EC535 Technical Project Report
Predictive Maintenance and Data Logger for Industry 4.0

James Conlon and Animisha Sharanappa

Link to Github: https://github.com/proconlon/ec535-final
Link to video demo: https://youtu.be/PAorCIHc_FU

Abstract

By evaluating the condition of in-service equipment, predictive maintenance techniques aim to determine when maintenance should be performed. Data loggers are embedded devices that use sensors or other equipment to capture data over extended periods of time. An ML model can be trained using the hours of recorded data to anticipate machine breakdowns and stop catastrophic failures in industrial machinery. For factories that depend on them, machine breakdowns can be catastrophic and result in unplanned downtime. A single malfunction can halt the entire working pipeline because factories depend on their machines operating on time with high uptime to meet their output requirements. Thus, being able to anticipate and stop machine breakdowns in advance is essential. Our project is aimed to allow for predictive maintenance to take place in a cost effective and simple way that saves on cloud costs and allows computation on site.

Introduction

Unplanned downtime of a machine can be incredibly costly for a factory. Imagine a worst case scenario where a critical valve in an injection molding machine fails, causing the molten plastic to flood the die, and requiring the entire machine to be shut down for cleaning and replacement of all downstream components that are damaged. [1] If you had someone doing routine maintenance on the machine: by looking for “rust” or other signs of wear, they may spot the failing valve before the failure and replace it. While in both cases, the part is still needing to be replaced: the key point is replacing **before** the valve fails is much better for the machine operator because the failure is preventable. A failure that wasn’t spotted in advance would lead to unplanned downtime, likely hours longer, and the failure would have taken place during a machine running, so there may be issues with the batch/output of the machine, as well as more work into restoring the machine.

Predictive maintenance is a step above the basic level of a machine operator that looks over the machine with a clipboard, checking for parts that need to be replaced. Predictive maintenance (PdM) uses sensors and data collection on key machine components in order to monitor machine uptime and historical information in order to predict when failures are likely to occur (Figure 1, [2]). In the valve metaphor, accelerometers would be used to collect high frequency vibration information that can reveal a failing part before any visible wear is noticed. The goal is to collect many data points and use machine learning to predict these types of failures before they take

place. However, this is an area of IIoT (Industrial IoT) that is currently underserved. Traditional industrial SCADA (Supervisory Control and Data Acquisition) loggers either upload everything to the cloud (which is expensive) or discard raw data at the edge (losing valuable context). [3,4] This leaves a gap for our project. This project aims to provide a way to approach the problem of network and storage constraints for Industry 2.0 operators. We chose to approach this problem by doing high frequency data collection at the edge (the edge device being the BeagleBone), in order to allow users to tune their device to upload only data they need to collect for historical purposes and collect higher frequency data for ML training. Our device also should continue to collect low frequency data that is already commonplace collected, for historical archiving.
We address by creating an edge gateway on a BeagleBone Black that:

1. Collects high-frequency (100Hz) data from OPC-UA simulated 5-stage injection molding machine.
2. Logging data at two rates: high-frequency for local ML and low-frequency (~4Hz) for cloud storage, with file rotation to prevent SD card overflow
3. Buffers HF and LF CSVs to AWS S3 bucket every 5 minutes in a network failure resistant way, without archiving unnecessary data.
4. Running on device ML model to predict simulated failures and email operator ~10s before simulated failure actually takes place.
5. Allows for remote SSH and machine access through Tailscale configured with exit node and subnet router.

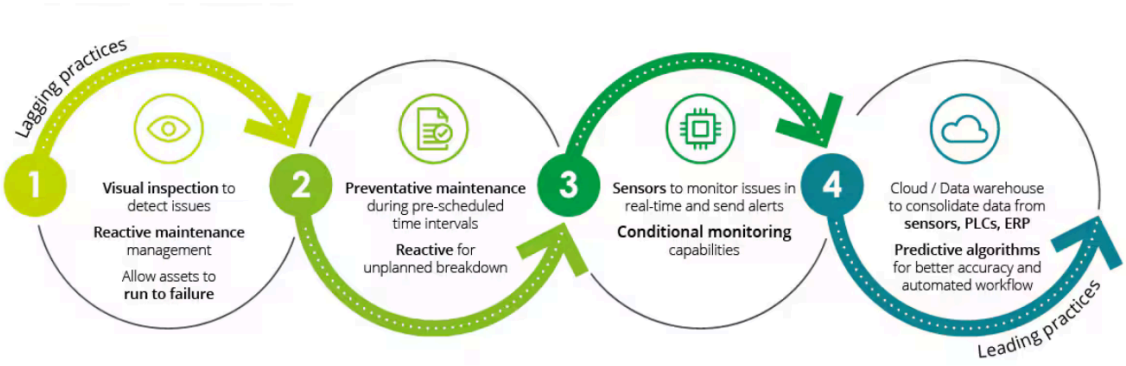

Figure 1: Deloitte[2] - Evolution of maintenance

Method

Here we describe each core component of the edge gateway. See figure 2 for a full overview of all components of the system.

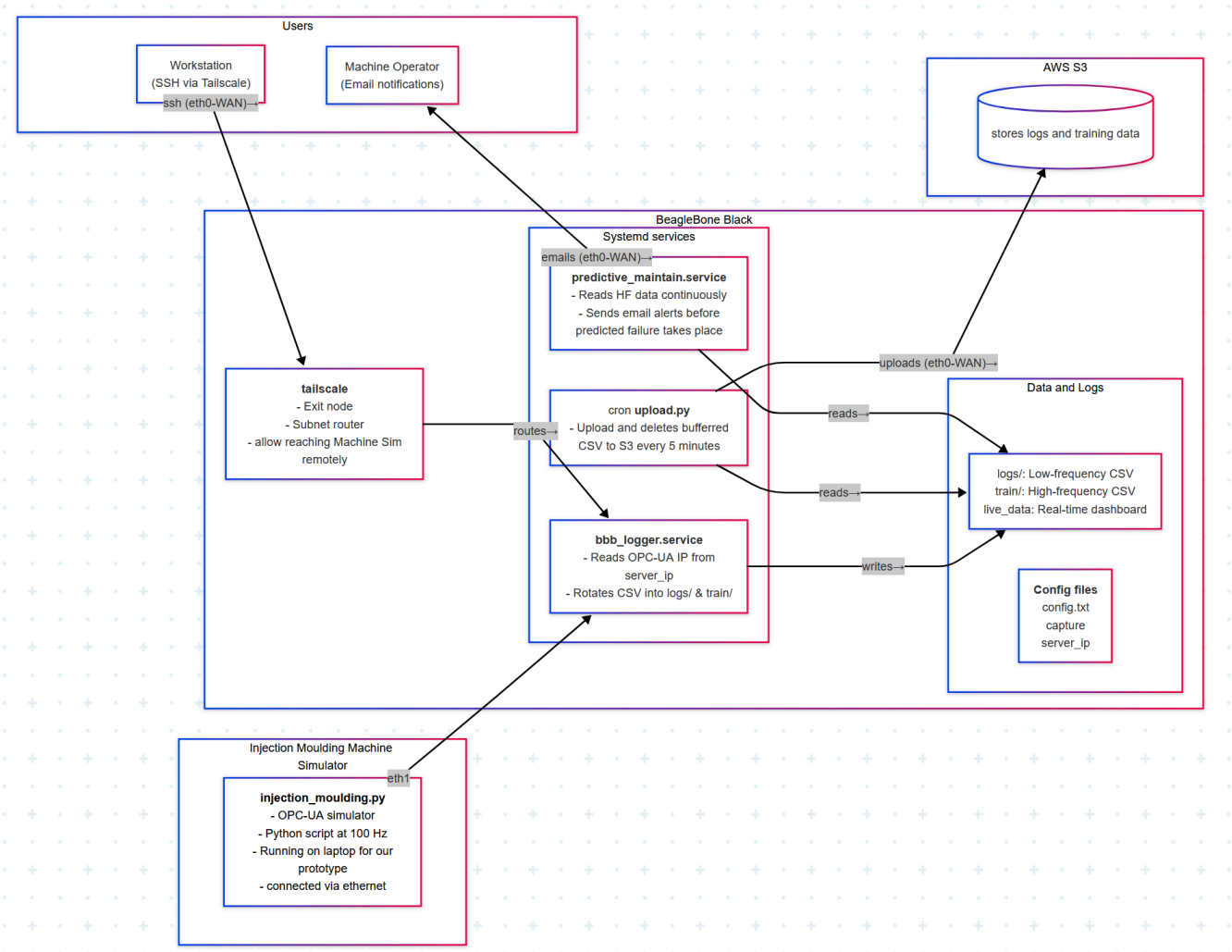

Figure 2: System Overview

Our first idea for this project was to implement a status page similar to the /proc entries we implemented in class. This would be exhaustive as a full system overview with all details of the system. We chose to not implement it as a kernel module like in class, but instead keep the program running with a systemd service. This was chosen to keep the program fail tolerant as it is possible for (a poorly written) C program to crash if left running overnight, and keeping in a systemd service allows automatic restarting.

The design of the system running on the BeagleBone was carefully designed to not introduce blocking threads in the main loop that would slow down the data collection. For that reason, we implement the uploading and predictive maintenance aspect of the system in a separate systemd service and through a crontab respectively.

In the upcoming sections, this report describes all the major features of the system that is laid out in Figure 2. The final file tree of all running services is shown in Figure 3.

```
/home/debian
├── bbb_logger_arm            (C binary)
├── predictive_maintain.py    (ML model/email notifier)
├── upload.py                 (cron to upload to AWS)
├── config.txt                (config file for bbb_logger)
├── capture                   (0 or 1, controls HF data capture)
├── ip_address                (ip address of OPC-UA server)
├── logs/                     (LF CSVs for cloud storage)
├── train/                    (HF CSVs for ML)
└── live_data                 (human-readable dashboard)
```

Figure 3: BeagleBone File tree

Primary Data Logger (bbb_logger.service)

We implement an OPC-UA client using open62541 [5] in C to read the data from the python machine simulator. These are hard coded in the "opcua_read_row" function in the opc.c file based on examples we found in the source repo for reading data. One important thing to note is that the namespace and node ids assigned to data points don't seem to be totally deterministic, so we had to use a combination of a test UA client (UA Expert [6]) and using a basic python script get_ns.py to pull all the node ids. For that reason, we chose to hard code the data we are reading in the binary because a config would need at least 4 pieces of information each: namespace, node id, data type, and data name– and this would be a lot to parse with plain C. We are comfortable hard coding the data points as a real machine would use far more than six data points, as it could have dozens or hundreds of data points, but for a proof of concept device all the numbers are pared down - like the way we choose to use KB rather than MB for storage metrics. Additionally, in real life situations, the device already must be configured for a specific machine – as the ML model must be tuned exactly for the machine itself, so the machine already must be configured – i.e. an out of the box solution is not a necessity for our target customers.

In figure 7, you see the primary status page. The primary function of this is the looping algorithm, which has a configurable High and low frequency (listed as HiHz and LoHz). We decimate the while loop to lower to the low frequency. For our testing, we test with HiHz at 100Hz (same frequency as the OPC-UA simulator– which should always be the case), and LoHz at 4 Hz (adaptable to user's collection needs). The decimateN integer calculation is HiHz/LoHz. Then in the primary loop, we sleep for usleep(1000000 / cfg.hiRateHz). See Figure 4 for C code details and Figure 5 for all the configurable parameters that can be configured during runtime with no issues.

```
110     while (g_keepRunning) {
248         // Low-rate logging always always goes to log_fp
249         if (tick % decimateN == 0) {
250             fprintf(log_fp, "%llu,%.2f,%.2f,%.2f,%.2f,%s,%u\n",
251                     row.ts_us, row.melt_temp, row.inj_press,
252                     row.vib_amp, row.vib_freq, row.stage, row.failure_label);
253             fflush(log_fp);
254             long pos = ftell(log_fp);
255             if (pos/1024 >= cfg.maxLogFileKB) {
256                 fclose(log_fp);
257                 log_fp = rotate_file("logs", "log");
258                 printf("Rotated LOG file as size reached %dKB\n", cfg.maxLogFileKB);
259             }
260         }
261         // High-rate logging WHEN CAPTURING goes to train_fp
262         if (cap && train_fp) {
263             fprintf(train_fp, "%llu,%.2f,%.2f,%.2f,%.2f,%s,%u\n",
264                     row.ts_us, row.melt_temp, row.inj_press,
265                     row.vib_amp, row.vib_freq, row.stage, row.failure_label);
266             fflush(train_fp);
267         }
268
269         tick++;
270         usleep(1000000 / cfg.hiRateHz); // sleep for hiRateHz
271     }
```

Figure 4: Decimation logic for sampling down from HF to LF data collection.

```
100     # hiRateHz - polling rate (Hz) (HF, for ML training)
4       # loRateHz - logging rate (Hz) (LF, for cloud storage)
15      # maxLogFileKB - max log file size before rotation
9500    # maxLogDirKB - max log file size before local deletion takes place (LF)*
10000   # maxTrainDirKB - max train file size before local deletion takes place (HF)*
```

* This would result in data not uploaded to AWS. It's the total size of the directory, not individual file sizes, which is different from maxLogFileKB which is for 1 file.

Figure 5: config.txt parameters

So for HF data we collect at 100Hz, it can be collected simply on each loop. For the lower frequency this is calculated with an additional variable tick. Tick is an unsigned long long (64 bits) that increments on each loop. We choose when to collect the low frequency by only calculating on tick mod decimateN. This calculation is key because a more simple mod calculation or a more basic counter that we originally used caused data collection at low frequency to come at inconsistent deltas from each other, especially when LoHz is not an even factor of HiHz.

To prevent SD card overflow and keep uploaded data a consistent size, we rotate and create a new CSV file once its size reaches maxFileKB – we set it to 15KB. This is a simplified buffering system that is key to the project's functionality. See Figure 4 for the file rotation logic for the buffered files, with Figure 6 as the actual rotation implementation. This is intended for

locations which may have inconsistent networking, so locally buffering the data makes the system network failure tolerant. If the device cannot reach AWS for cloud archiving, the data is preserved locally until a successful upload takes place, then the data is deleted from the SD card. In the future, we would like to implement this with a real buffer that is stored in RAM rather than on SD card. SD cards are not designed for continuously being read and written like this and can reduce machine life and cause storage failures, a better alternative would be implementing the buffer so it is stored in memory rather than on disk.

```c
// open a NEW file with timestamp prefix
static FILE* rotate_file(const char *dir, const char *prefix) {
    char fname[128];
    // name as logs/log_timestamp.csv or train/train_timestamp.csv
    snprintf(fname, sizeof(fname), "%s/%s_%llu.csv", dir, prefix, (unsigned long long)time(NULL));
    return fopen(fname, "w");
}
```

Figure 6: File rotation implementation

```
Every 0.1s: cat live_data                                  fedora-spectre: Wed Apr 30 10:34:00 2025

raw csv most recent data:
1745973055401014,168.23,50.39,0.86,14.62,PreInjection,0
2025-04-29 10:32:38, - Current failure probability: 0.04

Timestamp: 1745973055401014     Temp=168.2°C
Pressure=50.4psi        Vib=0.86g@14.6Hz
Stage=PreInjection

Config:
  HiHz          100
  LoHz          4
  maxLogFileKB  15
  maxLogDirKB   9500
  maxTrainDirKB 10000
  CaptureSec    900 (unused)
  CaptureEnable 0 (unused)
Capture mode: Training Mode (HF) and LF logging
Total Log dir usage:   1367KB / 9500KB (14% of cap)
Total Train dir usage: 2604KB / 10000KB (26% of cap)
Bytes/KBytes per second LF: 320B/s, 0.31KB/s
Bytes/KBytes per second HF: 8000B/s, 7.81KB/s
At current rates max cap will be full in:
    Log dir: 26235 seconds
    Train dir: 947 seconds
In order to fill the AWS S3 bucket (5GB), it will take 16777216.000000 seconds of LF data. (4660.337778 hours)
   The max time for the cron upload script should be: 30400 seconds to prevent reaching cap
   Current bufferred files: logs=92, train=2
   Next upload in: 245 seconds
```

Figure 7: View of live_data file for full system status.

ML Training

You can train the ML model by first collecting the high frequency data. This is done using a setup similar to writing to a gpio pin in class, by typing echo 1 > capture which will capture HF

data. This data is not subject to the maxLogFileKB size of 15KB, so the size of the CSV file for training data is only bound by the total size of the directory. This makes collecting and aggregating training data easier for us since the data is all in one file.

We use machine learning to predict simulated machine failures. The failure label is programmed as machine state: PartReplacement as seen in the ML trainer in Figure 8. For this we used a basic Random Forest classifier model and trained it on the csv data from the python machine simulator. We used this model as it is a light enough model and works well with the limited and mixed data we are working with. The trained model can predict if a failure in the machine is about to occur 10 seconds in advance. This was done by training the model to look at the metrics 10 seconds before a failure was found in the csv data, so when similar patterns are detected it can confidently say a failure is incoming.

```
# Find where PartReplacement happens and mark prior rows
for idx, row in df[df['stage'] == 'PartReplacement'].iterrows():
    failure_time = row['timestamp']
    early_window_start = failure_time - LOOKBACK_US
    # Mark rows in the 10s window before failure
    df.loc[(df['timestamp'] >= early_window_start) & (df['timestamp'] <= failure_time), 'failure'] = 1

num_failures = (df['failure'] == 1).sum()
print(f"Samples marked as failure: {num_failures}")
X = df[['temp', 'pressure', 'amplitude', 'frequency', 'stage']]
y = df['failure']
```

Figure 8: ML Failure labelling

ML Predictor and Notifier (predictive_maintain.service)

The notifier monitors the sensor data in realtime and uses the trained ML model to send an email notification when the probability of imminent failure is greater than 70%. See figure 9 for email notification showing risk of imminent failures. See figure 10 for the actual prediction calculation which is done after copying the failure_model.joblib file to the beaglebone.

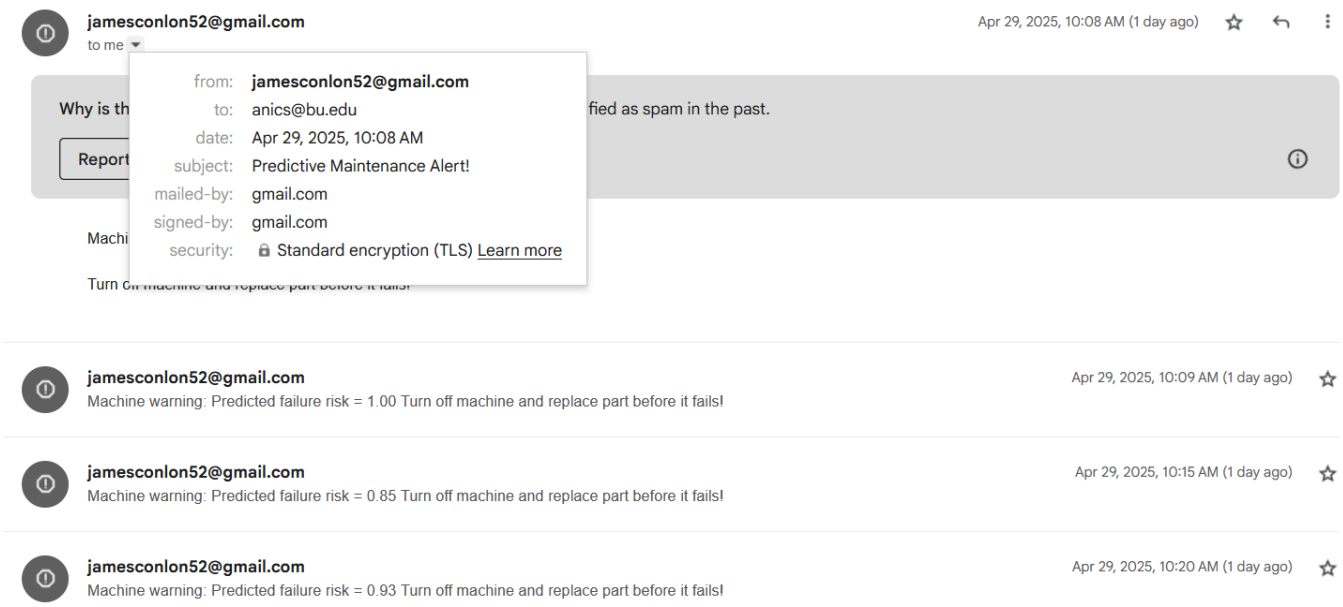

Figure 9: Email notification system

```
while True:
    data = get_live_data()
    df = pd.DataFrame([data])
    prob_failure = model.predict_proba(df)[0][1]
    print(f"Prediction: Failure probability = {prob_failure:.2f}")

```

Figure 10: On device machine learning detection

AWS/Uploader

The CSV data from the OPC UA logger is uploaded to an AWS S3 bucket. [7] (see Figure 13). This is done every 5 minutes on a cron job – although the timing of the cron job is not relevant for network usage. After the uploads are confirmed to be uploaded, the script deletes the buffered CSV data from the BeagleBone. It skips the most recent CSV as that is the CSV that is currently being written to. It is a key design of our device that it is network failure resistant. This is important as in real life many industrial machines are in rural areas (presumably because of cheaper land values), but those areas may have worse internet connectivity and reliability.

Remote Access

We include a Tailscale [8] exit node and subnet router so Users can ssh and reach OPC-UA and other devices that are isolated but still remotely accessible. Tailscale already runs on boot so that does not need to be set up, but we ensure that it shows as an exit node and is able to reach the OPC-UA network (subnet 192.168.50.0/24) with the following command. We are able to

connect over ssh to both the beaglebone itself and to the OPC-UA python server with the new exit node in place as seen in Figure 11.

```
sudo tailscale up \
  --hostname beaglebone \
  --advertise-exit-node \
  --advertise-routes=192.168.50.0/24
```

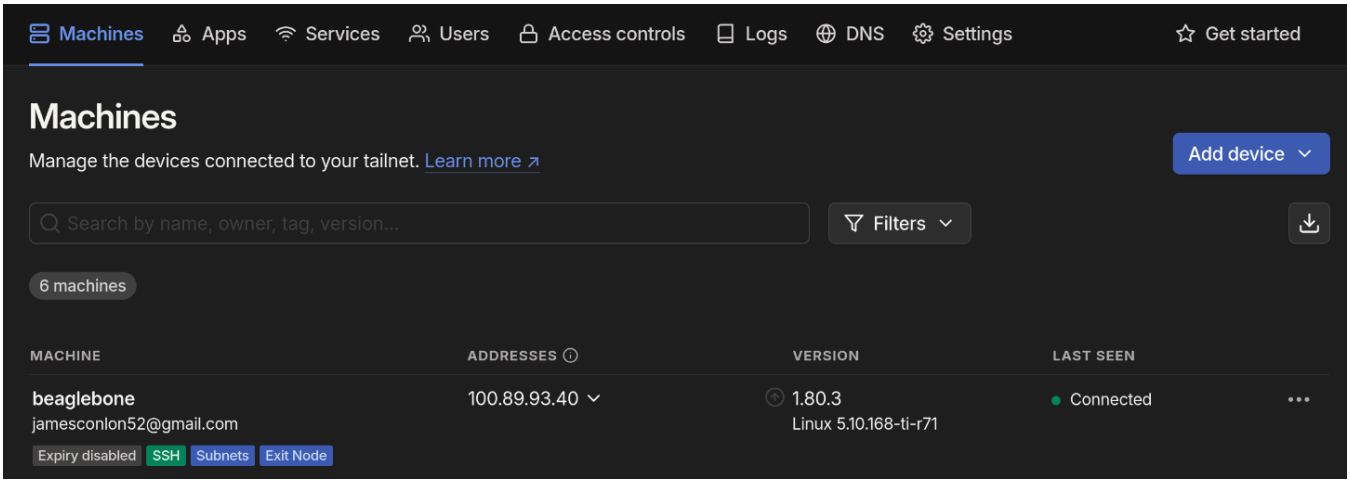

Figure 11: Tailscale admin page

Data Simulator (injection_moulding.py)

This simulates a number of data points in a 5-stage full state machine we modelled. The stages are Pre Injection, Injection, Holding, Cooling, and Waiting. [1] It generates noisy data by ramping between stages and simulating periodic data. Our project scope originally included visualization of different time-series data points (which is why we bothered to model periodic and ramped data points), but we pivoted to focusing on ML and predictive maintenance, so most of the data we generate is just filler. The data points we expose via OPC-UA in python are: timestamp, temperature, injection pressure, vibration frequency and amplitude, and stage.

In order to simulate data suitable for predictive modeling, we introduce a 20% chance of an incoming failure, which will happen within the next 3-8 states in the FSM. When that incoming failure takes place, we implement a drift during all stages that adjusts the vibration amplitude data point. This isn't a calculation based on the reality of these machines, it is simply a data point that is easily detectable by a ML model. See figure 12 for the simulated drift calculation. It's important that it is done cycles in advance with the var machinePartFailureImminent so it starts to drift towards the failure label early– this is key for allowing our ML model to detect the drift in Amplitude in advance of the failure.

```
171         vibration_amplitude = simulate_ramped_sensor_value(SENSOR_RANGES[state]
            ["vibration_amplitude"], current_time, stage_start, base_duration, "constant",
            anomaly=anomaly_vib)
172         if machinePartFailureImminent:
173             drift = 1 + DRIFT_PERCENT * ((current_time - stage_start) / base_duration)
174             vibration_amplitude *= drift
```

Figure 12: Drift calculation on Amplitude

Amplitude was chosen as the data point for ML after findings in a paper [9] that "vibration data from mining equipment is used to evaluate the operational state of the machinery and triggering alarms when anomalies arise," and we have found that vibration is a very common early predictor of failure that would be otherwise unnoticeable by a human machine operator. This is a very basic simulation, but the core of the project is the fact that this concept would be expandable to many more data points for ML and more data points for collection.

Results / Conclusion

The system did not fail overnight and continuously logged data. Every 5 mins it sends good sized files as defined in the cron job. See Figure 13. In the morning after being left to run overnight, there were no excess buffered files and systemd logs showed no crashes of the main program. Those two results mean the program is safe to run indefinitely with no memory overflowing issues or SD card overflowing issues over a long period. Even if the network goes down on the beaglebone, it will automatically delete older files over the storage quota to prevent overflowing the SD card.

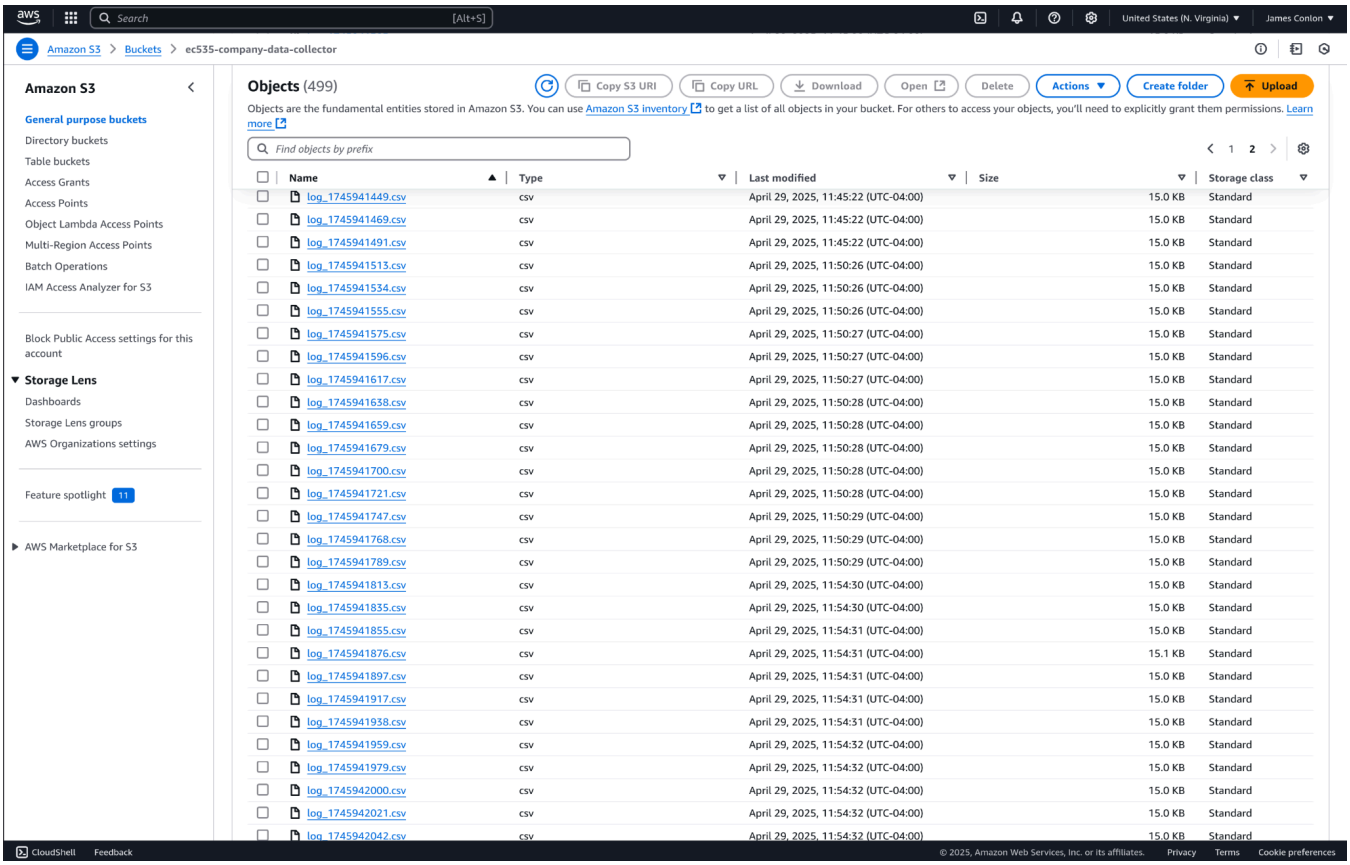

Figure 13: AWS S3 Bucket showing 5 minute cron job results.

One issue we noticed with the code is that the timing file estimates were not as accurate– because we estimated based on the size of the row in C rather than the size of row in csv.

The ML model was able to predict failures ~10 seconds in advance most of the time. We measured about 41% of emails received (12/29 total) from our model as false positives (where no machine failure was imminent). In practice, a false positive is acceptable (and preferable) to a false negative, because as we mentioned, replacing a part early is better than allowing it to fail. In general, we don't think it's possible to improve much on the accuracy rate or the forecast time to greater than 10 seconds without modifying the data simulation itself to show "warning signs" of failure far further in advance than the 3-8 cycles in advance we currently do. This is doable but not super useful to anybody since we do not have a real machine to collect failure data from – the amplitude drift we implemented is only a proof of concept of what a real machine could do.

Future scope and areas of improvement

The project presents and demonstrates an idea for predictive maintenance and data logging. The goal of our project was to fill the gaps of existing data loggers and expand them to be used for predictive maintenance. Future scope would be improving the workflow for ML training as it is currently very manual. We also did not tune the ML model very well as we do not know how to, nor do we know true warning signs of failure used in real life predictive maintenance applications. The use of Amplitude drift was arbitrary based on one paper and we would like to get more concrete examples of this. The accuracy of the model is not as relevant to the scope of the project, the project is about running the predictive maintenance on the "edge device" which is the BeagleBone.
Some areas that can be improved are:

- Making the data points configurable in the config.txt instead of hard coding them in the binary
- Making the ML model more accurate and maybe using TinyML models that can allow faster deployment and predictions on hardware such as the beaglebone.
- More realistic data simulator or using data from real machines.
- Move the HF buffer from SD to memory so logging doesn't wear on the SD card.
- Better frontend interface.
- Run on a multithreaded CPU (not beaglebone) so no risk of blocking threads between data collector and ML.

Technical Skills Learned

Learned how to cross compile with an included, also cross compiled and statically linked library (open62541) for shipping to the BeagleBone. We found systemd to be a good way to keep programs running and an easy way to keep a python script fail tolerant on an embedded system, as it is able to auto restart if it goes down. We also learned how to train a lightweight ML model and deploy on a low powered device like BeagleBone.

References

[1] *The Cycles of the Injection Moulding Process*, Automatic Plastics. [Online]. Available: https://www.automaticplastics.com/the-cycles-of-the-injection-moulding-process/

[2] *Asset Optimization: Predictive Maintenance*, Deloitte, 2022. [Online]. Available: https://www2.deloitte.com/us/en/pages/operations/articles/predictive-maintenance-and-the-smart-factory.html

[3] Froehlich, A, "How Edge Computing Compares With Cloud Computing," *Network Computing*, 2018. [Online]. Available:

https://www.networkcomputing.com/cloud-networking/how-edge-computing-compares-with-cloud-computing

[4] *Industrial Edge Computing Megatrends: Special Report* [Online]. Available: https://iebmedia.com/technology/edge-cloud/industrial-edge-computing-megatrends-special-report/

[5] *open62541 – Open Source OPC UA*, GitHub. [Online]. Available: https://github.com/open62541/open62541

[6] *UaExpert – OPC UA Client*, Unified Automation. [Online]. Available: https://www.unified-automation.com/products/development-tools/uaexpert.html

[7] *Amazon S3 – Cloud Object Storage*, Amazon Web Services. [Online]. Available: https://aws.amazon.com/s3/

[8] *Tailscale – Zero config VPN*, Tailscale Inc. [Online]. Available: https://tailscale.com/

[9] "Enhancing Predictive Maintenance in Mining Mobile Machinery through a TinyML-enabled Hierarchical Inference Network" Nov. 2024. [Online]. Available: https://arxiv.org/html/2411.07168v1#S1