# Development of an Open Source Data Visualization and Automated Testing Platform for CubeSat Subsystems

AE 8900 MS Special Problems Report
Space Systems Design Laboratory (SSDL)
Guggenheim School of Aerospace Engineering
Georgia Institute of Technology
Atlanta, GA

Author:
Alexander Bustos

Advisor:
Prof. Brian C. Gunter

August 4, 2023

# Development of an Open Source Data Visualization and Automated Testing Platform for CubeSat Subsystems

Alexander Bustos* and Brian Gunter†
*Georgia Institute of Technology, Atlanta, GA, 30332*

**This paper discusses the initial development and implementation of a web-based software tool for data visualization and automated testing designed to be used with CubeSat subsystems. The software tool is implemented as a pair of web servers deployed to a Raspberry Pi. One of these web servers serves a graphical user interface to users via a web browser, while the other handles data acquisition, project management, and test running tasks. Software architecture and design decisions are discussed extensively. Web-based architectures are contrasted against more traditional modes of software development. The development and maintenance of a consistent, reproducible runtime environment using Nix is also detailed. Current capabilities are demonstrated and documented using a representative device under test consisting of a microcontroller and simple sensor suite. Finally, future development efforts and next steps are outlined and discussed.**

## I. Nomenclature

| | | |
|---|---|---|
| API | = | Application Programming Interface |
| CI | = | Continuous Integration |
| CRUD | = | Create, Read, Update, Delete |
| DAQ | = | Data Acquisition |
| DI | = | Dependency Injection |
| GIL | = | Global Interpreter Lock |
| GNU | = | GNU's Not Unix |
| GNOME | = | GNU Object Model Environment |
| HAT | = | Hardware Attached on Top |
| HMR | = | Hot Module Reloading |
| HTTP | = | Hyper Text Transfer Protocol |
| $I^2C$ | = | Inter-Integrated Circuit |
| IP | = | Internet Protocol |
| JS | = | JavaScript |
| LAN | = | Local Area Network |
| LSP | = | Language Server Protocol |
| MCC | = | Measurement Computing Corporation |
| OBC | = | On-Board Computer |
| ReST | = | Representational State Transfer |
| SPI | = | Serial Peripheral Interface |
| TS | = | TypeScript |
| YAML | = | YAML Ain't Markup Language |
| PIP | = | PIP Installs Packages |
| TMUX | = | Terminal Multiplexer |

*Graduate Student, Daniel Guggenheim School of Aerospace Engineering
†Associate Professor, Daniel Guggenheim School of Aerospace Engineering

## II. Introduction and Overview

### A. Introduction

When developing subsystems for CubeSats, it can often be difficult and tedious to certify that the subsystem meets a particular set of requirements. This is particularly true of requirements that may need to be evaluated over the course of several hours or several days, requiring researchers to devote time and effort to collecting data and performing a test procedure. For small operations such as university research labs, it can be difficult to allocate time and researchers to either manually performing test operations, or developing a one-off solution for automated testing.

Another challenging aspect of CubeSat subsystem development is data validation. It is common for subsystems to collect a variety of data relevant to their operation and even perform some degree of post-processing on this data. Depending on the complexity of the data, it can be time-consuming to confirm the accuracy of these data products. Additionally, during the development life cycle, it can be beneficial to log and visualize subsystem data products, both to verify their accuracy and reliability over time, and to identify any inconsistencies in their data collection processes.

In the context of smaller CubeSat operations such as university research groups, both of these issues may be exacerbated by resource limitations. In these settings, it can be common to have only one or two hardware copies of other CubeSat components such as the on-board computer (OBC), or similarly, only a single FlatSat for integration testing. Naturally, this can make concurrent development of subsystems challenging, and researchers may have to implement stopgap measures to work around these limitations. An example of this might be a subsystem that is designed to interface with the OBC via an $I^2C$ or SPI interface, but due to resource limitations, researchers implement an additional application programming interface (API) via serial for development purposes, ultimately resulting in a time-consuming and tedious development experience.

This project seeks to address the issues described above with a software application deployed to a Raspberry Pi 4B. This software platform takes advantage of the Raspberry Pi's hardware interfaces, allowing researchers to develop, test, and monitor CubeSat hardware systems without having to perform the additional overhead tasks discussed earlier. Additionally, this project is built using entirely open source software, and is itself open source, which makes it relatively easy to replicate the system described in this paper, and also lowers the barrier to entry for this platform to just the cost of the necessary hardware used to host the software. The following sections describe the initial design and implementation of this software platform, an example use case, and next steps for future development.

### B. Overview

The software application introduced in the previous section is designed as a pair of web applications – a user-facing client application written primarily in a strongly-typed super set of JavaScript (JS) called TypeScript (TS)[1], and a headless back-end application written in Python that the client application interacts with via a representational state transfer (ReST) API[2]. The following sections will discuss various aspects of the development of this software, including its architecture, implementation, and current functionality.

## III. System Architecture

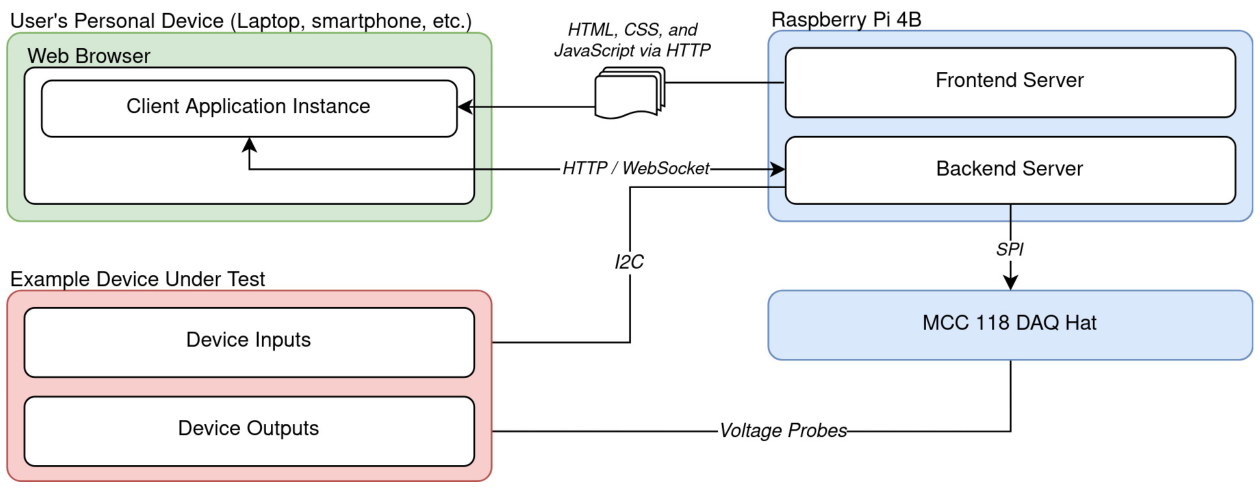

User's Personal Device (Laptop, smartphone, etc.)
Web Browser
Client Application Instance
HTML, CSS, and JavaScript via HTTP
Raspberry Pi 4B
Frontend Server
Backend Server
HTTP / WebSocket
I2C
SPI
Example Device Under Test
Device Inputs
Device Outputs
MCC 118 DAQ Hat
Voltage Probes

**Fig. 1 High level overview of system architecture.**

The software application described in the previous section is designed as a pair of web applications. This section will provide further explanation of the system architecture and explore the trade-offs of the web-based architecture developed for this project. Fig. 1 shows the general architecture of the entire system. This figure primarily aims to illustrate the interactions between the physically discrete systems that together form the basis of the software platform. These systems are color-coded in Fig. 1, and are enumerated as follows:

1) Raspberry Pi 4B – The Raspberry Pi hosts the two web servers introduced previously. Together, these applications are responsible for serving a web-facing user interface to users on the same local area network (LAN) as the Pi, and interacting with the device under test via the Pi's built-in hardware interfaces. Additionally, these web applications are designed to be able to take advantage of third-party hardware systems, such as the MCC 118 DAQ Hat shown in the diagram. The DAQ Hat provides high accuracy analog-to-digital voltage sampling functionality, which allows the software platform to externally validate voltage outputs of the device under test.
2) User's Personal Device (Laptop, smartphone, etc.)– Users primarily interact with the Raspberry Pi (and thus, the device under test) via a web-based user interface that is served to users in the form of HTML, CSS, and JavaScript that is delivered to a user's device via the front-end server hosted on the Raspberry Pi. Each instance of the front-end application that is served to users contains code to communicate with the back-end server hosted on the Raspberry Pi via HTTP and WebSocket protocols, which ultimately allows users to connect with and instrument the device under test.
3) Device Under Test– In Fig. 1, the device under test represents the physical hardware subsystem being instrumented by the Raspberry Pi and the software hosted on it. The device can be controlled using the Pi's hardware interfaces, and further tested using external tools such as the MCC 118 DAQ Hat shown in the figure.

To further clarify the layout shown in Fig. 1, the actual hardware is shown in Fig. 2. Note that, for demonstration purposes, a simple device under test demonstrator was developed for this project, which consists of a microcontroller and sensor suite. This setup is explored further in Section V.

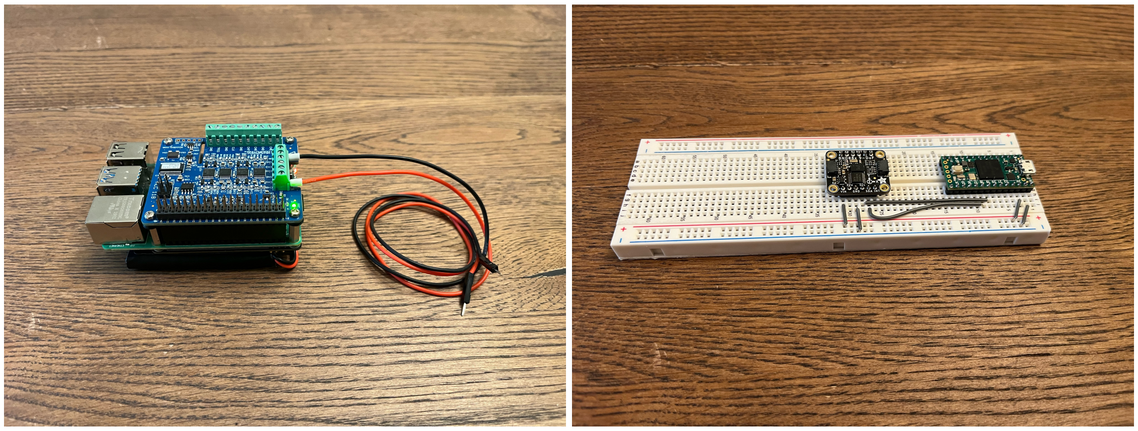


**Fig. 2 Assembled Raspberry Pi with DAQ hat and voltage probes (left) and example device under test (right).**

The Raspberry Pi and device under test shown in Fig. 2 can be connected via jumper wires and probes to form the basis of the complete system described in Fig. 1. The complete, connected system is shown in Fig. 3.

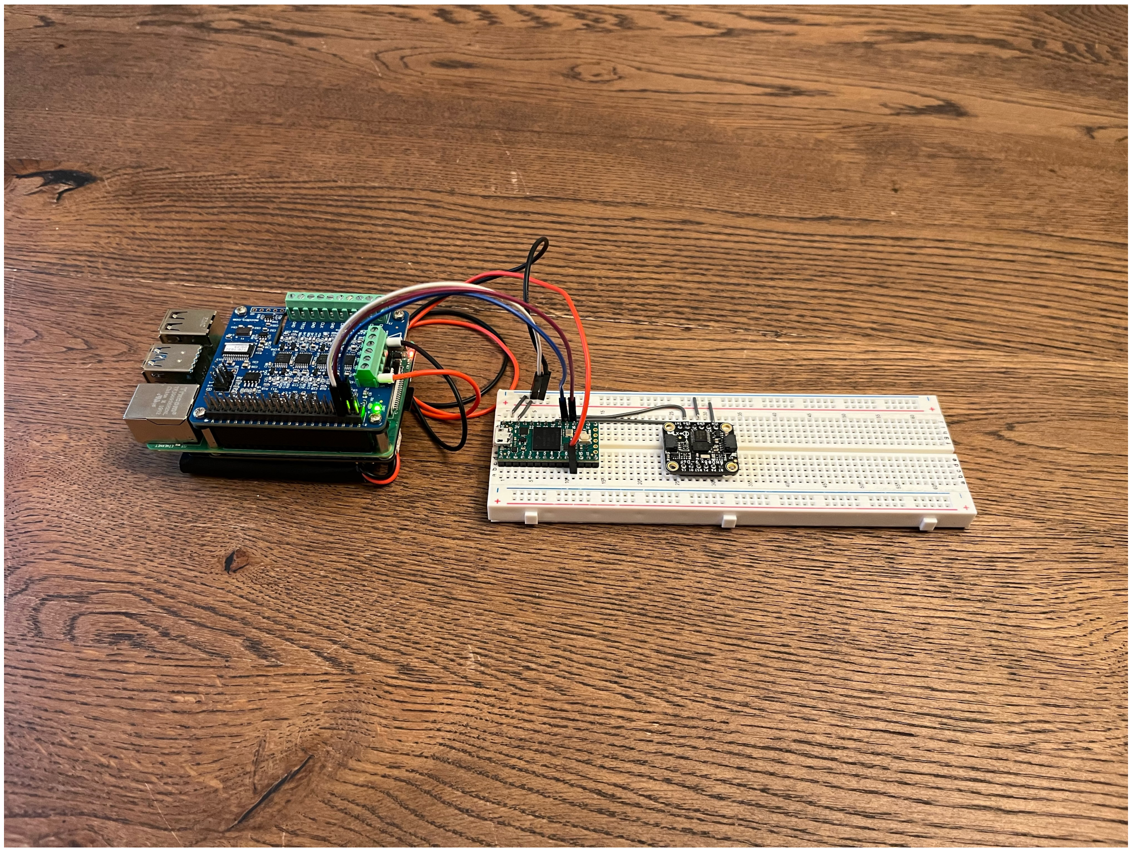


**Fig. 3 Raspberry Pi connected to example device under test via I$^2$C and voltage probes.**

## A. Hardware

The software for this project is hosted on a Raspberry Pi 4B, which was selected for a number of reasons. Historically, Raspberry Pi's have offered excellent software support for a wide variety of commonly used software packages including Python and Node.js. The Raspberry Pi Foundation also offers an officially supported fork of the Debian GNU/Linux distribution called Raspberry Pi OS, which is available in both headless (“Lite”) and desktop variants[3]. Raspberry Pi OS Lite was selected for this application to avoid the unnecessary overhead of running a display server such as Xorg.

To demonstrate the hardware extensibility of this project, a Measurement Computing Corporation (MCC) DAQ HAT 118 was selected as an additional hardware accessory to attach to the Pi. As the name Hardware Attached on Top (HAT) implies, the board simply connects to the top of the Pi via the Pi’s GPIO header pins. As discussed briefly earlier, this particular HAT is capable of taking accurate analog voltage measurements in the range of ±10V[4]. It should be noted that to achieve the base functionality of instrumenting a subsystem over its native hardware protocol, no additional hardware beyond the Raspberry Pi itself is needed, provided that the Pi supports the hardware protocol in question.

Another motivator of the hardware configuration selected for this project is cost. Section II discussed the goal and importance of developing a non-restrictive, open source platform with a low cost of entry. Table 1 shows the cost breakdown of the system, with the entire configuration costing under $200 2023 US Dollars. This one-time hardware investment is in contrast to a closed-source approach, where software packages for tools like LabView or MATLAB/Simulink often cost several hundred to several thousand dollars for a single license. These licenses are also often subscription-based, requiring users to pay high monthly or annual fees for continued use of the software.

**Table 1 System hardware cost.**

| **Hardware Component** | **Cost (2023 US Dollars)** |
|---|---|
| Raspberry Pi 4B, 4GB | 55.00 |
| MCC DAQ HAT 118 | 99.00 |
| SanDisk 128GB MicroSD Card | 13.25 |
| **Total** | 167.25 |

## B. Software

### *1. Overview and Benefits*

The software platform developed for this project uses a web-based architecture, where clients communicate with a server using HTTP and WebSocket communication protocols. At application runtime, there are two web server applications hosted on the Raspberry Pi. When a user connects to the appropriate socket using a web browser, they are sent the HTML, CSS, and JavaScript that make up the client application from the front-end server. Each instance of the client application contains bindings to the ReST API exposed by the back-end server, effectively granting users on the same LAN as the Raspberry Pi the ability to interact with it and its connected hardware using the various methods implemented by the back-end server.

This architecture has a number of benefits, starting with the developer experience. With the rise in popularity of web development in recent years, popular web programming languages including Python, TypeScript, Rust, and others enjoy first-class Integrated Development Environment (IDE) support in the form of well-maintained tree-sitters providing reliable syntax highlighting, Language Server Protocol (LSP) servers providing auto-complete functionality, and auto-formatters and linters enforcing a consistent code style[5]. Additionally, with popular web frameworks for a variety of applications made widely available for higher-level programming languages like Python and TypeScript, the skills barrier to entry for developers is relatively low, making it easy for new programmers to familiarize themselves with the software developed for this project.

In the context of this project, where software is deployed to a resource-limited platform like the Raspberry Pi, this web architecture offers another advantage – the offloading of compute-intensive UI rendering tasks to a user’s personal device. In a traditional application natively developed for GNU/Linux, a user would have to connect a display (either physical or virtual) to the Raspberry Pi, start a display server and desktop environment such as GNOME, and then make use of the application. Already, this approach imposes the overhead of running a desktop environment within which to run graphical applications. However, once the application has started, the Pi is then also responsible for all of the graphical rendering compute tasks required by the application. In the case of this project, this includes frequently updating graphs and other data visualization tools, which may be taxing to perform natively on the Raspberry Pi, while also running necessary tasks on the back-end server. With a web-based architecture however, there is no need to run a desktop environment or any graphical applications directly on the Pi. Instead, when a user connects to the front-end server via a web browser, all of the graphical rendering tasks are performed on the user's device, which in turn significantly frees up valuable RAM and CPU time on the Raspberry Pi.

*2. Drawbacks*

While the web architecture used in this project comes with many benefits, it is important to note some of the key drawbacks of this approach. This will help highlight some of the software's limitations and weak points, theryby giving a better overall picture of the platform's capabilities.

The previous section highlighted the first-class developer experience that can be achieved in a modern web development environment. However, it should also be noted that many of the tools enabling this experience – language servers in particular – are often themselves resource intensive[6]. This makes it relatively difficult to develop directly on the Raspberry Pi, as its limited resources are hampered by these tools. For this project, development was instead performed on a separate, more powerful machine, and then deployed to the Raspberry Pi. Additionally, while languages like Python and TypeScript can have a gentler learning curve for new programmers, they are generally far slower than compiled languages like Rust, Go, and C. Effectively, this performance hit will be realized on the back-end server, which is implemented with a single-threaded, asynchronous architecture. While this approach is suitable for the current project requirements, a multi-threaded approach in a compiled language would certainly deliver higher performance on the back-end, albeit at the expense of complexity. Finally, web architecture requires some sort of external networking to fully function. In the case of this project, users must be able to connect their personal computing device and the Raspberry Pi to the same Local Area Network. This is in contrast to a native application, which could feasibly run directly on the Raspberry Pi, without external networking dependencies.

*3. Client Architecture*

The client application can be viewed in a modern web browser. It is responsible for providing a graphical user interface (GUI) to the user, that can be used to interact with the back-end server running on the Raspberry Pi. Fig. 4 diagrams a high level overview of the most novel part of the front-end application – the dashboard. In the interest of not producing an overly complicated diagram, Fig. 4 makes some simplifications about the runtime operation of the client application, but still aims to provide useful information on its general operating strategy. A sample dashboard can also be seen in Fig. 5. In the context of this project, a dashboard is essentially a user-customizable collection of widgets that provide useful data and methods of interaction with the Raspberry Pi and the system under test. The sample dashboard provided shows several graphs of system resource utilization that update in real-time, as well as a widget that can be used to control the selective recording of data from all of the data sources available to the back-end server. The dashboard page is discussed further in Section V.

The software application as a whole is capable of supporting multiple unique projects, with each project having its own dashboard configuration and project assets stored in a directory on the Raspberry Pi. In order to persist project-specific dashboard layouts, everything shown on the dashboard is serialized into JSON and written to a YAML file in the appropriate project directory whenever a project is loaded on the client. Fig. 4 labels the information contained in one of these YAML files as "Project Metadata" and "Project Configuration". At runtime, a project configuration and any associated metadata (the project location on the Raspberry Pi file system, or the date the project was last modified, for example) is loaded into the client memory via an HTTP response from the back-end. The user is then free to make changes to the dashboard by adding, removing, reordering, and generally reconfiguring the dashboard to their liking. When the dashboard is saved by clicking the save icon in the upper right of the user interface shown in Fig.5, the project configuration stored in the client memory is written to file on the Raspberry Pi, thereby allowing it to persist across different client sessions.

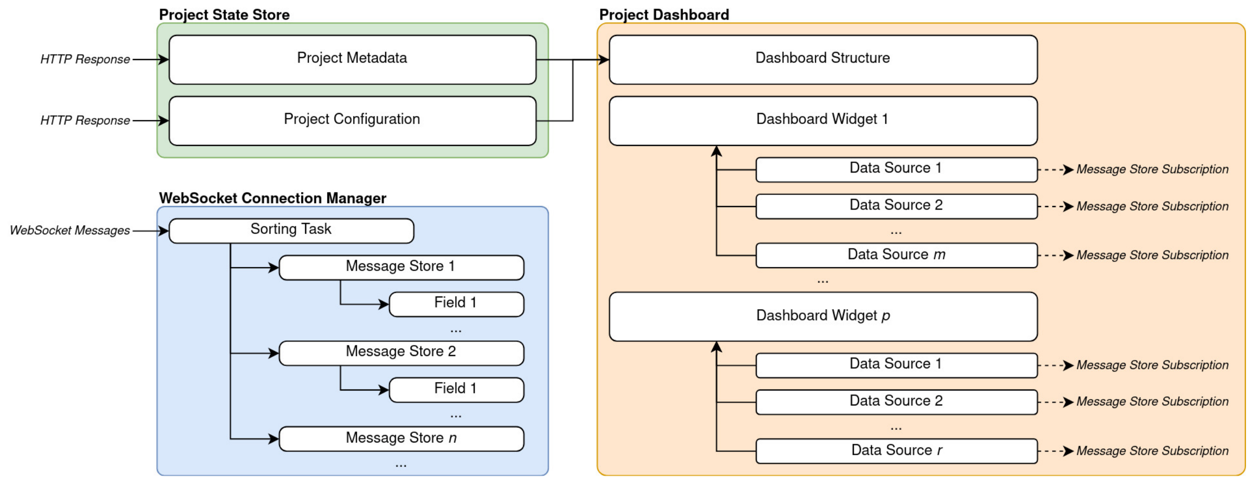


**Fig. 4 Client application runtime architecture.**

As Fig. 4 shows, the client application implements an entity called the "WebSocket Connection Manager". Since the application has a heavy focus on real-time data visualization and analysis, the client implements a WebSocket connection to the backend server, allowing it to receive data without having to first send an HTTP request for it. The WebSocket Connection Manager is responsible for initiating the connection to the backend server, and also sorting incoming WebSocket messages into their own data stores. The utilization of Svelte's stores for this application is discussed at length in Section IV, but for now they can be thought of as topics that can be subscribed to and unsubscribed from by different components on the dashboard. With this scheme, the client application can avoid creating duplicates of the same data sets, and instead allow multiple dashboard widgets to subscribe to the same data sets and automatically receive updates whenever a new message corresponding to that data set is received. Additionally, as Fig. 4 shows, each widget on the dashboard can have its own user-configurable collection of data sources. These sources are essentially just references to the message stores managed by the WebSocket Connection Manager and contain a unique JSON-serializable key, so that the configuration of data sources for each widget can be stored and persisted with the rest of the dashboard layout.

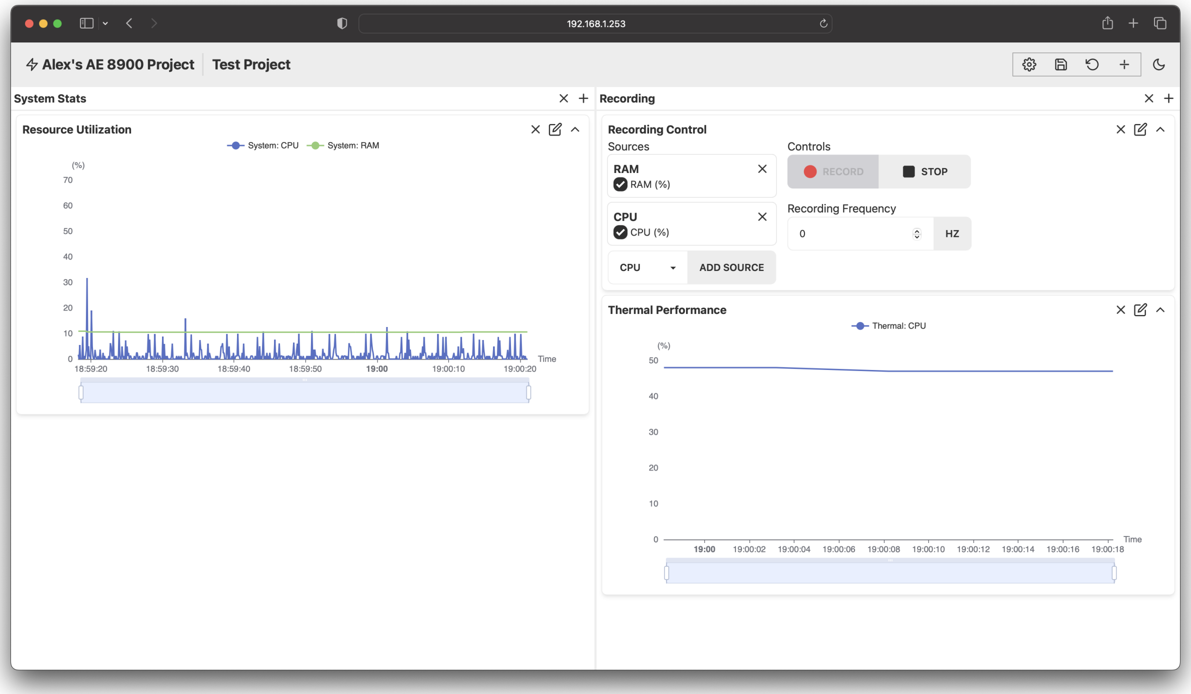


Fig. 5 Sample dashboard user interface.

## 4. Server Architecture

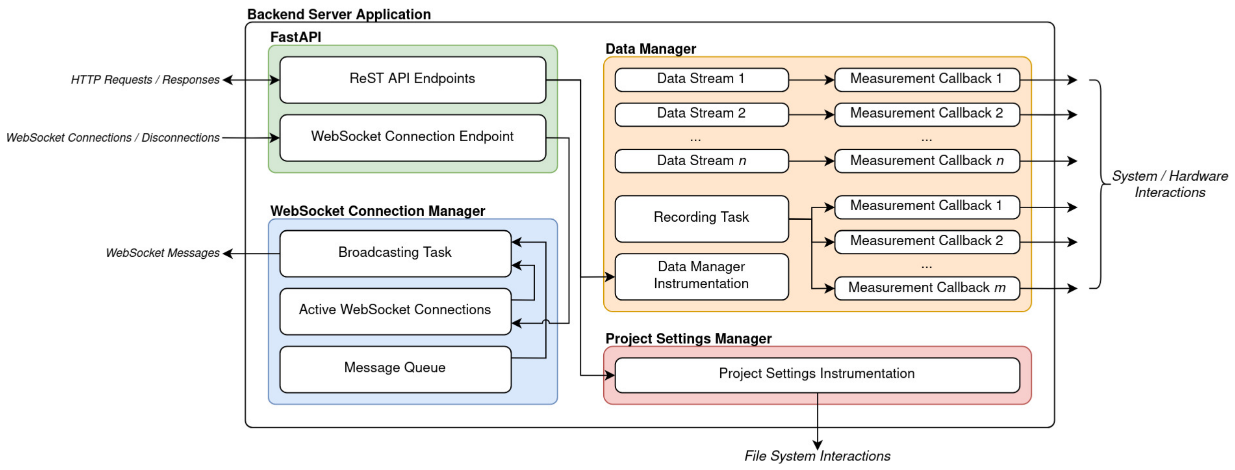


Fig. 6 Back-end server application runtime architecture.

Fig. 6 shows the general architecture of the back-end server at runtime. As with Fig. 4, this diagram opts for simplicity in order to provide a clearer explanation of the core functionality of the back-end system, so some of the finer intricacies of the back-end server operation may be omitted here. The back-end server is structured around a web server framework for Python called FastAPI, which is discussed in detail in Section IV. When an HTTP request is made to a particular endpoint on the back-end web server, FastAPI processes the request appropriately, ensuring that any functions or tasks associated with the endpoint are executed, and then returns the appropriate HTTP response. HTTP requests are the primary way of instrumenting the back-end server from the front-end user interface and can be used to perform common create, read, update, delete (CRUD) tasks, as well as more application specific tasks such as starting and stopping particular measurements streams from the device under test.

Similar to the client application, the back-end server implements a *WebSocket Connection Manager*. However, while this WebSocket Connection Manager *does* manage WebSocket connections, the tasks it performs are somewhat inverted compared to the tasks performed by the manager of the same name on the client-side. In the client application, the WebSocket Connection Manager is responsible for initiating a single connection to the back-end server, and distributing messages from that single connection to many distinct collections of data, so that components on the client-side can easily subscribe to updates from a specific dataset. In contrast, the WebSocket Connection Manager on the back-end server is responsible for managing multiple WebSocket connections, since multiple clients can connect to the back-end at the same time. Then, messages from various data sources are aggregated into a single queue, and each message is broadcast to every active WebSocket connection, so that every connected client receives the same data. In this way, the client WebSocket Connection Manager may be thought of as a fan-out pattern, where data from a single source is split into multiple sources, whereas the back-end server WebSocket Connection Manager may be thought of as a fan-in pattern, where data from multiple sources are multiplexed into a single stream of messages.

All of the messages sent by the Connection Manager are generated by "Data Streams" managed by the "Data Manager", as labeled in Fig. 6. Currently, when the back-end server is started, all of the available data sources (which, for the time being, must be configured in code before runtime) are loaded into the Data Manager, which then has the ability to enable, disable, change the sampling rate of, and record from each Data Stream. An interesting architectural point to expand upon here here is that the back-end server is entirely single-threaded, or rather, a single process, since Python is inherently single-threaded due to the Global Interpreter Lock (GIL)[7]. To that point, the back-end does not use Python's multiprocessing functionality – instead, it relies heavily on Python's support for asynchronous programming[8, 9]. Data Streams, specifically, are coroutines that run until signaled to stop, taking measurements at specific intervals and placing each message onto the Connection Manager queue. This approach has the advantage of simplicity and sidesteps the difficulties associated with multiprocessing in Python. Additionally, since measurements are taken at discrete intervals, where the interval between measurements is often far longer than the time required to take a single measurement, they can be thought of as highly I/O bound in nature, making asynchronous programming a good solution to the problem of gathering measurements from multiple sources concurrently. This approach also allows for measurements to be sampled at different rates, with little added complexity to the application. The downside to this approach, when compared with a multiprocessing scheme, is, of course, performance. A single Python process will only be able to take advantage of a single processor core of the Raspberry Pi 4's quad-core processor, whereas multiple Python processes could hypothetically scale to all four cores. Ultimately this design decision is a trade off between simplicity and scalability, and the degree of scalability that can be achieved with the single-process asynchronous approach is a subject of future work.

The final piece of the server architecture shown in Fig. 6 is the Project Settings Manager. This is effectively just a data structure that can be accessed by different parts of the back-end server to provide context for the current runtime state of the application. As mentioned earlier, this project is designed to support an arbitrary number of projects, each of which have their own directory in the Raspberry Pi file system. An example usage of the Project Settings Manager would then be maintaining knowledge of the current active project on the front-end, so that different entities on the back-end can read and write files from the appropriate project directory on the Raspberry Pi. In addition to this shared data structure, the back-end server utilizes some miscellaneous utility functions to read, write, and update project layout files. These functions also perform type validation on project-specific files, making it possible to identify if a file has been improperly modified or corrupted.

## IV. Implementation

This section aims to expand on the architecture discussed in Section III, and describe the technical details behind the software's implementation. It has been established that this project is predominantly written with TypeScript for the client application and Python for the back-end server application. Specifically, the client application is developed using a component framework called Svelte and its corresponding meta-framework SvelteKit[10]. The back-end server application is developed using a web framework called FastAPI[11]. However, before discussing the more granular details of application development, it is important to first establish a reproducible development environment that contains all of the necessary dependencies to run the application.

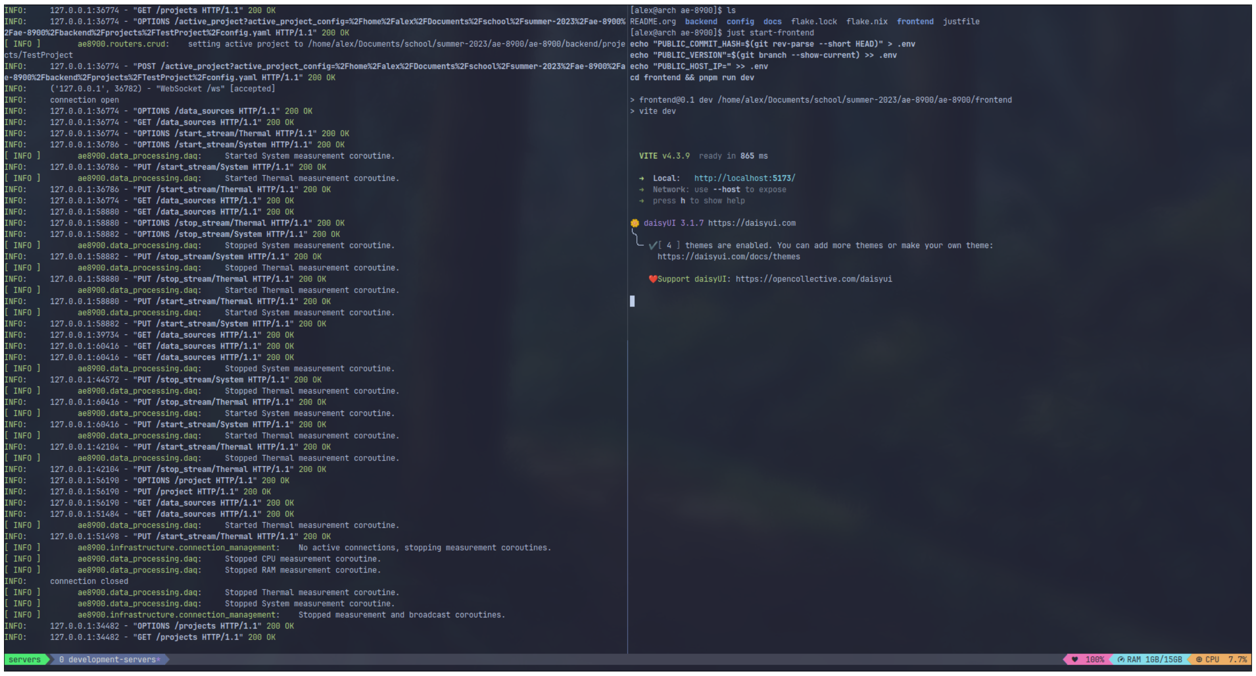
**A. Development Environment**

This project involved a common challenge in software development – maintaining the same environment across different machines, or, in other words, the "it works on my machine" problem. This issue simply refers to the difficulty of managing a complex graph of dependencies between different computers that may vary drastically in their configurations. In the case of this project, the problem manifests itself as managing dependencies between a development environment running on a 64-bit x86 GNU/Linux machine and the targeted production environment of a 64-bit ARM GNU/Linux machine (the Raspberry Pi).

For projects developed predominantly in a single language, there is often tooling available to generate reproducible environments. For example, `conda-lock` can be used to generate YAML-based environment specifications for Python projects using the `conda` package manager[12]. However, since this project uses multiple languages, and also relies on a number of binary dependencies like `tmux` and the command-runner `just`, a different approach was taken[13, 14]. Specifically, the entire runtime and development environment for this project is specified within a single Nix flake. Nix is a package manager, functional programming language, and build system (among other things) that is renowned for producing highly reproducible, isolated software environments[15]. The NixPkgs package repository is also the largest software repository in the world, containing more than 80,000 packages. This is in contrast to the Raspbian package repository's 35,000. Additionally, the NixPkgs repository is the most up-to-date repository in the world, with approximately 75% of its packages up to date, again, in contrast to Raspbian's 38%[16]. Ultimately, for this project, using Nix and NixPkgs simply makes it far easier to install more packages that are more up to date than using the default Raspbian repositories. A prime example of this is Python – as of the time of this paper, the most recent version of Python in the Raspbian Stable package repository is Python 3.9.2, which is already troublesome as the back-end server requires at least Python 3.11. Here, Nix also has the advantage of being a single solution for package management, from binary dependencies to Python packages, which is a far more attractive approach than cobbling together a set of disparate tools like `pyenv` and `nvm` to manage different versions of a single software package like Python or Node.

Unfortunately, Nix is not without its downsides, which are particularly evident when a package is not already on the NixPkgs repository. This situation did arise in the course of this project – The MCC DAQ HAT used to take analog voltage measurements requires Python and C libraries that are not on NixPkgs. Initially, this did create difficulties in development, as software built with Nix tends to interact most cleanly with other software built with Nix. That is to say, there wasn't simply a straightforward solution to use Nix for everything except these specific libraries, and just install these libraries manually. Fortunately, however, a solution was arrived at by building custom Nix derivations into the project flake. These derivations pull the C and Python source directly from MCC's GitHub and build both libraries from source, thus making them available in the Nix development environment. It should be noted, though, that Nix has a somewhat steep learning curve and may not be a feasible build tool for all types of software packages. For this project, however, Nix has proven to be an invaluable tool for speeding up deployment across machines and reducing time spent managing dependencies.

Outside of package management, the development environment for this project notably makes use of two tools, `tmux` and `just`, to significantly improve the development experience. `just` is a command runner that can be used to save and run project-specific commands, somewhat similar to GNU Make[14]. `tmux` is a very popular tool for terminal multiplexing and session management similar to GNU Screen[13]. Used together, these tools make it very easy to start both locally hosted servers for development purposes, and servers for use on a LAN. Additionally, a Python package called `tmuxp` allows multi-pane, multi-window `tmux` sessions to be loaded from YAML configuration files[17]. This makes it possible to deploy a split-pane `tmux` session running both the front-end and back-end web servers side-by-side, as shown in Fig. 7. `tmux` sessions like these can be easily detached from a terminal emulator, making it trivial to persist the web server processes after logging out of the Raspberry Pi.



```
INFO:     127.0.0.1:36774 - "GET /projects HTTP/1.1" 200 OK
INFO:     127.0.0.1:36774 - "OPTIONS /active_project?active_project_config=%2Fhome%2Falex%2FDocuments%2Fschool%2Fsummer-2023%2Fae-8900%
2Fae-8900%2Fbackend%2Fprojects%2FTestProject%2Fconfig.yaml HTTP/1.1" 200 OK
[ INFO ]        ae8900.routers.crud:    setting active project to /home/alex/Documents/school/summer-2023/ae-8900/ae-8900/backend/proje
cts/TestProject
INFO:     127.0.0.1:36774 - "POST /active_project?active_project_config=%2Fhome%2Falex%2FDocuments%2Fschool%2Fsummer-2023%2Fae-8900%2Fa
e-8900%2Fbackend%2Fprojects%2FTestProject%2Fconfig.yaml HTTP/1.1" 200 OK
INFO:     ('127.0.0.1', 36782) - "WebSocket /ws" [accepted]
INFO:     connection open
INFO:     127.0.0.1:36774 - "OPTIONS /data_sources HTTP/1.1" 200 OK
INFO:     127.0.0.1:36774 - "GET /data_sources HTTP/1.1" 200 OK
INFO:     127.0.0.1:36774 - "OPTIONS /start_stream/Thermal HTTP/1.1" 200 OK
INFO:     127.0.0.1:36786 - "OPTIONS /start_stream/System HTTP/1.1" 200 OK
[ INFO ]        ae8900.data_processing.daq:     Started System measurement coroutine.
INFO:     127.0.0.1:36786 - "PUT /start_stream/System HTTP/1.1" 200 OK
[ INFO ]        ae8900.data_processing.daq:     Started Thermal measurement coroutine.
INFO:     127.0.0.1:36786 - "PUT /start_stream/Thermal HTTP/1.1" 200 OK
INFO:     127.0.0.1:36774 - "GET /data_sources HTTP/1.1" 200 OK
INFO:     127.0.0.1:58880 - "GET /data_sources HTTP/1.1" 200 OK
INFO:     127.0.0.1:58880 - "OPTIONS /stop_stream/Thermal HTTP/1.1" 200 OK
INFO:     127.0.0.1:58882 - "OPTIONS /stop_stream/System HTTP/1.1" 200 OK
[ INFO ]        ae8900.data_processing.daq:     Stopped System measurement coroutine.
INFO:     127.0.0.1:58882 - "PUT /stop_stream/System HTTP/1.1" 200 OK
[ INFO ]        ae8900.data_processing.daq:     Stopped Thermal measurement coroutine.
INFO:     127.0.0.1:58880 - "PUT /stop_stream/Thermal HTTP/1.1" 200 OK
[ INFO ]        ae8900.data_processing.daq:     Started Thermal measurement coroutine.
INFO:     127.0.0.1:58880 - "PUT /start_stream/Thermal HTTP/1.1" 200 OK
[ INFO ]        ae8900.data_processing.daq:     Started System measurement coroutine.
INFO:     127.0.0.1:58882 - "PUT /start_stream/System HTTP/1.1" 200 OK
INFO:     127.0.0.1:39734 - "GET /data_sources HTTP/1.1" 200 OK
INFO:     127.0.0.1:60416 - "GET /data_sources HTTP/1.1" 200 OK
INFO:     127.0.0.1:60416 - "GET /data_sources HTTP/1.1" 200 OK
[ INFO ]        ae8900.data_processing.daq:     Stopped System measurement coroutine.
INFO:     127.0.0.1:44572 - "PUT /stop_stream/System HTTP/1.1" 200 OK
[ INFO ]        ae8900.data_processing.daq:     Stopped Thermal measurement coroutine.
INFO:     127.0.0.1:60416 - "PUT /stop_stream/Thermal HTTP/1.1" 200 OK
[ INFO ]        ae8900.data_processing.daq:     Started System measurement coroutine.
INFO:     127.0.0.1:60416 - "PUT /start_stream/System HTTP/1.1" 200 OK
[ INFO ]        ae8900.data_processing.daq:     Started Thermal measurement coroutine.
INFO:     127.0.0.1:42104 - "PUT /start_stream/Thermal HTTP/1.1" 200 OK
[ INFO ]        ae8900.data_processing.daq:     Stopped Thermal measurement coroutine.
INFO:     127.0.0.1:42104 - "PUT /stop_stream/Thermal HTTP/1.1" 200 OK
INFO:     127.0.0.1:56190 - "OPTIONS /project HTTP/1.1" 200 OK
INFO:     127.0.0.1:56190 - "PUT /project HTTP/1.1" 200 OK
INFO:     127.0.0.1:56190 - "GET /data_sources HTTP/1.1" 200 OK
INFO:     127.0.0.1:51484 - "GET /data_sources HTTP/1.1" 200 OK
[ INFO ]        ae8900.data_processing.daq:     Started Thermal measurement coroutine.
INFO:     127.0.0.1:51498 - "PUT /start_stream/Thermal HTTP/1.1" 200 OK
[ INFO ]        ae8900.infrastructure.connection_management:    No active connections, stopping measurement coroutines.
[ INFO ]        ae8900.data_processing.daq:     Stopped CPU measurement coroutine.
[ INFO ]        ae8900.data_processing.daq:     Stopped RAM measurement coroutine.
INFO:     connection closed
[ INFO ]        ae8900.data_processing.daq:     Stopped Thermal measurement coroutine.
[ INFO ]        ae8900.data_processing.daq:     Stopped System measurement coroutine.
[ INFO ]        ae8900.infrastructure.connection_management:    Stopped measurement and broadcast coroutines.
INFO:     127.0.0.1:34482 - "OPTIONS /projects HTTP/1.1" 200 OK
INFO:     127.0.0.1:34482 - "GET /projects HTTP/1.1" 200 OK
```

```
[alex@arch ae-8900]$ ls
README.org  backend  config  docs  flake.lock  flake.nix  frontend  justfile
[alex@arch ae-8900]$ just start-frontend
echo "PUBLIC_COMMIT_HASH=$(git rev-parse --short HEAD)" > .env
echo "PUBLIC_VERSION"=$(git branch --show-current) >> .env
echo "PUBLIC_HOST_IP=" >> .env
cd frontend && pnpm run dev

> frontend@0.1 dev /home/alex/Documents/school/summer-2023/ae-8900/ae-8900/frontend
> vite dev



  VITE v4.3.9  ready in 865 ms

  ➜  Local:   http://localhost:5173/
  ➜  Network: use --host to expose
  ➜  press h to show help

🌼 daisyUI 3.1.7 https://daisyui.com
╰─ ✔︎[ 4 ] themes are enabled. You can add more themes or make your own theme:
     https://daisyui.com/docs/themes

   ❤︎Support daisyUI: https://opencollective.com/daisyui

```

```
servers  0 development-servers*                                  ♥ 100%  RAM 1GB/15GB  CPU  7.7%
```

**Fig. 7 Example `tmux` server session configured with `tmuxp` and started with `just`.**

### B. Client

As mentioned earlier, the client application was developed using TypeScript and Svelte. Svelte is a component framework that compiles directly to HTML, CSS, and JavaScript and interacts with the Document Object Model (DOM) directly, making it a comparatively fast web framework. The development team behind Svelte also develops a meta-framework called SvelteKit, which provides a host of useful features for web applications like file-system routing and hot module reloading (HMR)[10]. SvelteKit is also used in this application. Fig. 8 shows the directory structure of the front-end application. Here, certain folder names are handled in certain ways with SvelteKit. For example, SvelteKit uses file-system routing with the `routes` folder to determine the URLs of different pages in the application[18]. So, for example, the index file located in the `routes/project` directory would have the URL `192.168.1.1:5173/routes/project`. Here, the IP address and port provided are just an example, meant to be representative of the base URL of an instance of the front-end server deployed to a Raspberry Pi on the LAN.

```
.
└── src
    ├── lib
    │   ├── api
    │   ├── components
    │   │   ├── core
    │   │   ├── dashboard
    │   │   │   └── settings
    │   │   └── general
    │   ├── stores
    │   ├── types
    │   ├── utils
    │   └── websocket
    └── routes
        └── project
            └── settings
```

**Fig. 8 Front-end directory structure.**

As Fig. 8 shows, the `src/lib` directory contains the bulk of the files used to implement the functionality in the client application. Here, `src/lib/api` is a type-safe implementation of the back-end server ReST API that is automatically generated using OpenAPI, which is discussed in more detail in the following section. This module allows components to safely make HTTP requests to the back-end and maintain knowledge of the type of information that may be received following any given request.

The `src/lib/components` directory contains all of the Svelte components that are intended for reuse in the application. In component-based frameworks like Svelte, any particular component is simply a piece of reusable code that encapsulates some particular behavior. A component may be something as simple as a button, or something as complex as some of the dashboard components implemented for this project.

The remaining directories in `src/lib` house a variety of smaller utilities. The WebSocket Connection Manager is hosted in `src/lib/websocket`, whereas the definition for the messages stores used by the Connection Manager are stored in `src/lib/stores`. The `src/lib/stores` directory also contains the code used to create new project state stores. As mentioned in Section IV, a project dashboard is rendered using a YAML-serialized layout specification loaded from the back-end into memory. An example of what this specification looks like is shown in Fig. 9. Whenever a project state is loaded into the front-end, this YAML data is used to set a store data structure in memory. This store is made generally available to all components on the dashboard, so that user interactions with any particular component that result in changes either to the component configuration (changing data inputs to a graph, for example) or the overall dashboard configuration (removing or adding a component, for example) are then reflected in the store.

```yaml
description: A new project.
panels:
- components:
  - component: ReactiveChart
    expanded: true
    settings:
      data_sources:
      - header:
          name: System
          timestamp: 2023-07-06 05:08:27.494051
        payload:
        - enabled: true
          name: CPU
          units: '%'
        - enabled: true
          name: RAM
          units: '%'
        - enabled: false
          name: Recording
          units: null
    title: Resource Utilization
  title: System Stats
title: Test Project
vertical: true
```

**Fig. 9 Sample of YAML-serialized state.**

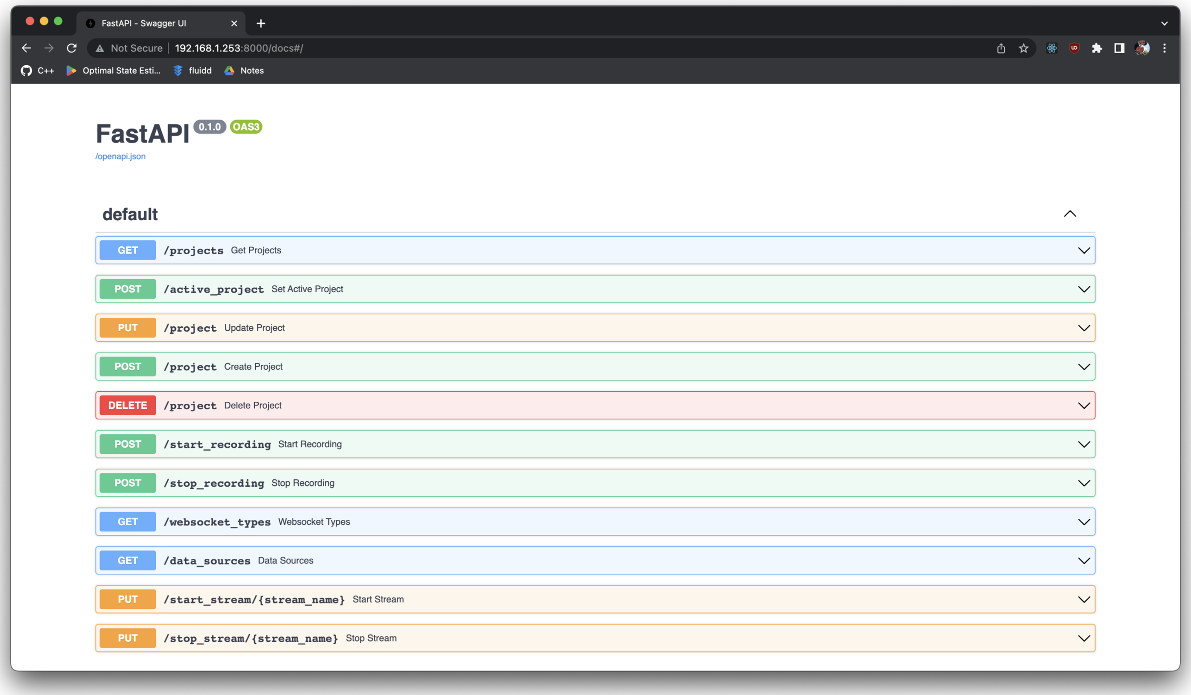
### C. Server

The back-end server is implemented using FastAPI, a framework for building high-performance web applications in Python[11]. Additionally, the server application is structured as a standard Python package and configured with a `pyproject.toml` file, allowing the application and its Python dependencies to be easily installed on various systems, even those without the Nix package manager[19]. The directory structure of the back-end is shown in Fig. 10. It should be noted that all of the version controlled code for the back-end is stored either at the root level (`./`) or in the `src/` directory. The `projects/` directory contains project-specific data, such as the dashboard configuration of a particular project, or any data products recorded with that project. In Fig. 10, the `projects/` directory contains a single project called `TestProject`. This directory structure is representative of what any new project created using the client application would have on the back-end. Any time a project is created, a new folder in the `projects/` directory is created containing `assets`, `data`, and `tests` folders.

Within the actual source code directory of the back-end server (in `src/ae8900/`) code is organized into four Python modules: `data_processing`, `infrastructure`, `models`, and `routers`. The `data_processing` module contains all of the code used to implement the Data Manager and Data Streams discussed at length in Section III. The `infrastructure` module contains generally useful utilities and tools that may be relevant to different parts of the application. Notably, this module contains dependencies that are shared across the back-end code base and safely accessed using a technique called dependency injection (DI). The `models` and `routers` modules are developed to take advantage of FastAPI's capabilities. `models` contains classes that define shapes of all of the unique types of data that are passed around the back-end server application using a Python module called Pydantic, which effectively enforces Python type hints at runtime, making it far easier to catch typing errors before they happen during runtime[20]. The `routers` module contains all of the various HTTP endpoints of the application.

```
.
├── projects
│   └── TestProject
│       ├── assets
│       ├── data
│       └── tests
└── src
    └── ae8900
        ├── data_processing
        ├── infrastructure
        ├── models
        └── routers
```

**Fig. 10 Back-end directory structure.**


FastAPI - Swagger UI

Not Secure | 192.168.1.253:8000/docs#/

C++ Optimal State Esti... fluidd Notes

FastAPI 0.1.0 OAS3

/openapi.json

default

```
GET     /projects                    Get Projects
POST    /active_project              Set Active Project
PUT     /project                     Update Project
POST    /project                     Create Project
DELETE  /project                     Delete Project
POST    /start_recording             Start Recording
POST    /stop_recording              Stop Recording
GET     /websocket_types             Websocket Types
GET     /data_sources                Data Sources
PUT     /start_stream/{stream_name}  Start Stream
PUT     /stop_stream/{stream_name}   Stop Stream
```

**Fig. 11 OpenAPI auto-generated documentation.**

When working across a multi-language stack such as the TypeScript/Python approach used in this project, it is common to run into the issue of code duplication. For this project, this issue manifested itself in the form of API implementation. The back-end server exposes a ReST API via HTTP. This API definition contains all of the necessary information that would be needed to implement it in a different language, like TypeScript, but it would be cumbersome to manually write an implementation of the same API, just in a different language. Additionally, this approach would scale poorly, as any modification to the API specification, or even just any new endpoint addition to the API, would result in additional, unnecessary work, with developers having to implement the same change in two different places. This approach also introduces the highly undesirable possibility of these two API instances getting out of sync with each other, such that one implementation of the API is written to expect a certain data structure returned from a certain request, where another implementation might expect a different data structure returned from the same request, creating issues that would be difficult to debug. Fortunately, FastAPI can automatically generate an OpenAPI specification, as shown in Fig. 11[21]. This specification describes a particular HTTP API using a JSON file, and solutions exist that can create an entire type-safe API implementation from an `openapi.json` file in various languages. For this project, the client application uses `openapi-typescript` and `openapi-fetch` to generate a TypeScript API from the `openapi.json` file exposed by the FastAPI back-end[22, 23]. This approach effectively eliminates the issues of code duplication discussed in the alternative approach, and when combined with `just` can be an automated task that runs whenever the back-end server is started, for example.

## V. Results

This section provides a description of the current state and capabilities of the software application developed for this project. As discussed previously, the example device under test is used to showcase these capabilities. In its current state, when users access the front-end client user interface via a web browser, they are presented with all of the projects stored on disk on the Raspberry Pi, as shown in Fig. 12. On this page, users can also opt to create a new project, which, when saved, will create a new directory structure similar to the one discussed in Section IV.

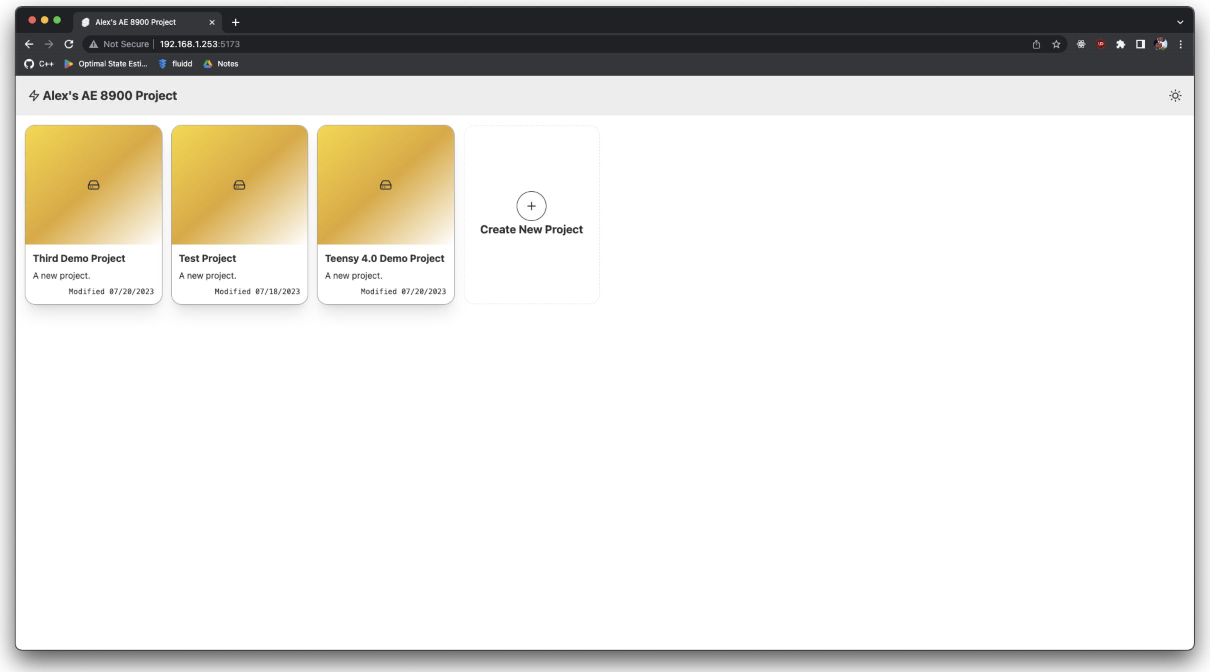


**Fig. 12 Example client home page with multiple configured projects.**

From the software home page, users can simply click on the desired project to load its corresponding dashboard. A dashboard configured for the example device under test shown in Fig. 2 is shown in Fig. 13. Notably, this dashboard displays real-time data from a variety of sources. In the upper left, the data displayed reflects quaternion elements sampled from the device under test. These are gathered using the Raspberry Pi's $I^2C$ interface. The graph on the lower left displays resource utilization data coming directly from the Raspberry Pi itself. The graph on the upper right displays a 3D visualization of the orientation data gathered from the device under test. The graph on the lower right displays analog voltage measurements taken using the MCC DAQ HAT. The HAT's channel 0 probes are connected to a GPIO pin on the microcontroller that is configured to turn on and off at a regular interval, as the graph shows. As discussed earlier, these dashboards are highly user-customizable, and can be rearranged into an arbitrary number of columns, with an arbitrary number of widgets in each column.

One notable feature of this example dashboard is the 3D orientation widget shown in the upper right of Fig. 13. This widget uses a `three.js` scene to render a real-time visualization of the quaternion data retrieved from the device under test[24]. It can be common to work with data products that may be complex and relatively unintuitive like quaternions. This widget serves to demonstrate some of the capabilities of the modern web that can be leveraged to make analysis of these complex data products more useful and intuitive for the user. Additionally, this widget affords a high degree of interactivity to users, allowing one to orbit, pan, and zoom around the model to achieve any particular desired viewpoint. By using existing libraries like `three.js` to handle more complicated routines – like the implementation of 3D rendering in a browser – development time and effort for widgets like this is relatively low. This makes it easy for developers to implement custom widgets for complex or novel data types when preexisting options may not be a good fit.

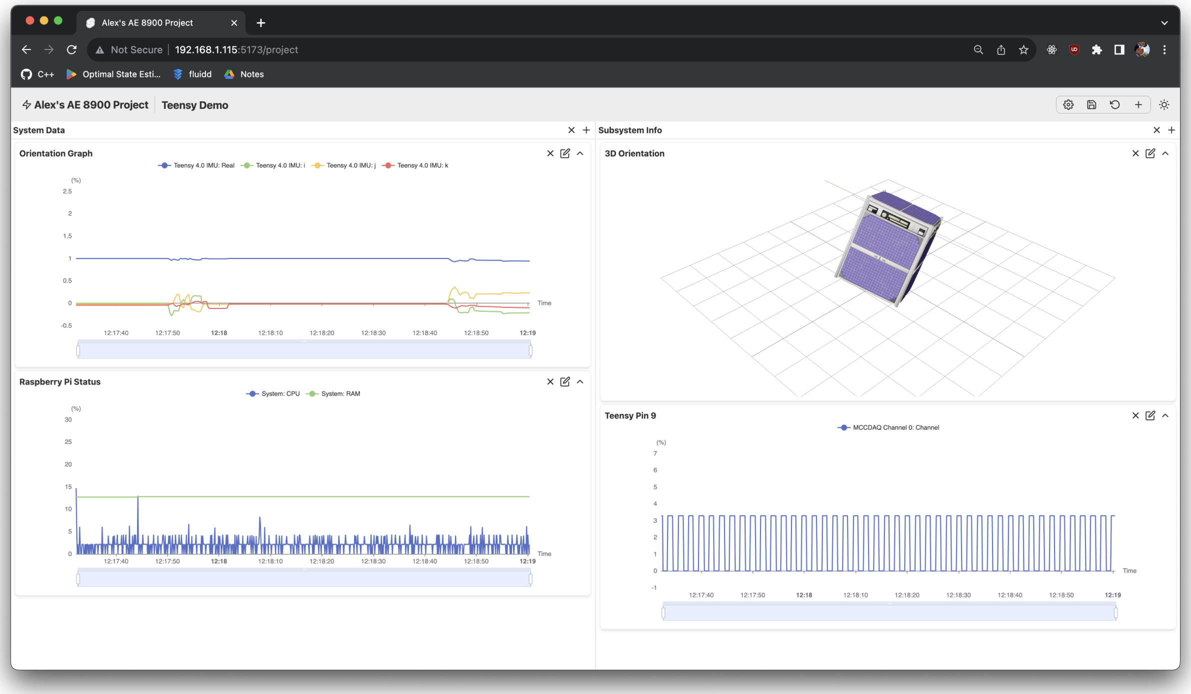
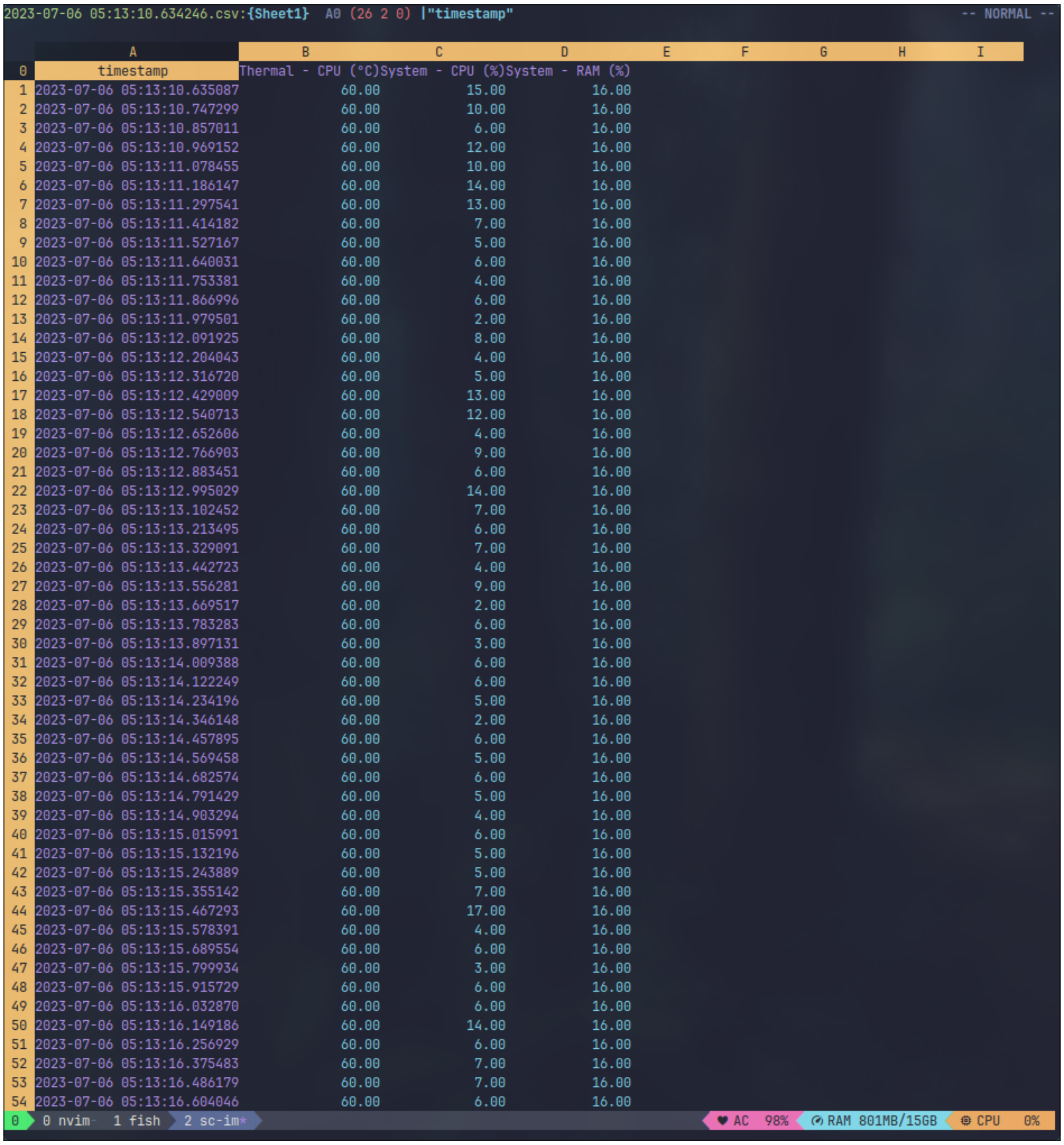


**Fig. 13 Example device under test dashboard.**

The current version of the software also has the ability to selectively record timestamped data at a regular sampling rate from any of the available data streams. A widget developed to allow users to control this recording capability is shown in the lower right of Fig. 5. This widget allows users to configure a recording with a high degree of granularity, down to particular fields within a certain data stream. Users can also configure the sampling rate at which these measurements are taken. It should be noted that this recording functionality has been implemented on the back-end server as an asynchronous coroutine, which allows the rest of the application to continue functioning while a recording is in progress. Currently, recordings are written to a CSV file in the `data` sub-directory of the active project directory. Each CSV file is named with the timestamp of the time when the recording was started, making it easy to prevent naming collisions within the file system. An example of a data recording written by the back-end server is shown in Fig. 14.

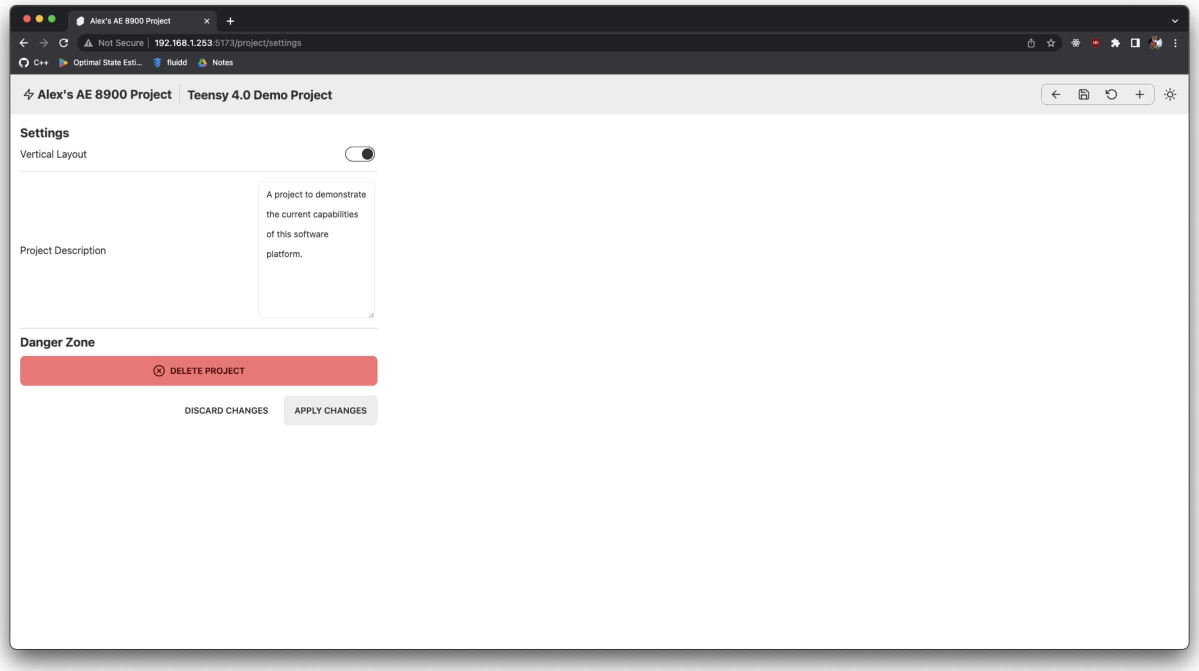


```
2023-07-06 05:13:10.634246.csv:{Sheet1}  A0 (26 2 0) |"timestamp"                                                -- NORMAL --

               A                    B                   C               D           E          F          G          H          I
 0         timestamp          Thermal - CPU (°C)System - CPU (%)System - RAM (%)
 1 2023-07-06 05:13:10.635087        60.00          15.00          16.00
 2 2023-07-06 05:13:10.747299        60.00          10.00          16.00
 3 2023-07-06 05:13:10.857011        60.00           6.00          16.00
 4 2023-07-06 05:13:10.969152        60.00          12.00          16.00
 5 2023-07-06 05:13:11.078455        60.00          10.00          16.00
 6 2023-07-06 05:13:11.186147        60.00          14.00          16.00
 7 2023-07-06 05:13:11.297541        60.00          13.00          16.00
 8 2023-07-06 05:13:11.414182        60.00           7.00          16.00
 9 2023-07-06 05:13:11.527167        60.00           5.00          16.00
10 2023-07-06 05:13:11.640031        60.00           6.00          16.00
11 2023-07-06 05:13:11.753381        60.00           4.00          16.00
12 2023-07-06 05:13:11.866996        60.00           6.00          16.00
13 2023-07-06 05:13:11.979501        60.00           2.00          16.00
14 2023-07-06 05:13:12.091925        60.00           8.00          16.00
15 2023-07-06 05:13:12.204043        60.00           4.00          16.00
16 2023-07-06 05:13:12.316720        60.00           5.00          16.00
17 2023-07-06 05:13:12.429009        60.00          13.00          16.00
18 2023-07-06 05:13:12.540713        60.00          12.00          16.00
19 2023-07-06 05:13:12.652606        60.00           4.00          16.00
20 2023-07-06 05:13:12.766903        60.00           9.00          16.00
21 2023-07-06 05:13:12.883451        60.00           6.00          16.00
22 2023-07-06 05:13:12.995029        60.00          14.00          16.00
23 2023-07-06 05:13:13.102452        60.00           7.00          16.00
24 2023-07-06 05:13:13.213495        60.00           6.00          16.00
25 2023-07-06 05:13:13.329091        60.00           7.00          16.00
26 2023-07-06 05:13:13.442723        60.00           4.00          16.00
27 2023-07-06 05:13:13.556281        60.00           9.00          16.00
28 2023-07-06 05:13:13.669517        60.00           2.00          16.00
29 2023-07-06 05:13:13.783283        60.00           6.00          16.00
30 2023-07-06 05:13:13.897131        60.00           3.00          16.00
31 2023-07-06 05:13:14.009388        60.00           6.00          16.00
32 2023-07-06 05:13:14.122249        60.00           6.00          16.00
33 2023-07-06 05:13:14.234196        60.00           5.00          16.00
34 2023-07-06 05:13:14.346148        60.00           2.00          16.00
35 2023-07-06 05:13:14.457895        60.00           6.00          16.00
36 2023-07-06 05:13:14.569458        60.00           5.00          16.00
37 2023-07-06 05:13:14.682574        60.00           6.00          16.00
38 2023-07-06 05:13:14.791429        60.00           5.00          16.00
39 2023-07-06 05:13:14.903294        60.00           4.00          16.00
40 2023-07-06 05:13:15.015991        60.00           6.00          16.00
41 2023-07-06 05:13:15.132196        60.00           5.00          16.00
42 2023-07-06 05:13:15.243889        60.00           5.00          16.00
43 2023-07-06 05:13:15.355142        60.00           7.00          16.00
44 2023-07-06 05:13:15.467293        60.00          17.00          16.00
45 2023-07-06 05:13:15.578391        60.00           4.00          16.00
46 2023-07-06 05:13:15.689554        60.00           6.00          16.00
47 2023-07-06 05:13:15.799934        60.00           3.00          16.00
48 2023-07-06 05:13:15.915729        60.00           6.00          16.00
49 2023-07-06 05:13:16.032870        60.00           6.00          16.00
50 2023-07-06 05:13:16.149186        60.00          14.00          16.00
51 2023-07-06 05:13:16.256929        60.00           6.00          16.00
52 2023-07-06 05:13:16.375483        60.00           7.00          16.00
53 2023-07-06 05:13:16.486179        60.00           7.00          16.00
54 2023-07-06 05:13:16.604046        60.00           6.00          16.00
0  0 nvim-  1 fish  2 sc-im*                                                 ♥ AC  98%  RAM 801MB/15GB  CPU  0%
```

**Fig. 14  Sample recorded data file.**

Each project dashboard also exposes a settings page to the user, which can be accessed by clicking the gear icon shown in the upper right corner of Fig. 13. Here, users can modify project-level settings, including the description of the project that is displayed on the landing page of the client application (Fig. 12) and the layout configuration of the project, which allows users to adjust whether dashboard divisions are either vertical or horizontal. Additionally, projects can be deleted on this page, which, if selected, will remove a project's folder and all its contents from the Raspberry Pi file system and redirect users to the landing page of the client application. The settings page is shown in Fig. 15.



Settings

Vertical Layout

Project Description: A project to demonstrate the current capabilities of this software platform.

Danger Zone

**Fig. 15 Project settings page.**

## VI. Future Work and Conclusion

### A. Future Work

The work detailed in this paper can be considered a first step in the direction of a fully-featured web application for CubeSat subsystem data acquisition and test automation. This early in the development life cycle, there are many areas where this project could be improved and a lengthy list of features that should be considered for future implementation to further build out the capabilities of this system.

Much of the work completed thus far has been to develop a maintainable, extensible software foundation that is generic enough to be applied to a variety of projects while at the same time providing enough value to make the system worth integrating into a project's development workflow. Now that this foundation has been established, development efforts should be focused on building features that extend the core functionality of the application. Specifically, much of the test automation aspect of this project has yet to be implemented. The tools and framework necessary to run asynchronous, long-running tasks like automated tests have been developed, as well as the capability to record detailed, multi-channel data concurrently, but there is still significant work to be done in implementing a robust, easy-to-use interface for test automation within the software application. Some initial work has been done to survey potential solutions for this automated test framework, and options such as Robot Framework and pytest are being considered, as well as potentially implementing a custom solution that is able to fully take advantage of Python's asynchronous programming capabilities.

While on the topic of future feature implementation, the dashboard layout system developed for this project has demonstrated its utility and versatility in displaying a variety of data types in real-time. Currently, the system has a focus on line graphs, but the 3D orientation visualization shown in Fig. 13 serves as a good example of a dashboard widget that takes full advantage of the capabilities of the modern web. Widgets like this are useful for visualizing more complex data types, like quaternions or Euler angles, and can be invaluable in validating these more complicated data products. A key focus of future development should be the creation of more of these dashboard widgets that enable users to visualize complex data in an intuitive and useful manner.

For the demonstrator device under test in this paper the application used $I^2C$ for communication, albeit not in a particularly generic way. In the future, it would be highly beneficial to build out the application's hardware protocol capabilities in such a way that hardware APIs can be described declaratively, perhaps using a YAML or TOML configuration file. This will keep the user-defined information needed to implement a new hardware device to a minimum, and ultimately help reduce code duplication. Other common hardware interfaces like SPI and CANBUS should also be considered for implementation in this scheme, so that the application may be used to test a wider range of hardware devices.

The three areas outlined above should be the immediate focus of future development efforts, as these features will most significantly extend the capabilities of the software tool. In addition to these features, however, there are a number of other areas that could benefit from future work. As discussed earlier, the development and runtime environments for this tool are defined entirely using Nix, which has made deployment to the Raspberry Pi a fairly trivial task. Currently, however, this development environment contains tools like auto-formatters and linters that are really only useful in a development setting. Additionally, Nix's relatively steep learning curve may deter new users from trying out the application for themselves. For these reasons, it would be beneficial to set up an automated pipeline for building custom Raspberry Pi OS images with all the necessary software already included. This way, users could simply download and flash an image to a MicroSD card, just as they would for any other Raspberry Pi operating system image. This would significantly decrease the barrier to entry for this application, and make initial setup of the application much more trivial. This sort of pipeline could be developed with GitHub Actions or some other continuous integration (CI) tool, such that new images would be produced automatically on every commit or merge to a particular branch in the git repository where the project source code is stored. A CI pipeline such as this could also be used to regularly perform important tasks on the code base, such as static application security testing (SAST) to automatically analyze source code for security vulnerabilities.

During future development cycles, an emphasis should be placed on testing new features and capabilities with real hardware, such as with the demonstrator device under test detailed in this paper. This will help keep the scope of the project focused on capabilities that directly affect users' ability to test hardware devices, while also helping to make sure that new features are properly validated and tested against new hardware.

### B. Conclusion

This project aimed to develop a proof-of-concept software application for data acquisition and test automation for CubeSat subsystems. In addition to these goals, the project maintained a fully open source approach to software development and also aimed to keep hardware costs low, such that the barrier to entry for interested users was low. Additionally, this project adopted a web-based software architecture as opposed to more traditional modes of software development. This approach afforded a number of benefits, especially in the context of the resource-constrained Raspberry Pi on which the software is deployed. To demonstrate the current capabilities of this project, a simple hardware demonstrator was developed that could be interfaced with via the Raspberry Pi's GPIO pins. A third party analog-to-digital data acquisition HAT was also incorporated into the design to demonstrate the using the software platform to gather analog voltage measurements. Significant efforts were made to develop the code base for this project in a maintainable, extensible manner, such that future development can be completed quickly and effectively. Time was also spent developing a highly reproducible development environment using Nix, which helps ensure that the application can run seamlessly across different devices. Ultimately, significant progress was made over the course of this development cycle to establish a strong foundation for this software tool, but further work is needed to fully realize its full potential.

## Appendix

The source code repository for this project is publicly available at `https://github.com/bustosalex1/ae-8900`.

## Acknowledgements

The CubeSat 3D model shown in Fig. 13 was obtained through the NASA 3D Resources online archive at `https://nasa3d.arc.nasa.gov/`.

## References

[1] Microsoft, *TypeScript: Documentation - TypeScript for the New Programmer*, 2023. URL `https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html`.

[2] Mozilla Foundation, *MDN Web Docs Glossary: Definitions of web-related terms - REST*, 2023. URL `https://developer.mozilla.org/en-US/docs/Glossary/REST`.

[3] Raspberry Pi Foundation, *Raspberry Pi OS*, 2023. URL `https://www.raspberrypi.com/software/`.

[4] Measurement Computing Corporation, *MCC 118 - Voltage Measurement DAQ HAT for Raspberry Pi*, 2020.

[5] "Stack Overflow Developer Survey 2023," , 2023. URL `https://survey.stackoverflow.co/2023/`.

[6] Microsoft, *Language Server Extension Guide | Visual Studio Code Extension API*, 2023. URL `https://code.visualstudio.com/api/language-extensions/language-server-extension-guide`.

[7] Python Software Foundation, *Initialization, Finalization, and Threads - Python 3.11.4 Documentation*, 2023. URL `https://docs.python.org/3/c-api/init.html`.

[8] Python Software Foundation, *Process-based parallelism - Python 3.11.4 Documentation*, 2023. URL `https://docs.python.org/3/library/multiprocessing.html`.

[9] Python Software Foundation, *Asynchronous I/O - Python 3.11.4 Documentation*, 2023. URL `https://docs.python.org/3/library/asyncio.html`.

[10] Svelte.js, *Svelte Docs - Introduction*, 2023. URL `https://svelte.dev/docs/introduction`.

[11] Ramírez, S., *Features - FastAPI*, 2023. URL `https://fastapi.tiangolo.com/features/`.

[12] Conda Community, *conda-lock - Documentation*, 2023. URL `https://conda.github.io/conda-lock/`.

[13] Marriott, N., *tmux/tmux Wiki - Getting Started*, 2022. URL `https://github.com/tmux/tmux/wiki/Getting-Started`.

[14] Rodarmor, C., *Just Programmer's Manual - Introduction*, 2023. URL `https://just.systems/man/en/`.

[15] Dolstra, E., de Jonge, M., and Visser, E., "Nix: A Safe and Policy-Free System for Software Deployment," *Proceedings of the 18th USENIX Conference on System Administration*, USENIX Association, USA, 2004, p. 79–92.

[16] "Repology - Repository statistics," , 2023. URL `https://repology.org/repositories/statistics`.

[17] Narlock, T., *tmuxp 1.28.1 Documentation*, 2013. URL `https://tmuxp.git-pull.com/`.

[18] Svelte.js, *SvelteKit Docs - Routing*, 2023. URL `https://kit.svelte.dev/docs/routing`.

[19] Reitz, K., and Schlusser, T., *The Hitchhiker's Guide to Python*, 1st ed., O'Reilly Media, 2016.

[20] Pydantic Company, *Welcome to Pydantic*, 2023. URL `https://docs.pydantic.dev/latest/`.

[21] OpenAPI Initiative, *What is OpenAPI?*, 2022. URL `https://www.openapis.org/what-is-openapi`.

[22] Powers, D., *OpenAPI TS - Introduction*, 2023. URL `https://openapi-ts.pages.dev/introduction/`.

[23] Powers, D., *OpenAPI Fetch - Introduction*, 2023. URL `https://openapi-ts.pages.dev/openapi-fetch/`.

[24] three.js, *Fundamentals - three.js manual*, 2023. URL `https://threejs.org/manual/#en/fundamentals`.