Hops":13,"Line":2}},
{"line":"13","payload":{"Status":"Normal","timeStampIso":"2016-12-12T00:18:09.022463","DeviceID":"e183d42f-a2f3-4171-
b176-966f4bf93a34","location":{"lon":-
78.948237,"lat":35.929673},"timeStampEpoch":1481501889022,"Voltage":113.79411847030198,"Modifier":0,"Hops":12,"Line":13}},
{"line":"8","payload":{"Status":"Normal","timeStampIso":"2016-12-12T00:17:43.995606","DeviceID":"374ce233-ef41-49c2-8b2e-
e48792a8b57f","location":{"lon":-
78.948237,"lat":35.929673},"timeStampEpoch":1481501863995,"Voltage":112.74945454946915,"Modifier":0,"Hops":12,"Line":8}},
{"line":"9","payload":{"Status":"Normal","timeStampIso":"2016-12-12T00:17:49.001036","DeviceID":"aa347ee8-153c-
494f-9cc0-5eac4153aa8f","location":{"lon":-
78.948237,"lat":35.929673},"timeStampEpoch":1481501869001,"Voltage":109.3620080023164,"Modifier":0,"Hops":12,"Line":9}},
{"line":"1","payload":{"Status":"Normal","timeStampIso":"2016-12-12T00:18:49.061290","DeviceID":"cb8d835a-cb16-4dc5-
b959-94d0fc707119","location":{"lon":-
78.948237,"lat":35.929673},"timeStampEpoch":1481501929061,"Voltage":108.42093538623492,"Modifier":0,"Hops":13,"Line":1}},
{"line":"6","payload":{"Status":"Normal","timeStampIso":"2016-12-12T00:17:33.984745","DeviceID":"f6f23f07-c2e7-4643-8875-
ce1c8b0fbecd","location":{"lon":-
```

Figure 8. Amazon API Output for Web Dashboards.

We can also test the API from the command line interface:

```
[ec2-user@ip-10-0-0-21 ~]$ curl
https://1ojb5vltmg.execute-api.us-east-1.amazonaws.com/prod
```

Anomaly Detection using AWS Machine Learning

This next section describes the test run of the AWS Machine Learning component to detect anomalies based on prediction and then send an Email alert when these are occurring. The training set ‘TrainDataForML.csv” contained 2,500 rows.

The ML model we created is using Numerical Regression. We selected saving 30% of training data to evaluate the model. For regression tasks, Amazon ML uses the root mean square error (RMSE) metric.

$$RMSE = \sqrt{1/N) \sum_{i=1}^{N} (actual\, target - predicted\ target)^2}$$

Amazon ML provides a baseline metric for regression models (Baseline RMSE). It is the RMSE for a hypothetical regression model that would always predict the mean of the target as the answer. In our evaluation test run, the ML model's quality score is worse than the baseline.

- RMSE: 49.3088
- RMSE baseline: 48.862
- Difference: 0.447

The next chart shows the distribution of the residuals of the ML model. We can see a slightly positive residual indicating that the model is underestimating the target (the actual target is larger than the predicted target). We expect this is because of the anomalies that we have created with near zero voltage.

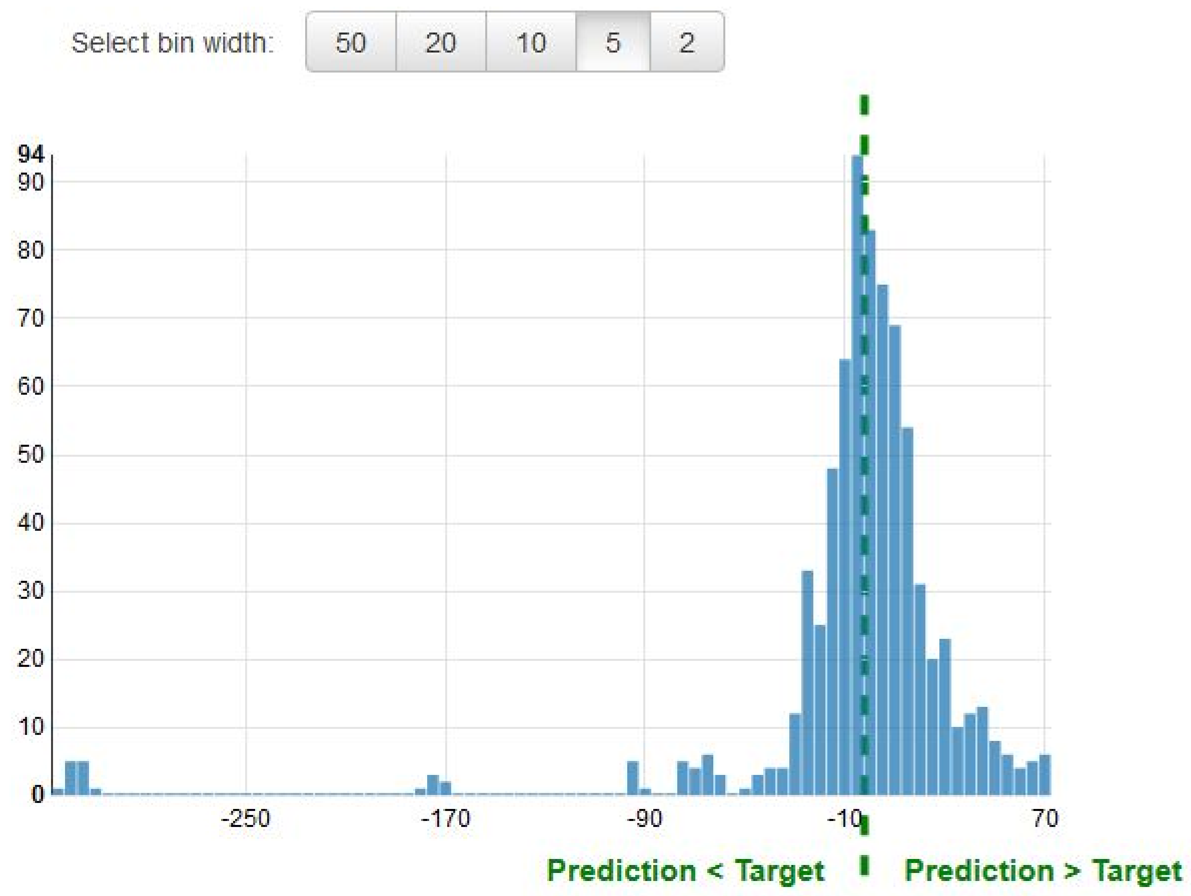


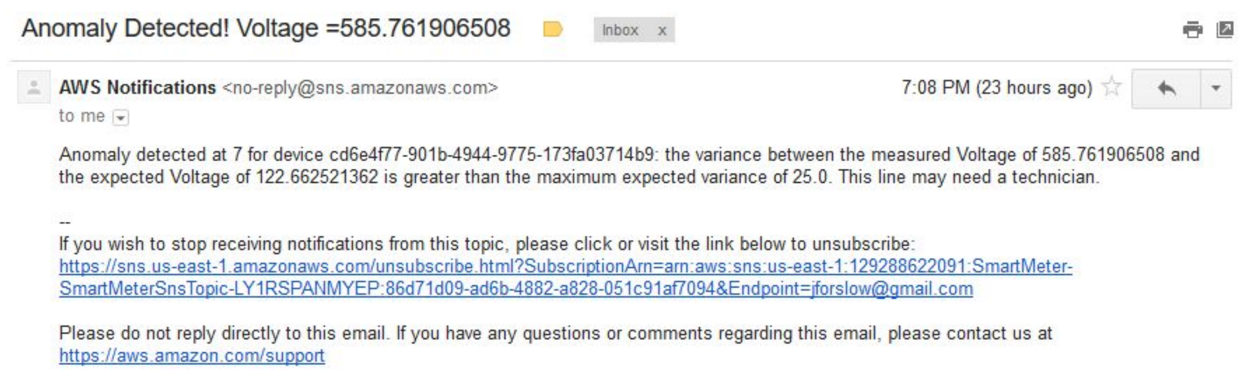
Figure 9. Machine Learning Model Performance Residuals.

For Email Alerts, we already provided an email address at the beginning of the CloudFormation setup. However, as needed we can also change this email address by going to the [AWS SNS Console]() and select the ARN for our SmartMeter topic and apply it in a new Subscription. As part of this process, a Subscription Confirmation email is received that needs to be acknowledged.

We are now ready to run our Simulator to validate that the anomaly detection using Machine Learning creates Email SNS Alerts:

```
[ec2-user@ip-10-0-0-21 ~]$ python MIDS-W205_Project/Simulator/app.py
```

Within a few minutes, we can see the first email alerts coming in.



Anomaly Detected! Voltage =585.761906508

Inbox x

AWS Notifications <no-reply@sns.amazonaws.com>
to me

7:08 PM (23 hours ago)

Anomaly detected at 7 for device cd6e4f77-901b-4944-9775-173fa03714b9: the variance between the measured Voltage of 585.761906508 and the expected Voltage of 122.662521362 is greater than the maximum expected variance of 25.0. This line may need a technician.

--
If you wish to stop receiving notifications from this topic, please click or visit the link below to unsubscribe:
https://sns.us-east-1.amazonaws.com/unsubscribe.html?SubscriptionArn=arn:aws:sns:us-east-1:129288622091:SmartMeter-SmartMeterSnsTopic-LY1RSPANMYEP:86d71d09-ad6b-4882-a828-051c91af7094&Endpoint=jforslow@gmail.com

Please do not reply directly to this email. If you have any questions or comments regarding this email, please contact us at https://aws.amazon.com/support

Figure 10. Email Alert on Detected Voltage Anomaly.

We had set the expected max variance to be 25.0 from the trained expected value of 122V. As our Simulator creates some voltage values below that limit even in “Status: Normal”, we get a slightly overly verbose email alert system in this test run.

<u>Data Processing and Cost Assessment</u>

We did verify the CPU utilization of our EC2 instance (t2.medium) during the test run and it stayed well below our system capacity as can be seen below. Still as part of expanding this system to commercial scale, we have proposed the option to add AWS Autoscaling Groups as an extension (see Future Extensions).

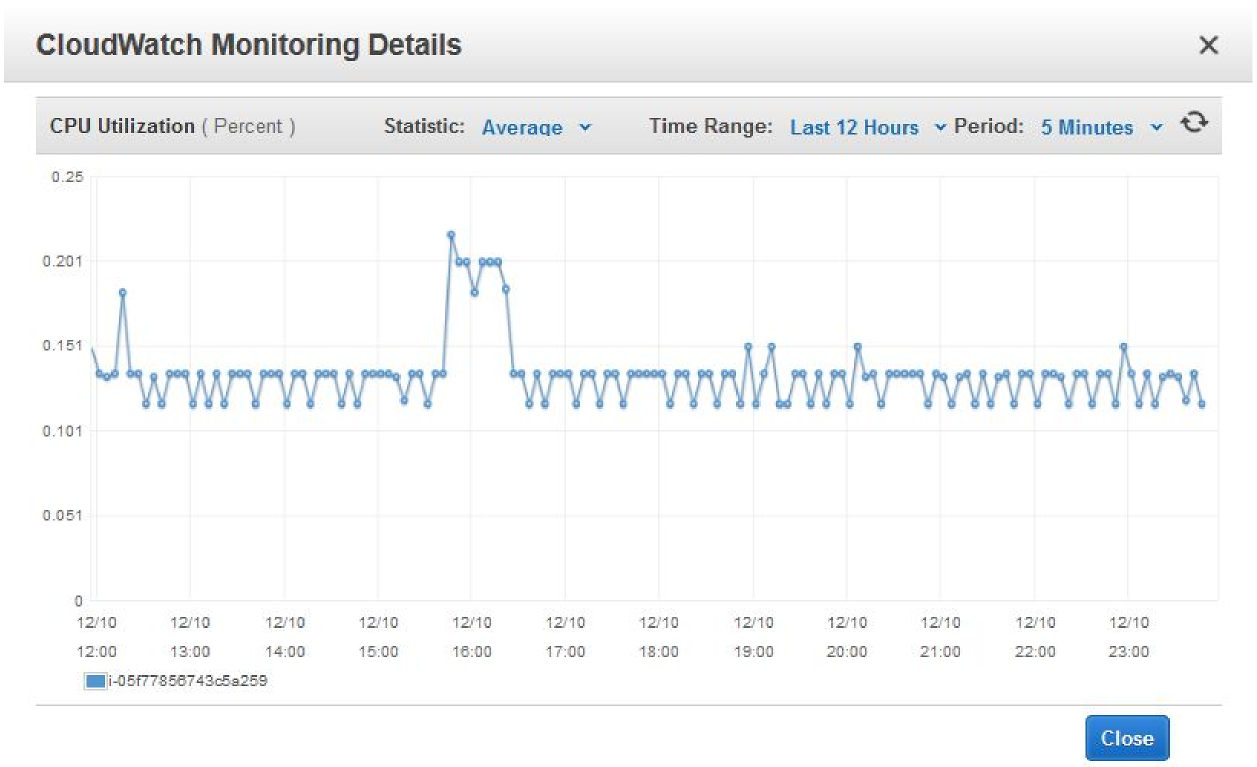


Figure 11. CloudWatch Monitoring Details of EC2 instance.

The data usage of the AWS IOT Gateway was also verified and showed that 3,505 messages were published during the 25 minute test window. Given AWS Pricing policy of $5 per million messages, the cost for continuous 24*7 of this pilot setup for a year would only reach ~ $300 for the IOT Gateway. This seems more than reasonable as compared to investing in on-premise infrastructure as current Conservation Voltage Reduction systems do. Note that the scale out of the AWS infrastructure is automatically taken care of in the case of AWS IoT as it is provided as an AWS Service and not running on our EC2 instance.

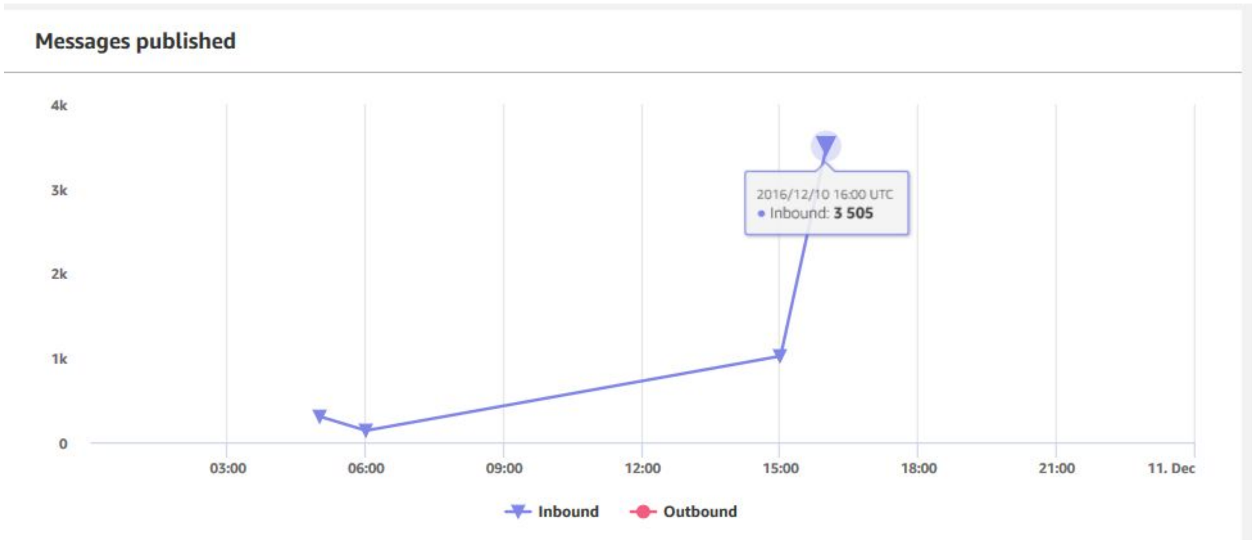


Figure 12. Number of Publish Messages Coming into the AWS IOT Gateway.

As for the Lambda server-less processing, we could see that the processing time was within 4 sec for each operation, which is a bit high but within the realm of what is tolerable for a voltage control mechanism.

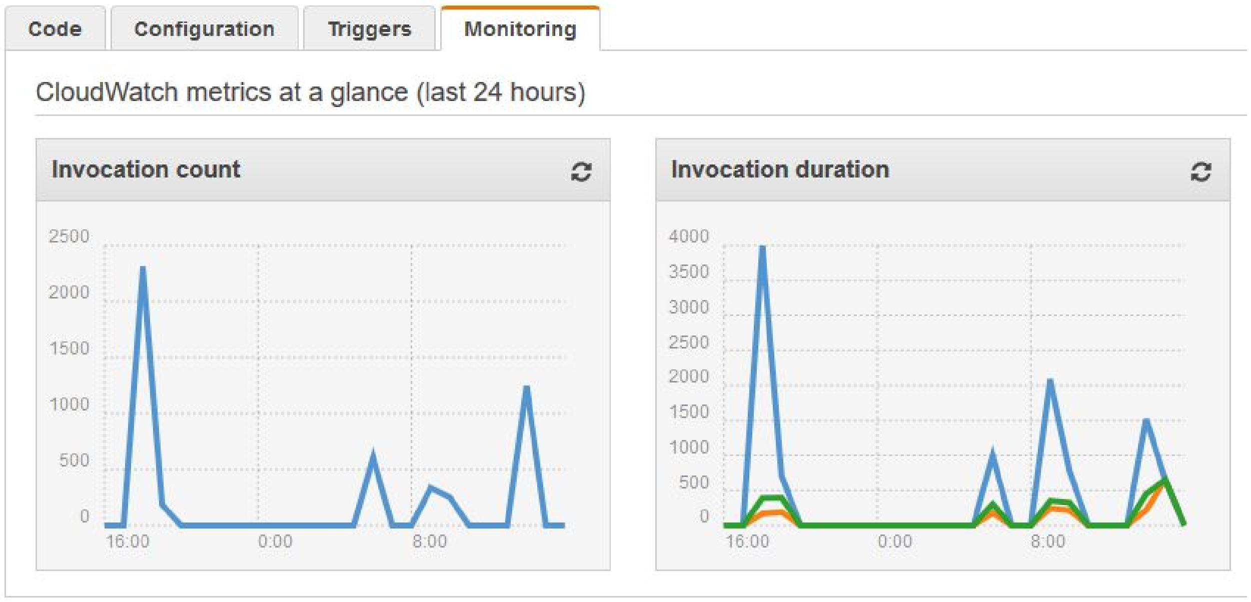


Figure 13. Lambda Functions Invocation Count and Processing Duration.

Future Extensions

Based on the Kibana test run and analysis it is clear that we would need to spend more time fine tuning the Simulator to match better realistic Voltage levels. As mentioned already, one other extension we have prepared hooks for, but did not have enough time to implement, is to add an automatic control loop from the IOT Server to the Device Simulator and associated algorithm. We could also enable location-data to be sent as part of the SmartMeter push messages. The natural extension after that is to actually interface the system towards real live SmartMeters and PowerGrid Voltage Regulators. One limitation is that AWS IoT only supports HTTP and MQTT today, which is not commonly used in SmartMeters.

In terms of IOT Server scalability, several components are already independent of the EC2 instance we created for the project. Both the AWS IOT Gateway and the Lambda server-less functions will run on any available resource inside AWS and are not limited to our EC2 configuration. However, in order to create Disaster Recovery for the complete setup we have plans to extend the CloudFormation template to include AWS autoscaling groups. This would allow us to enforce the use of a minimum of two EC2 instances placed in two different regions and also thresholds for when to scale out and back again with new instances as volume shifts. We also should run the system with more load (approx 100,000 of bellwether meters) to correspond to a million household system. Other extensions on the server side includes more work on an advanced voltage assessment and control algorithm as well as to complete a web dashboard for power grid operators.