CS 437 IoT Final Project Report

Zachary Moulton / moulton6

Demo Video:

https://youtu.be/43KvL3e-f_Y

Source Code:

https://github.com/moults31/hydra

https://github.com/moults31/hydra-jwt-updater

https://github.com/moults31/hydra-gsheet-permission-updater

0. Table of Contents

1. Motivation

I have designed and implemented an all-in-one sensor for homebrewing. It tracks three phases of the brewing process: Coldcrash, Primary Fermentation, and Secondary Fermentation. In these three phases, we can benefit from a combination of light and temperature data to inform us about the quality of our brewing process.

Coldcrash:

Also known as wort chilling. This phase occurs immediately after the “boil” stage in brewing, and is the step where the liquid is forcibly cooled. For context, here is an article about why this process is important. In short, the liquid must be within a certain temperature range in order for the yeast to begin fermentation properly, and it is important to reach that temperature range as quickly as possible to limit the possibility of contaminants entering the batch.

My IoT design intends to track the temperature-over-time curve, a feature which no device on the market offers. This provides the brewing team with both 1. the ability to stop coldcrash as soon as the target temperature range is met, and 2. historical data logs to enable experimentation to decrease the cold crash duration, reducing the likelihood of contaminated batches. This is represented in requirement R-1 in my Project Proposal.

Primary Fermentation:

The simplest way to describe this phase is to say we have yeast consuming sugar and producing alcohol and CO_2. We desire a “stable” fermentation so that the yeast will produce only those desired products, and no undesirable (or even hazardous) byproducts. Some conditions that can affect “stability” of the fermentation process are temperature (reference) and light intensity (reference).

My IoT design intends to log timestamped samples of both temperature and light over time. And crucially, it intends to warn the brewing team via an Asana comment (that the users can receive push notifications for on the Asana iOS/Android app) in the event that temperature or light exceeds the configured threshold. These features are represented in requirement R-2 in my Project Proposal. The benefit that these features provide to the brewing team is twofold: 1. The warnings allow immediate action to be taken to fix a poor fermentation environment, and 2. The data logged allows the brewing team to compare and contrast brewing environments across batches to draw conclusions on causality of beer quality.

Secondary Fermentation:

This phase is functionally the same as primary fermentation from the perspective of my IoT project. However, since it is a distinct step in the brewing process, it is important to track it separately. My design therefore provides the same features here for the same reasons, but clearly organizes its data and warnings into Primary and Secondary categories.

Market Research:

Here is a brief discussion on the niche that my IoT project fills, which no other device available does. Let’s revisit the table below that appeared in my Project Proposal.

	Temp. Sensing	Gravity Sensing	Cold Crash Tracking	Light Sensing	Asana Integration
Proposed Device	√	*R-3 (optional)*	√	√	√
FLOAT	√	√			
Tilt	√	√			
PLAATO	√	√			
iSpindel	√	√			

As shown, my device provides 3 new features that no other device does. The value that those features bring to the brewing team was clearly explained earlier in this section. Note that the "Gravity Sensing" feature is shown as *R-3 (optional)* here since that feature represents the optional requirement R-3 which was time-prohibitive due to mechanical hardware complexity as predicted in my Proposal, but which I can add to my prototype in the future as a passion project.

Additionally, I reached out to the proprietors of Tilt and FLOAT with my Project Proposal, soliciting opinions and advice. Both companies graciously offered general design advice, and discounted hardware prices to help with prototyping. I ended up taking up Tilt on their hardware offer since they are based in North America while FLOAT is in Europe. My final prototype is housed in the water-tight Tilt tubes, and I also benefited from having a Tilt hydrometer on-hand to correlate my sensor data against.

2. Technical Approach

2A. Hardware Design

In this section I will describe and justify the design of my hardware (both electrical and mechanical).

The electrical design of my project was fairly straightforward. Based on my motivation, I knew I needed only a controller of some sort, a power source, an ambient light sensor, a temperature sensor, and a gyro sensor. I decided to select I2C to have all of those devices communicate, since it allows daisy chaining which greatly reduces the complexity of the circuit compared to SPI. The overall circuit design is shown in Figure 1.

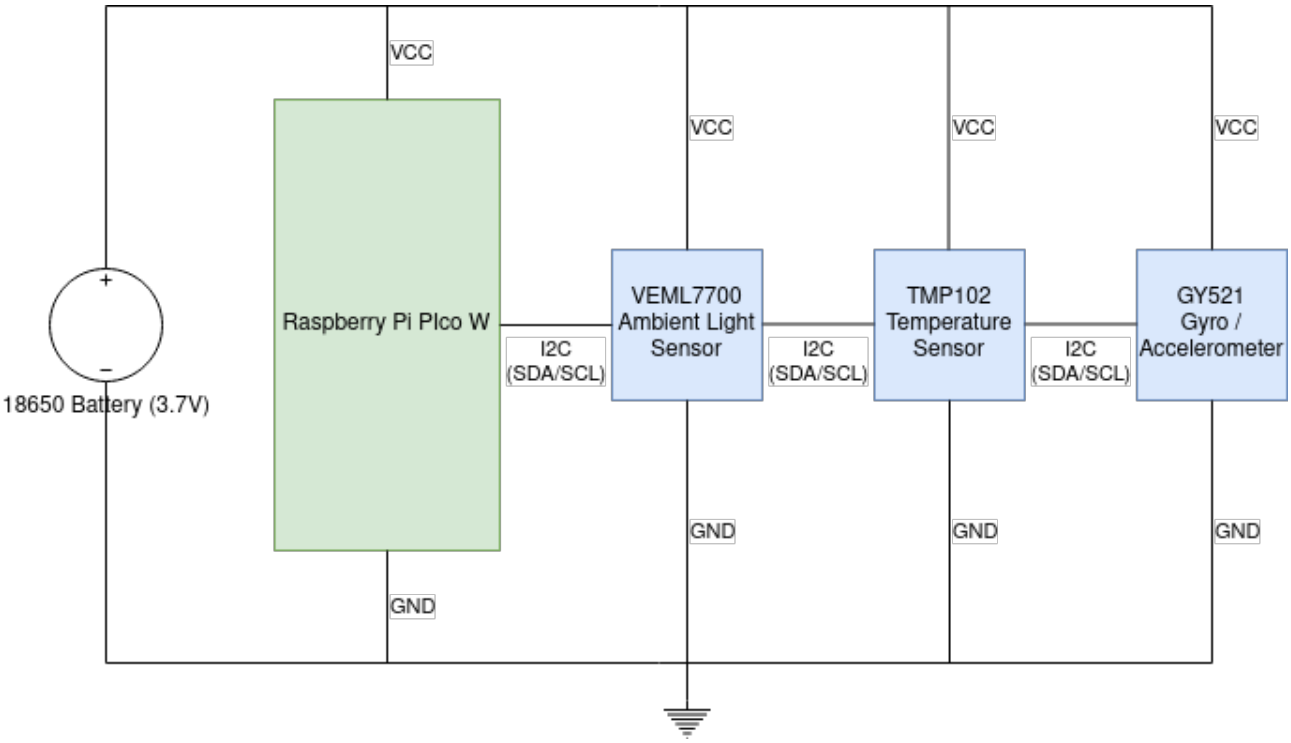


Figure 1: Circuit Diagram

The following are descriptions and justifications of the components I selected.

Raspberry Pi Pico W: My main motivations for choosing this component as the controller were: price/availability, power efficiency, built-in Wi-Fi, and Micropython support. Price/availability: This was readily available at my local electronics store for $15 CAD unit price, so I was able to buy 3 for redundant prototypes. Power efficiency: Unlike traditional Raspberry Pi products, this device does not run a general-purpose OS. It runs a user application directly on the metal, more similar to an Arduino. This allows for deep sleep as well as a generally lower power draw while awake. Built-in Wi-Fi: I knew I would need Wi-Fi to communicate directly with the services I had in mind, which I will discuss in detail in the Software Design section. Micropython support: This was an intriguing learning opportunity for me since I am fairly experienced in implementing embedded programs in C/C++/Arduino and instead wanted to get some first-hand experience in the popular up-and-coming alternative that is Micropython. So, I chose a controller that supported that.

VEML7700 / TMP102 / GY521: My motivations for selecting these were all the same. I required I2C and support for a 3.7V power source. Aside from that, I chose the parts with the best price and availability. I did not place emphasis on existing Micropython drivers, since I wanted to develop those myself anyway.

18650 Battery: I chose this battery as a power source since I had them (along with a charger) on-hand from Lab 1, and extra holders for them were readily available at my local electronics shop. Two AA batteries in series would have been an acceptable power source as well.

2B: Software Design

Requirements: In this section I will describe how I designed a software solution to run on the aforementioned hardware to satisfy the requirements I laid out in the Project Proposal. For reference, the requirements, verbatim, were:

R-1: *"Cold Crash Tracking"*. Device shall be inserted into the kettle immediately following the boil. User shall specify final "target" temperature for cold crash. Device will notify user immediately when target temperature is reached, and report temperature vs time curve on relevant Asana task.

R-2: *"Light and Temperature Warning"*. Device shall be inserted into primary or secondary fermenter immediately after beer is. User shall specify warning values for both temperature and light intensity. Device must track both temperature and light intensity periodically and store them in CSV with timestamps. If light intensity or temperature exceeds warning value, device must notify user immediately on Asana.

Design: Figure 2 shows two high-level components of my software system design: the main program, and the libraries. I will describe them in detail in the following paragraphs.

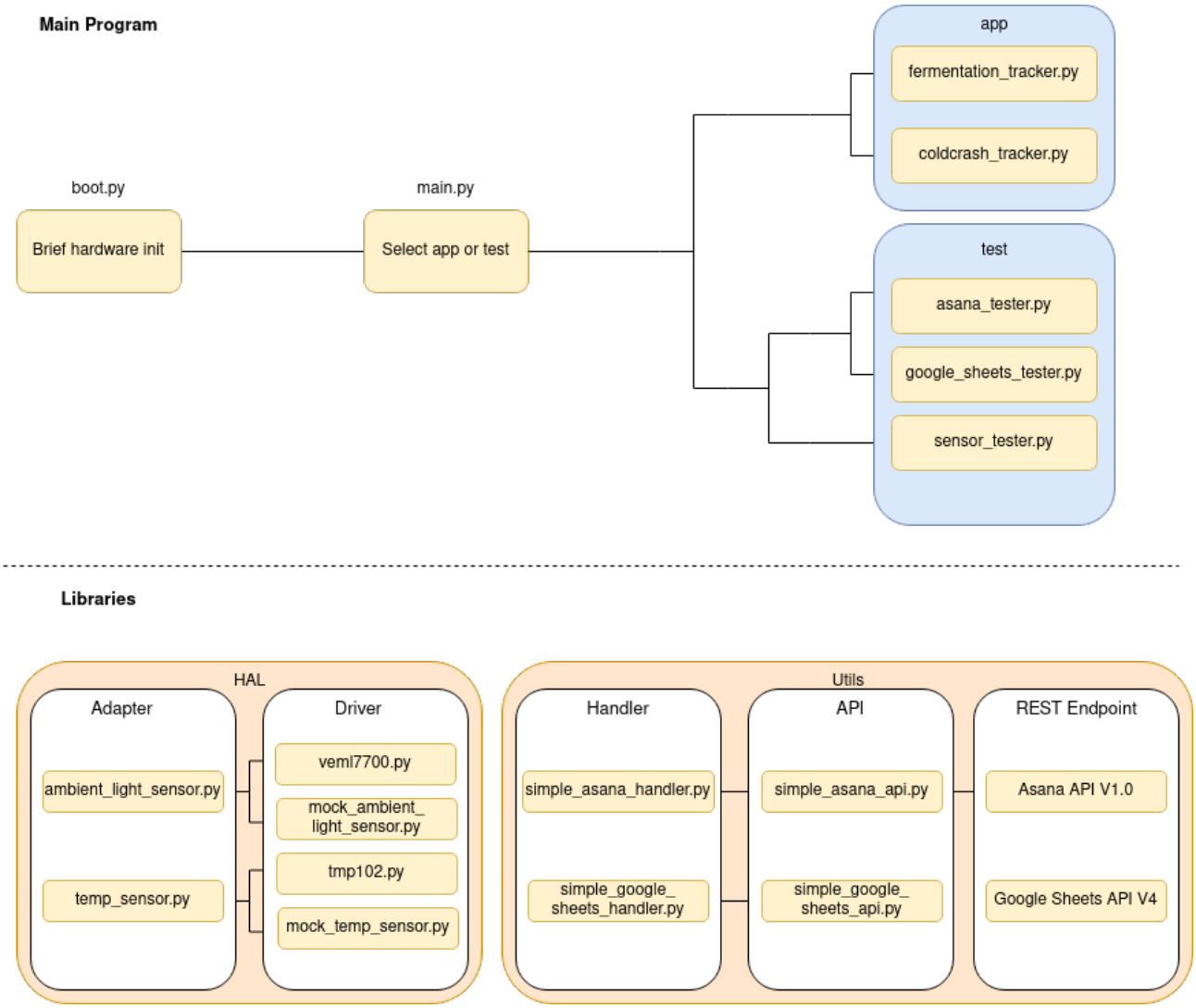


Figure 2: Software Design

Main program:

boot.py: The Pi Pico requires that a file named boot.py be placed in the root of the Micropython, which is executed automatically on power-up. My boot.py implementation simply checks if the micro-USB serial port is connected, and if not, jumps to main.py. The reason for not jumping to main if USB is connected is so that the interactive micropython interpreter can be used instead in that case.

main.py: This file's purpose is to choose which actual program will run. The programs that I have designed and implemented are grouped into "app" and "test". The "test" programs intend to test a particular library module, and throw an exception if it is not working. The "app" programs are the actual applications that I have designed to satisfy the requirements R-1 and R-2. The user can choose to force a specific program, but the default behavior I designed is to check for Asana tasks assigned to the MAC address of the particular Pico running, and start either coldcrash tracking, primary fermentation tracking, or secondary fermentation tracking based on the "section" that the

assigned task is in. Once decided, the program will also comment “On it!” on the Asana task that it found assigned to it.

fermentation_tracker.py: Figure 3 below shows the high-level flow of fermentation_tracker. The fermentation tracker can be initialized in either “primary” or “secondary” mode, which has a meaningful difference to the brewing process but not as much to the tracking process. For the sake of tracking, the only difference is that the data will be logged to the appropriate page (either named “Primary” or “Secondary”) within the Google spreadsheet. The initialization process proceeds as shown in the Init block of Figure 3. The “predetermined active task” refers to the Asana task that main.py discovered that is assigned to the currently running Pico. The act of “read or create Google Sheet link” means that the aforementioned active Asana task’s description should be a link to the Google Sheet where the light and temperature data over time will be stored. This action will check whether a valid Google sheet link exists in the description, and store it for use if so, or go to the Google API to create a new one if not, and write it into the Asana task description once it’s created. Then, in a given iteration of the main loop, the fermentation tracker program will read the temperature and light sensors, post a comment on the active Asana task with a warning if the threshold is exceeded for either parameter, and then upload the sensor values with the associated timestamp to the proper page in Google Sheets. Then, in between iterations, the Pico should sleep for a long time. The scale of fermentation tracking is on the order of weeks, so the device should enter deep sleep in between readings, and it should be reasonable to sleep for up to an hour between readings without impacting data quality. The stopping condition for the main loop does not exist since it would be specified by R-3 which was an optional stretch goal. Therefore the fermentation tracking process can be manually stopped by powering off the device at the scheduled fermentation stop date.

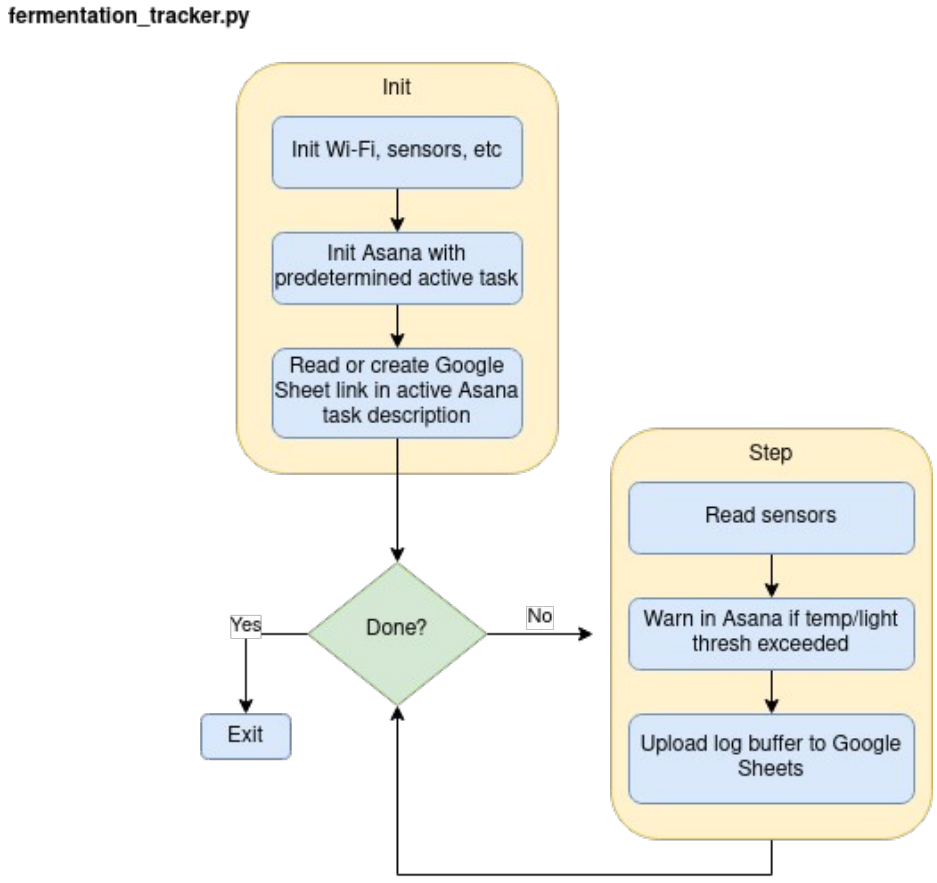


Figure 3: Fermentation Tracker

coldcrash_tracker.py: Coldcrash Tracker is very similar to Fermentation Tracker, except for two major differences. Firstly, we want to know as soon as the target temperature is reached, so we should sample no less frequently than 30 seconds. Secondly, in coldcrash we are only interested in temperature, so we will not read or log the ambient light sensor values.

Libraries:

HAL: HAL stands for Hardware Abstraction Layer and provides an API that the application layer can call to get abstract sensor values without the app layer needing to know about sensor driver implementation details. In other words, it allows the app layer to call something like temp_sensor.read() instead of needing to call tmp102.read(). Therefore, if one of my prototypes was hooked up to a non-tmp102 temperature sensor, my application layer would not need to change at all in order to use it. In my experience this approach has generally been helpful for large-scale products that ship to many customers and therefore need multiple sources for hardware components. But, I noticed that almost all components of all types are liable to go in and out of stock with no rhyme or reason in this day and age, so I felt that this software design would keep my project resilient to the whims of the supply chain.

Utils: This was the name I came up with for my module that wrapped functionality and communication with 3rd-party REST APIs (Asana and Google Sheets). I split *utils* into a *handler* layer and an *api* layer. My *handler* layer handles all of my application-specific functionality, and my *api* layer wraps REST API endpoints from the given service in micropython syntax without providing any additional functionality. Finally, the *REST Endpoint* layer in Figure 2 represents the actual REST API endpoints that my *api* layer communicates with via HTTP.

3. Implementation Details

3A. Hardware Implementation

In this section I will show and describe the prototype implementations I made of the circuit design in Figure 1.

Breadboard prototypes: These prototypes allowed me to validate that the Pico and the sensors were compatible with each other, and allowed me to begin working on the software before necessarily having a field-ready hardware prototype. The implementation of these was very simple: it only required hooking up VCC, GND, I2C SCL, and I2C SDA. In both of these prototypes I was able to swap between powering everything with either an 18650 battery or the onboard micro-USB port at will. To save time I did not hook up every sensor to every breadboard prototype, but made sure that all sensors were tested on at least one breadboard.

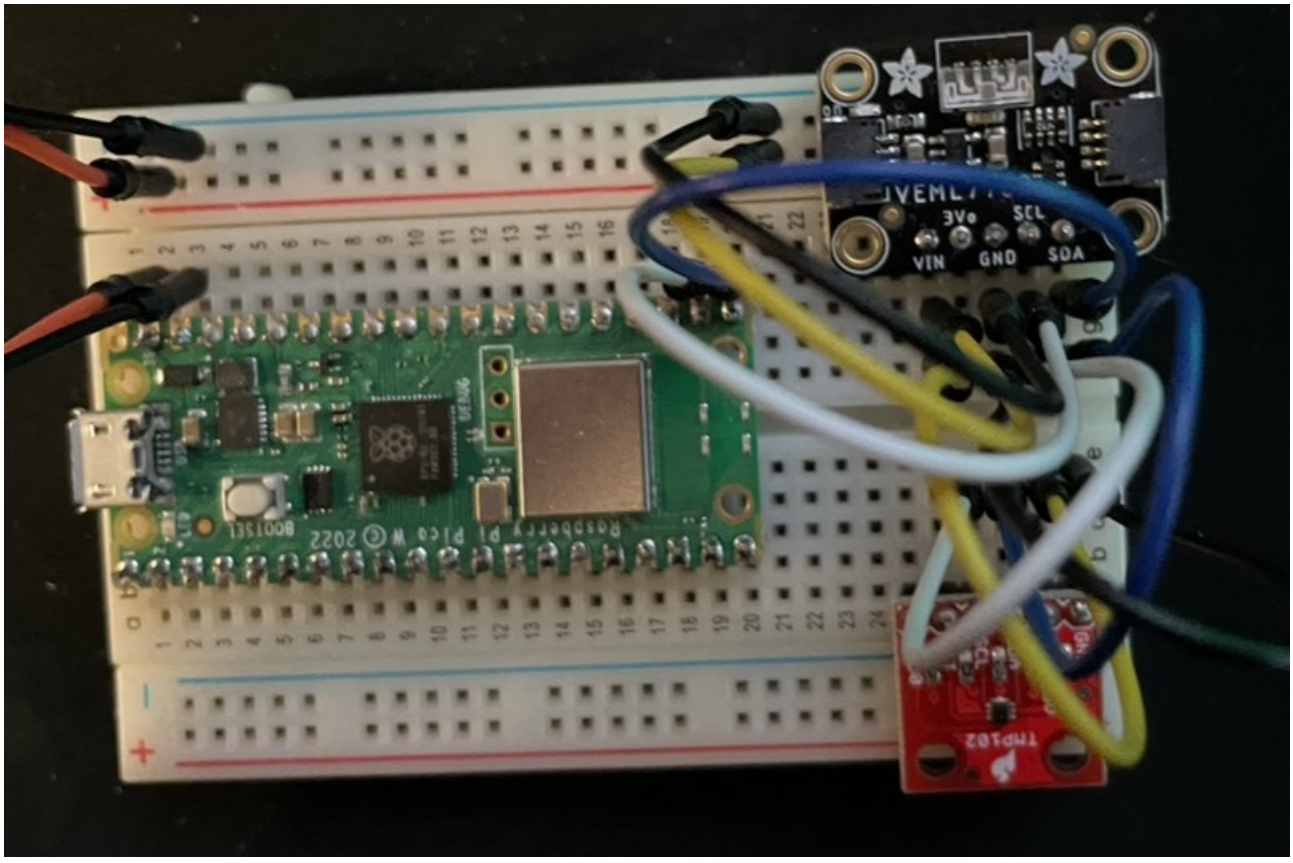


Figure 4: Breadboard Prototype 1 (No gyro)

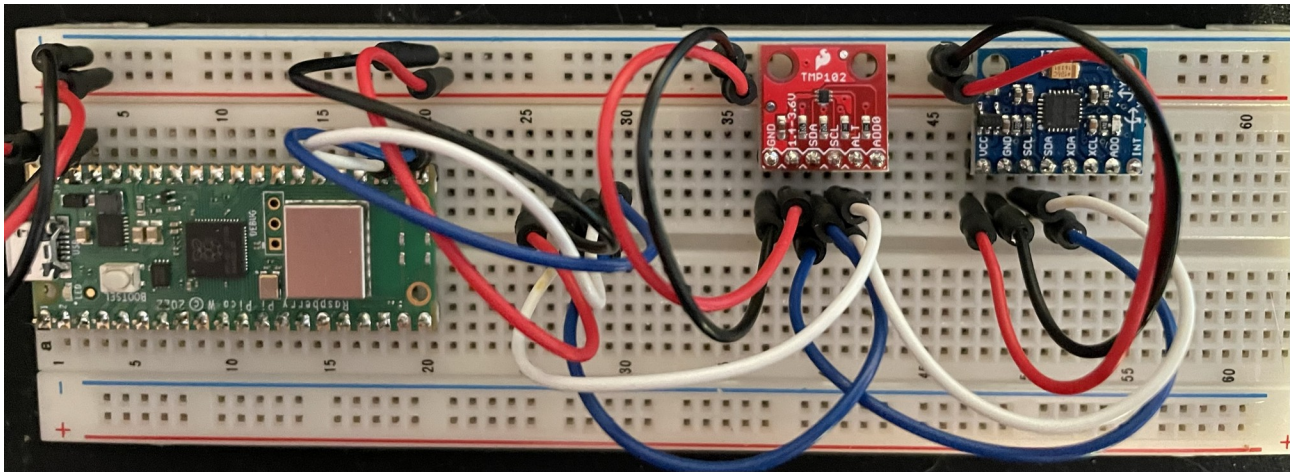


Figure 5: Breadboard Prototype 2 (No light sensor)

Soldered Perfboard Prototypes:

These prototypes were intended to be used in the field; to be the implementations that actually met requirements R-1 and R-2.

Mechanical Details: I received some empty Tilt Hydrometer tubes from the gracious folks who run that company, and implemented my hardware to fit inside of those tubes. For a perfboard to fit in the tube, it would need to be approximately 5”x1.5”. I was unable to find perfboards close to that size, so I ordered larger ones and cut them to size. I did so by etching a straight line on them using a hobby knife and a ruler, and then snapping them along the straight line. I also needed to make sure the components on the board did not extend too far in any direction so as to make the assemblage larger than what the tube could fit. I had never done anything like this before, so this was an exciting and satisfying learning opportunity.

Electrical Details: All of the connections between the components and on the perfboard were executed using solder bridges and jumper wires. I have done some soldering in the past, but never had to make clean, precise solder bridges between unconnected pads like this, which presented a challenging but rewarding learning opportunity. On Perfboard Prototype 1, the battery holder was connected via a detachable JST connector to allow powering on/off. Since the light sensor needed to be exposed to the non-submerged end of the tube, I also connected it via long wires and a JST connector. On Perfboard Prototype 2, the battery holder was soldered directly onto the board to save space radially in the tube, and I added a switch instead to control power. Also, instead of using long cables and a JST connector for the light sensor, I bent the light sensor’s header pins 90 degrees using pliers and then soldered them directly to the end row of through-holes on the perfboard.

Figure 6: Perfboard Prototype 1 Front

Figure 7: Perfboard Prototype 1 Back

Figure 8: Perfboard Prototype 1 End

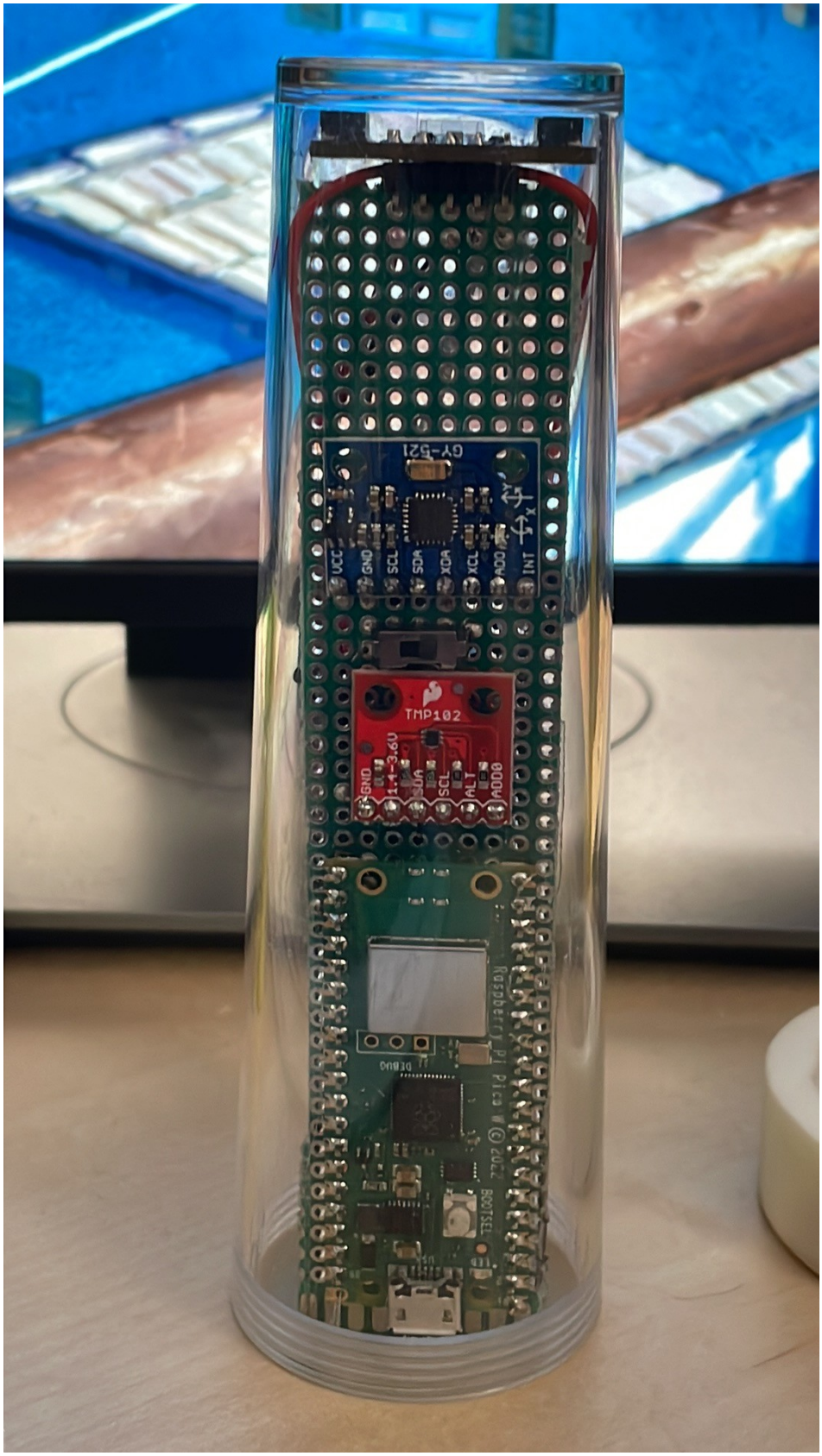


Figure 9: Perfboard Prototype 2 Front

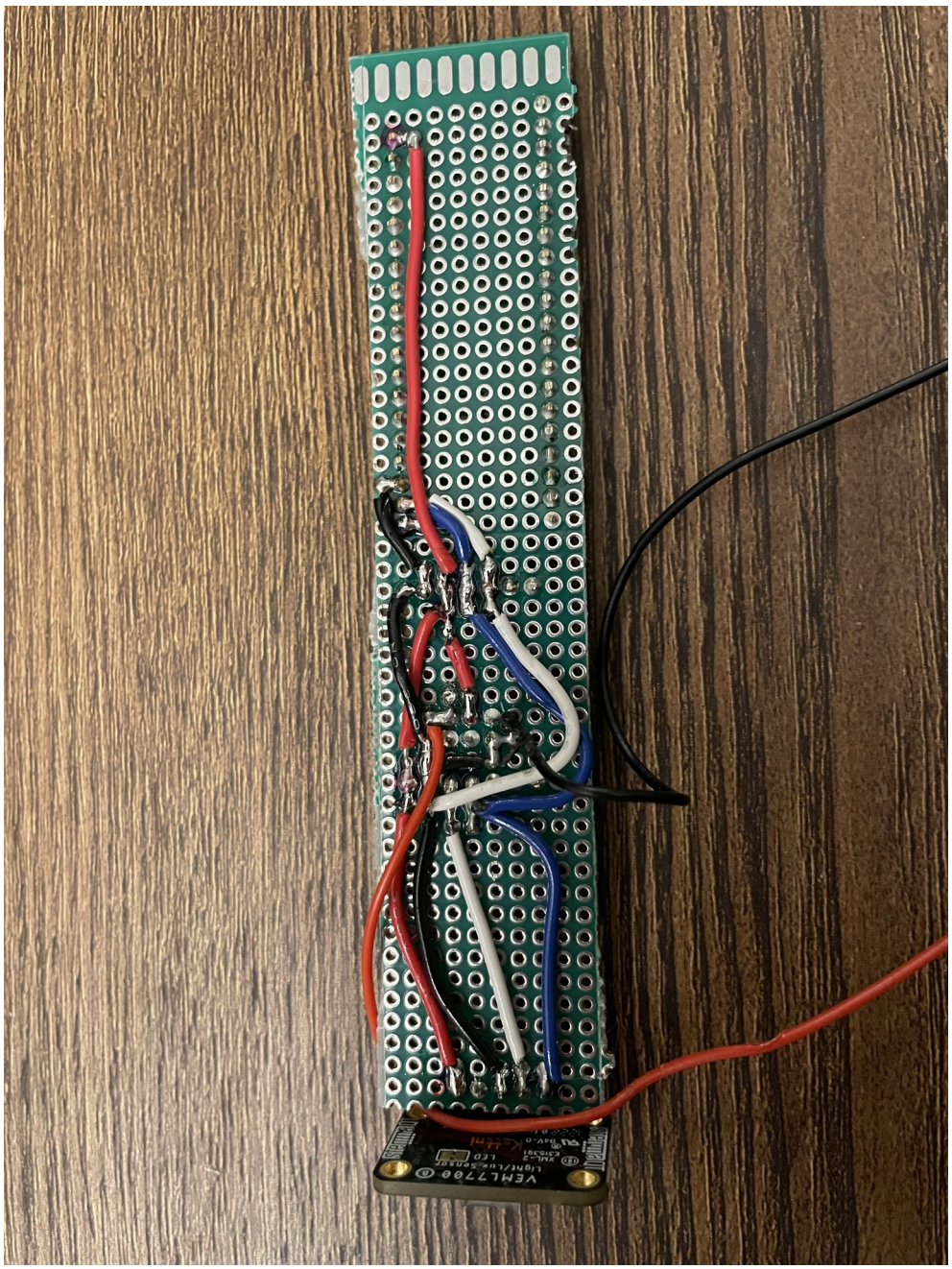


Figure 10: Perfboard Prototype 2 Back, Showing Solder Bridges

3B. Software Implementation

Overall, I was able to implement the design laid out in section 2B (Software Design). My source code in the “hydra” repository is well-documented and matches the structure of the diagrams in section 2B, which means that the interesting aspects of my software architecture were already discussed in that section. Therefore I will use this section to discuss the more interesting aspects of the implementation where I had to be resourceful to overcome challenges.

Debugging on UNIX:

Since the Pi Pico’s CPU frequency is only 133MHz, a single “debug shot” can be significantly slower than a program running on a desktop CPU like computer science students (such as myself) might be used to, whose frequency is commonly in the 2-4GHz range (over 100x faster than the Pico). So, I decided to make all of my code portable to the Micropython UNIX interpreter. To do this, I used sys.platform to detect whether the platform is “linux” or not, and if so, made sure not to try to use the Pi Pico Wi-Fi chip or my I2C sensors. This meant of course that I could not get any meaningful temperature or light readings when running the UNIX port, but by creating the “mock” sensor drivers I was able to still generate fake sensor values that allowed me to make sure that the software modules were all integrated properly. This saved me a ton of time when debugging modules like my Asana and Google Sheets API implementations, and the flows between them, since it was much, much faster than testing on the Pico hardware.

Third-party APIs:

Implementing Micropython versions of the Asana and Google Sheets REST API wrappers was an interesting endeavor. Both of these services provide pre-packaged, user-friendly Python3 API wrappers as one would expect. However, in both cases, there were dependencies such as urllib3 or rsa that there are no Micropython implementations for, neither official nor third-party. So, I decided to implement my own minimal wrappers (using “simple” in the filenames to indicate a simplified feature set). I used the same model in both cases. My model was to create an “api” file that simply wraps the REST API endpoints I need, using Micropython’s built-in urequests module, which provides a syntatically-compatible subset of the functionality of the Python3 requests module. The, I created a “handler” file that implemented the application-specific flows I needed, using the primitives in the “api” layer. This approach worked quite well, and showed that a good, clean, simple design can save a lot of work (and dependencies!).

Garbage Collection:

Micropython garbage collection turned out to be one of the greatest debugging challenges I faced. I assumed that it would have been handled automatically like in Python, but this turned out to not be true, and presented a unique learning opportunity. When testing my overall application, I faced a failure mode where the device would suddenly hang inexplicably: it would stop logging to Google Sheets, and the onboard LED would stop flashing, indicating that it was no longer making iterations of the main loop. Note that in the field testing, I had essentially no debuggability: I had no serial console connected since it’s incompatible with deep sleep, and the onboard memory is only 2MB, so not big enough to redirect STDOUT to. I ordered a SPI-compatible microSD breakout board, but ended up solving the bug before it arrived, so never hooked it up.

I first suspected that maybe this hang was caused by an unknown exception, so I wrapped my **step** function in a try/except that catches BaseException, so no exception could cause a full-on hang.

However, the hang persisted. So, I thought about garbage collection – maybe this is an out-of-memory error? I devised an efficient logging scheme to write the output of gc.free() to a file at a few spots throughout the main loop. I came to find that some urequests calls actually did fail to get garbage collected! And, since the Pico's RAM is on the order of ~150kB, if we miss garbage collection after even a single http response, we come dangerously close to out-of-memory immediately.

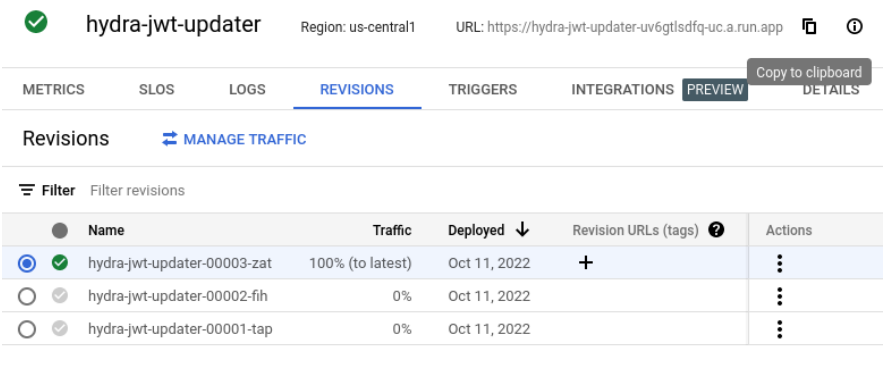
So, my solution was to place gc.collect() calls strategically wherever an http response might get stored, which turned out to be a direct and proper fix for the hang I was facing. This was an interesting first-hand learning experience dealing with one of the drawbacks of HTTP in IoT: it is a lot heavier-weight than some other communication protocols such as MQTT.

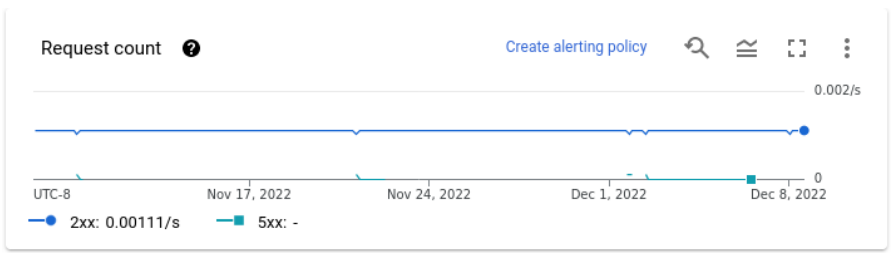
RSA Keys:

The problem here was that Micropython provides no way to create an RSA-signed JWT, which Google Sheets API requires (and requires a new one every hour or so)! So, I solved it by using the spirit of IoT: a cloud-based microservice (my source code here). I realized that Micropython doesn't actually need to generate these keys itself, it just needs a way to obtain them. So, I created a microservice that uses Google's Python3 wrapper to create the key and simply push it to a specific place in my Asana board, frequently enough that the latest one is never expired. Then, whenever my Pico edge device wants to use the Google Sheets API, it can just grab the latest key from Asana first.

Creating this cloud microservice was an exciting learning opportunity since I had never used cloud services like AWS or Azure or Google Cloud before, and I had not started Lab 3 yet. So I blazed my own trail and used Google Cloud Run, and implemented my microservice as a Dockerized Flask server, triggered by Google's built-in Cron scheduler, on a 15-minute interval. It worked like a charm, and ran for two months with no downtime while managing to stay in the Free pricing tier.

If I had completed Lab 3 before implementing this part, perhaps I would have chosen AWS and implemented this service as a lambda function to run on my Raspberry Pi 4B as a "Greengrass core." MQTT may have been a more elegant solution than using Asana as a mailbox. In any case, it worked!

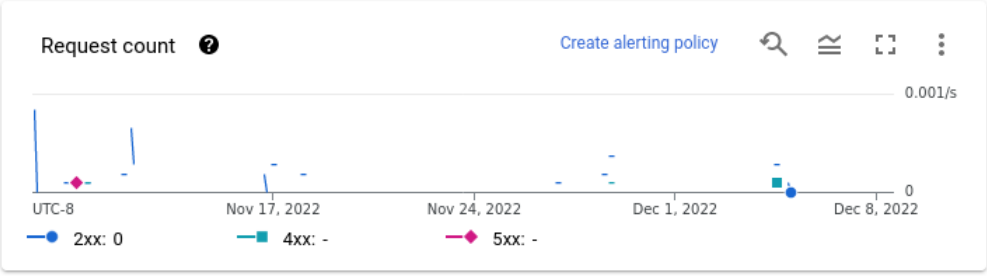




Google Sheets Permissions:

Here I created another microservice (source code here) to solve a problem which frankly feels like an oversight on Google's part. For context, a headless account that can use Google APIs (such as Sheets) is called a service account. When a service account creates a file, it creates it in its own Google Drive, and no humans have access to it! Not even the human who has admin access to the service account! The only way to grant any human access to such a file is to make a call to the Google Drive API which is notably a different thing than the Google Sheets API (and does not have a Micropython-compatible wrapper). So, instead of creating a whole "api" and "handler" layer in Micropython, I decided to simply create a microservice that uses the Python3 Google Drive API wrapper to log in as my service account, and share any files in its Google drive with a configurable list of email addresses.

To implement this microservice I again created a Dockerized Flask server and deployed it in Google Cloud Run since I already had some comfort with that model from hydra-jwt-updater. The major difference was the trigger: Instead of using a cron scheduler, this time I exposed the microservice URL to the public internet, so that my edge device could make a request to this service immediately after creating a new spreadsheet that needs to be shared. This asynchronous model worked very well for this application. The request count metrics are expectedly a lot more sporadic than hydra-jwt-updater, as shown below.



Deep Sleep:

As mentioned in section 2A Hardware Design, I considered "deep sleep" (in other words, shutting down peripherals like Wi-Fi, I2C, etc) to be mandatory in this application. I realized that even though Micropython has deep sleep built in, that version of deep sleep turns off RAM as well and therefore causes a system reset, which is not ideal in my application. My design would have to make at least two HTTP requests on wake to figure out what mode it's in and what it should do next, which would be a waste of time and power. Therefore I used this open-source micropython fork which provides "picosleep," a deep sleep implementation that does not turn off RAM and therefore allows execution to resume in-place on wake.

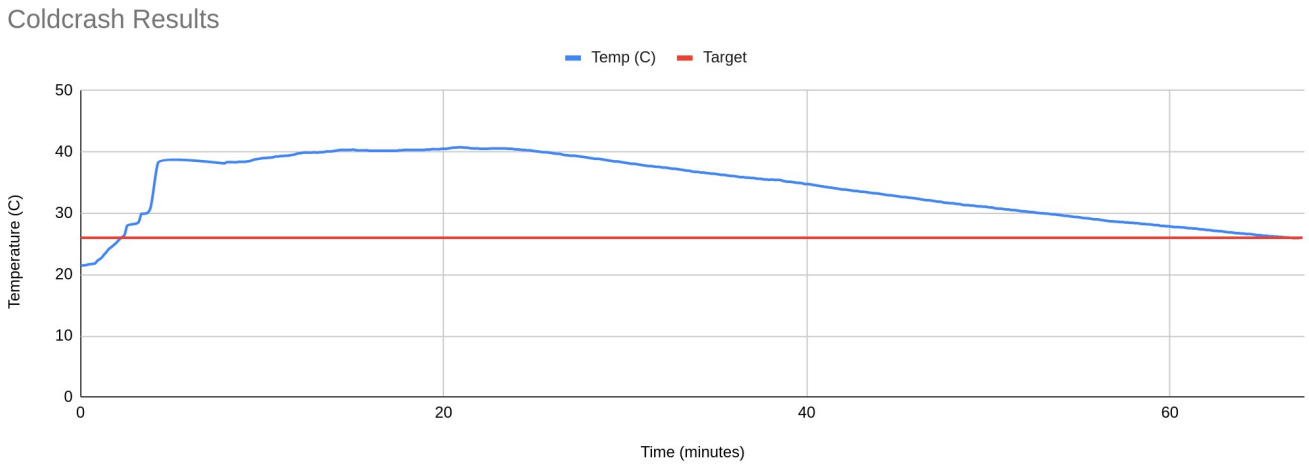
4. Results

… It worked!

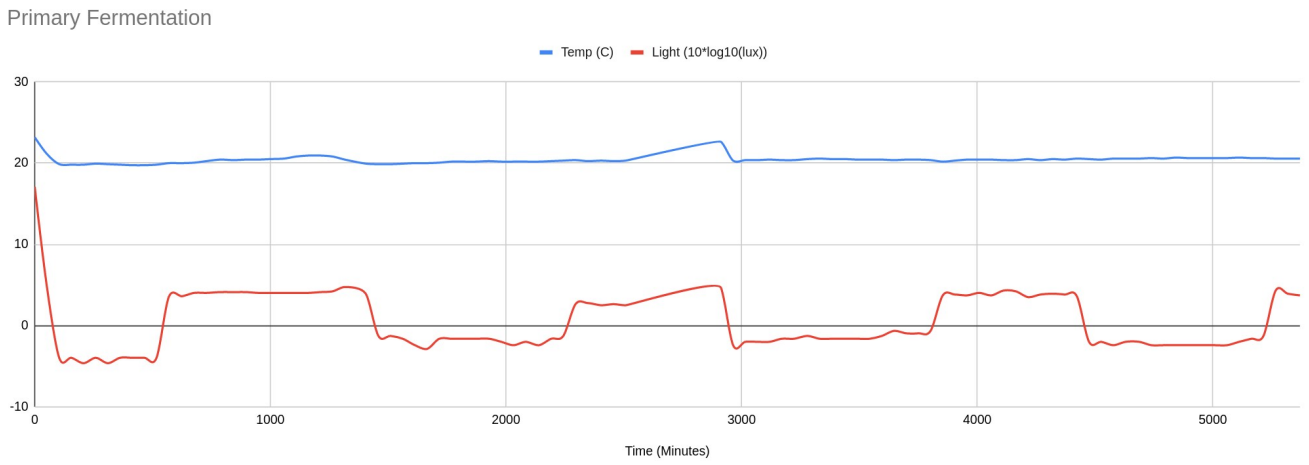
For a full end-to-end demonstration, please see the video. In this section, I will focus on the final results, and a discussion of their significance.

4A. Final Results

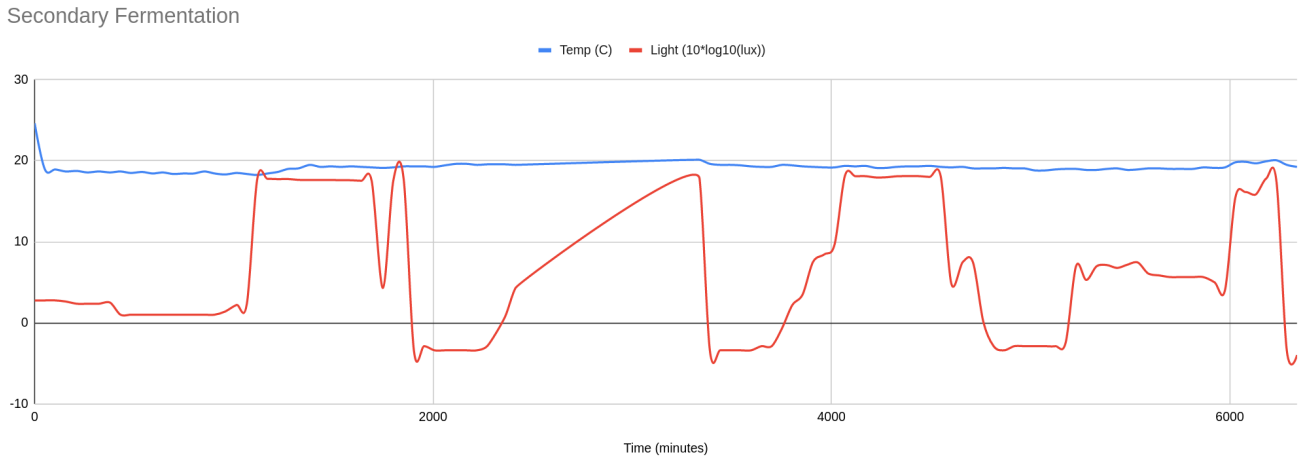
4A1. Coldcrash



4A2. Primary Fermentation



4A3. Secondary Fermentation



4B. Discussion

4B1. Coldcrash

The coldcrash result is exciting for four major reasons. Here is a quick screenshot from my demo video of the immersion wort chiller we use to cool down our brew at this stage, which allows cold water to flow through a coil submerged in the hot wort, pulling the heat out:

With that in mind, along with the explanation in my Motivation section of why coldcrash is important, here is what makes my result useful:

1. It allowed for stopping the coldcrash as soon as the target was reached. In other words, the requirement R-1 that I set out in the Project Proposal was met. I decided that having the device post a comment to Asana when the target was reached is not necessary, since during the brewing process, we the brewers are watching this chart constantly anyway. So, huge success on this aspect.

2. It allows for keeping records of coldcrash durations across brews. For example, the recorded coldcrash that lasted ~65 minutes was done in ~0°C outside temperature. In the summer for example, when the outside temperature is much higher, that duration might get much longer, which may explain imperfections in the final beer product. Having this data to explain this will be invaluable.

3. It showed my brewing team that our coldcrash takes longer than we estimated. So, it provided immediate value by presenting the actual temperature vs. time curve, indicating that there is more room for improvement than we thought in terms of accelerating the wort chilling.

4. All of my components stayed within safe temperature range during tracking. Note that the battery, the Pico, and some of the sensors are only rated to operate up to 80°C. Since the wort is 100°C when the chilling technically begins, I worried that it may not cool to below 80°C by the time the air temperature inside my device's tube intercepts the wort temperature. However, it turned out that the intercept point was not even close to 80°C, and was in fact approximately 40°C. This is also great because it likely leaves enough margin that this should remain true in higher outdoor temperatures.

4B2. Primary Fermentation

The primary fermentation results were useful, but perhaps not groundbreaking. First I will provide some context on what was collected. The primary fermentation is conducted in the exact buckets pictured below. My IoT device is placed inside a bucket as shown in the video. You will have noticed that I scaled the lux (light intensity) value in the chart by $10*\log_{10}(\text{lux})$ which was an arbitrary scale I came up with in order to view it properly on the same axes as temperature.

Now, here are my takeaways from this phase:

1. Requirement R-2 was met in field testing! This is a positive for the IoT project. I was able to satisfy the requirement R-2 that I set out in the Project Proposal, logging timestamped temperature and light data, and having warning triggers enabled.

2. The light and temperature warning thresholds were never met. This is a positive for brewing process quality. I expected that this would be the case, since the buckets are opaque and the fermentation is conducted indoors. I was pleased to see how stable the temperature was, though: I expected the fluctuation between day and night to be greater. The data collected here provides a very useful baseline for my brewing team, and allows us to immediately recognize that if either temperature or light thresholds ever trigger during primary fermentation, then we have a serious anomaly in our fermentation environment that should be addressed immediately.

4B3. Secondary Fermentation

The secondary fermentation results slightly more interesting than the primary, while very similar by design. The only difference between primary and secondary fermentation is that the brew is transferred from the plastic bucket to a glass “carboy” like the one shown below in order to get it off of the cake of spent yeast at the bottom of the bucket. Therefore from the perspective of my IoT device is exactly the same: the device is placed in the bucket, and it tracks temperature and light with timestamps, and warns if necessary.

Here are my takeaways from the data collected in secondary fermentation:

1. R-2 was met again! It is important to note that both primary and secondary were required to satisfy R-2, and they both did.

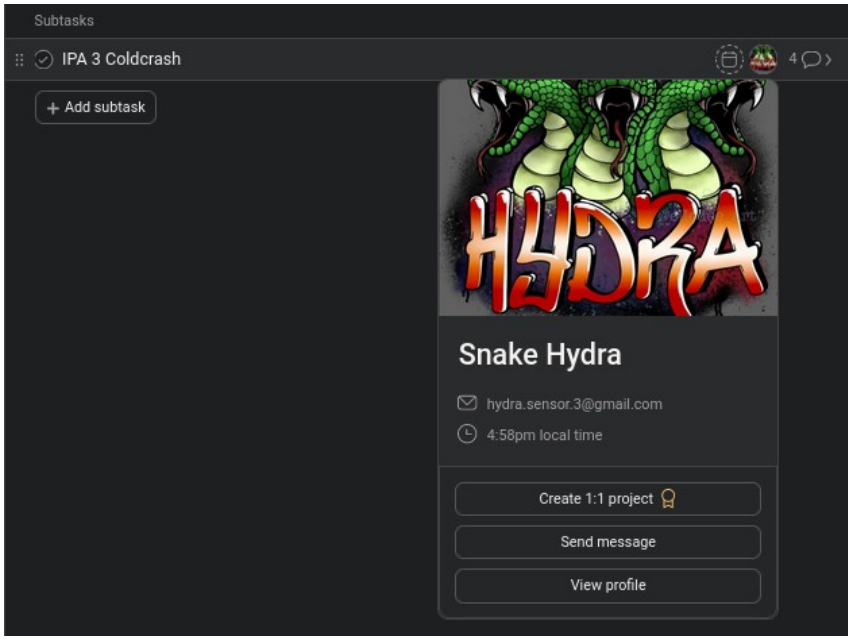
2. Light warning was triggered. Since this vessel is made of transparent glass instead of opaque white plastic, the light intensity was a lot stronger. In fact, the warning threshold was exceeded every time the sun came up (or the indoor lights got turned on!). This was useful to the brewing team as it provided a daily reminder in the form of an Asana notification about the warning, in addition to the set-in-stone Google Sheet chart. This provided motivation to the brewing team to discuss alternate storage locations for the carboys during secondary fermentation, with less daily light fluctuation.

3. No temperature warning. Just like in primary, the temperature was *very* stable. This is again a good sign and means that Asana notifications for temperature warnings should be seen as serious anomalies.

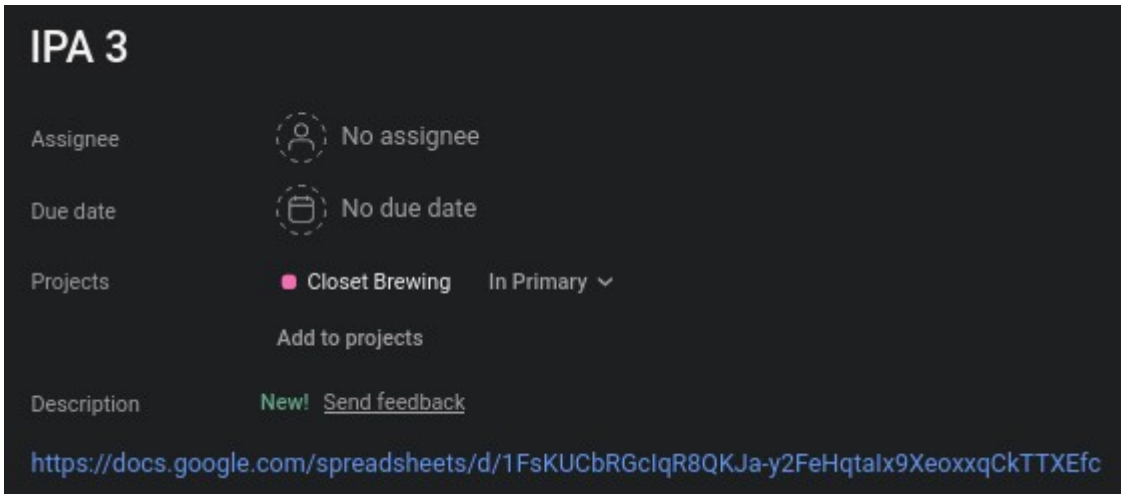
4B4. Overall Usability

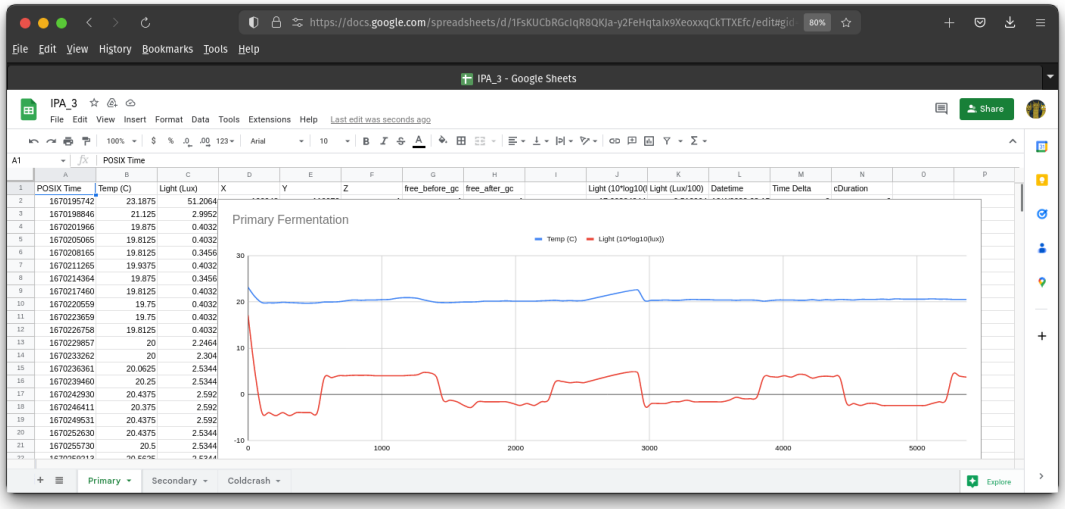
I am very pleased to say that I met the goals I set for overall user-friendliness and simplicity, which will encourage my brewing team and I to actually continue using this device going forward. Here are the key reasons for that:

1. Using the device is dead-simple. For a given beer batch, we will already have an Asana task with an appropriate title in the appropriate section (e.g. Planned, In Primary, or In Secondary). The only thing we have to do to use my IoT device is assign any subtask of the existing task to the "hydra" that matches the physical device we want to use in that batch, and then power the device on. It's really that easy.



2. Accessing data afterwards is dead-simple. The aforementioned Asana task representing a beer batch gets automatically populated with a Google Sheet link in its description, which has tabs for all three phases, and no noise. It’s truly only one click from the beer batch to all of its data, and zero manual intervention to create or maintain any links or metadata.





5. Overall Project Discussion

In this section I will briefly tie up any loose ends with regards to the following questions:

Skill-building: Did the project cause the group members to extend their skill sets?

I learned many new things over the course of this project, and mentioned them throughout this report. Here is a summary of the main ones:

- Learning Micropython (2A Hardware Design)
- Cutting PCBs to size (3A Hardware Implementation)
- Solder bridging (3A Hardware Implementation)
- The pitfalls of garbage collection (3B Software Implementation)
- A drawback of HTTP vs MQTT (3B Software Implementation)
- Microservices in Google Cloud Run (3B Software Implementation)

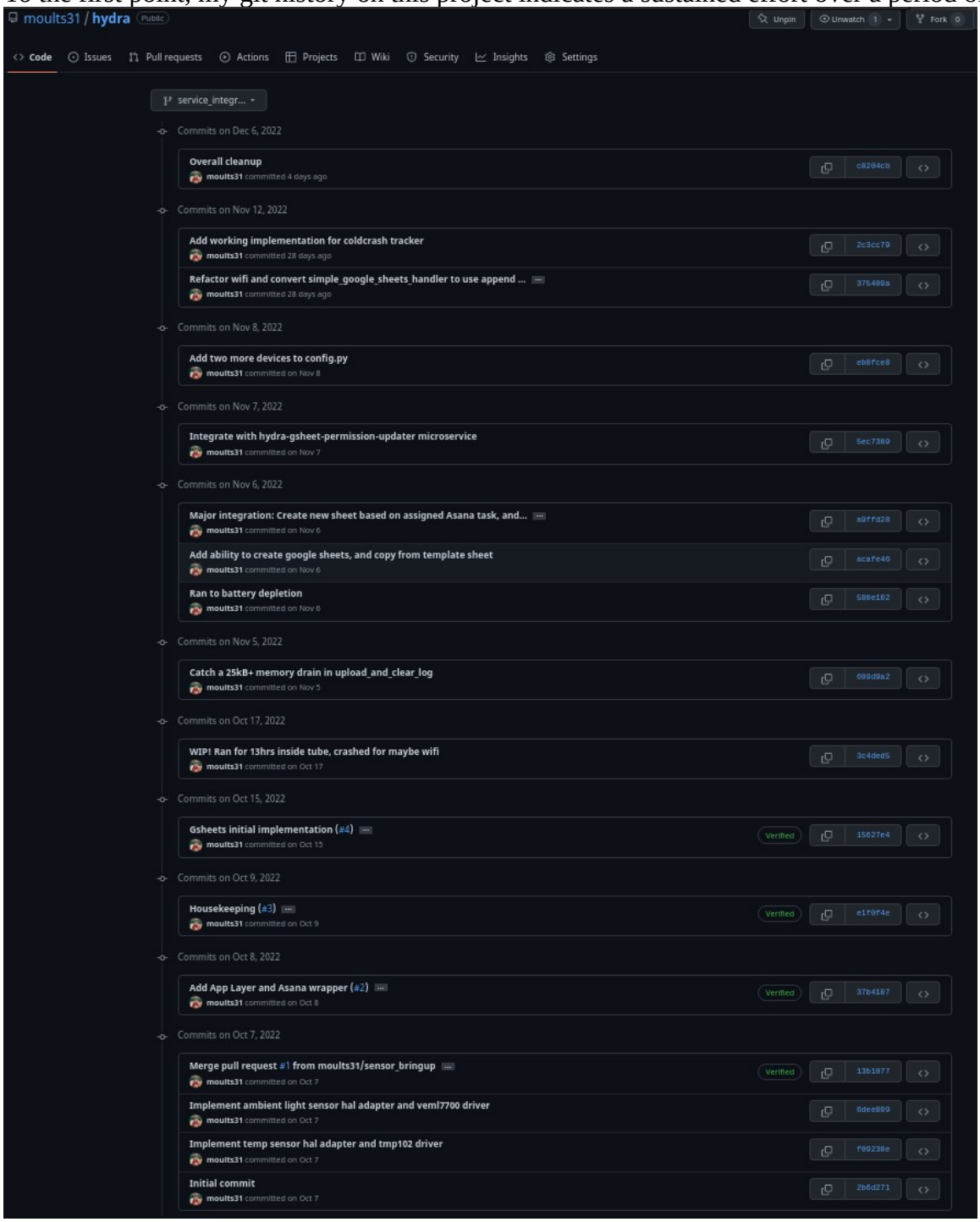
Innovation: Was the project "new" in some way?

To explicitly answer this, I will reiterate the table that appeared in Section 1 Motivation, and mention that all of the checked boxes were clearly demonstrated to have been justified and implemented.

	Temp. Sensing	Gravity Sensing	Cold Crash Tracking	Light Sensing	Asana Integration
Proposed Device	√	*R-3 (optional)*	√	√	√
FLOAT	√	√			
Tilt	√	√			
PLAATO	√	√			
iSpindel	√	√			

Scope: Was it clear the project was done over an extended period of time, and in a thoughtful manner? Was the project especially involved and difficult for the group members?

To the first point, my git history on this project indicates a sustained effort over a period of months:



And to the second point, I believe that the challenges and solutions described in Section 3 Implementation Details show a very involved, difficult, interesting, and rewarding project.