T.R.A.C.K.
Telemetry Rendering And Capture Kit

SAE TEXAS A&M UNIVERSITY

Final Report

Alonso Peralta Espinoza

Jack Williams

Brayden Bailey

Cameron Stone

Campbell Wright

Justin Busker

Department of Computer Science

Texas A&M University

04/29/2026

Table of Contents

1 Executive summary 2
2 Project Background 4
2.1 Updates to the proposal design (if applicable) 4
2.2 System description 5
2.3 Complete module-wise specifications 7
3 Final Design 9
Effectiveness and Improvements 12
Summary 13
3.1 Updated implementation schedule 13
3.2 Updated validation and testing procedures 14
3.3 Updated division of labor and responsibilities 15
4 Implementation Notes 16
5 Experimental Results 23
6 Users manual 25
7 Course Debriefing 34
8 Budgets 35
9 References 38

1 Executive Summary

Formula SAE Electric vehicles generate high volumes of real-time telemetry across powertrain, energy storage, and safety subsystems. This telemetry is typically distributed over the Controller Area Network (CAN), a bus standard widely used in automotive and embedded systems because it supports prioritized messaging, robustness under electrical noise, and a simple shared-medium wiring model. A driver display sits at a critical boundary between the embedded network and the human operator. It must convert raw CAN frames into actionable information with high readability, stable visuals, and low latency. At the same time, the team needs data logging to enable post-run analysis for performance tuning, reliability improvement, and safety verification. Beyond the car itself, the pit crew requires live visibility into vehicle state during a run, and position and speed data from a GPS module provides an additional layer of situational awareness for both the driver display and post-session analysis.

In professional motorsport, these needs are commonly met by integrated dash displays and dash loggers. Products like the PLEX SDM-330 [1] emphasize configurability and compactness for CAN-based dashboards, with variants that can include logging and additional sensing upgrades. AiM products like the MXm emphasize tight integration of data acquisition and display for racing use cases, including the ability to capture crucial racing signals and support analysis workflows. While these products show what is possible, student teams often face barriers when adopting them fully, including high cost, constrained customization beyond vendor tooling, and workflow friction when rapid changes are needed during test days or competition. Student teams also benefit strongly from open data formats and open configuration, because tooling and expertise change year to year.

This project targets a specific operational need for Texas A&M Formula SAE: an embedded driver display and logging system that is race-appropriate yet designed for student iteration, paired with a web based pit tool for real-time monitoring and configuration. The system must be configurable by the pit crew quickly, maintainable across multiple seasons, and compatible with open analysis workflows. Embedded Linux is a strong foundation for the on-car side because it provides stable process isolation, mature storage and networking capabilities, and standardized CAN integration through SocketCAN. SocketCAN treats CAN devices as network interfaces and exposes a socket API that mirrors common TCP/IP development patterns, which lowers development complexity and improves testability. Additionally, development and validation can leverage linux-can can-utils, a widely used user-space toolkit for displaying, recording, generating, and replaying CAN traffic. On the pit side, a React-based web configurator backed by cloud infrastructure allows engineers to build display layouts, define CAN signal bindings, and stream live telemetry from the car over WebSocket without requiring any local tooling installation.

The goals of this project are to deliver a fully operational driver display with configurable multi-screen layouts, continuous binary data logging, live pit telemetry streaming, and GPS-based position and speed acquisition, all running as isolated processes on a single embedded platform. The system must meet design constraints appropriate for a racing vehicle: display updates must remain stable and low-latency under CAN bus load, the hardware must tolerate the electrical noise environment of an FSAE vehicle, and every component must be understandable and modifiable by a new team member without proprietary tooling or vendor support.

This Critical Design Review marks the transition from individual component development to system integration and validation. All major modules are implemented; the remaining work is wiring them together, validating end-to-end data flow, and completing the GPS and log upload pipelines. Validation will proceed at multiple levels: CAN signal decoding is verified using can-utils frame replay against known DBC definitions, the web stack is tested locally using the Firebase emulator suite, hardware power delivery and signal integrity have been validated on the fabricated PCB, and end-to-end integration testing

will verify that configuration changes authored in the browser propagate to the car display within an acceptable latency bound. The remainder of this document covers the full system description, detailed module-level design specifications, validation and testing procedures, implementation schedule, division of labor, and preliminary results.

2 Project Background

2.1 Updates to the proposal design (if applicable)

During implementation, two additions to the original scope were identified and shipped that significantly increased system value with relatively low integration cost.

The first is a GPS reader process. As integration progressed, it became clear that adding a dedicated GPS process was a high-return investment. A GPS module provides independent position and speed data that does not depend on the vehicle's CAN configuration, is immediately useful for post-session track mapping and lap analysis, and integrates naturally into the existing multi-process architecture without modifying any existing process. The templated shared-memory queue let us instantiate a typed gps_queue (single-producer, multi-consumer) alongside the existing telemetry_queue to serve GPS data to the cloud bridge and downstream consumers. The GPS reader joined the pipeline as a fifth isolated process alongside can-reader, graphics-engine, data-logger, and cloud-bridge, with its own systemd unit, track-gps-reader.service. The frontend exposes the data through a dedicated GPS Mapping page that plots the live track in real time over the /ws/client socket.

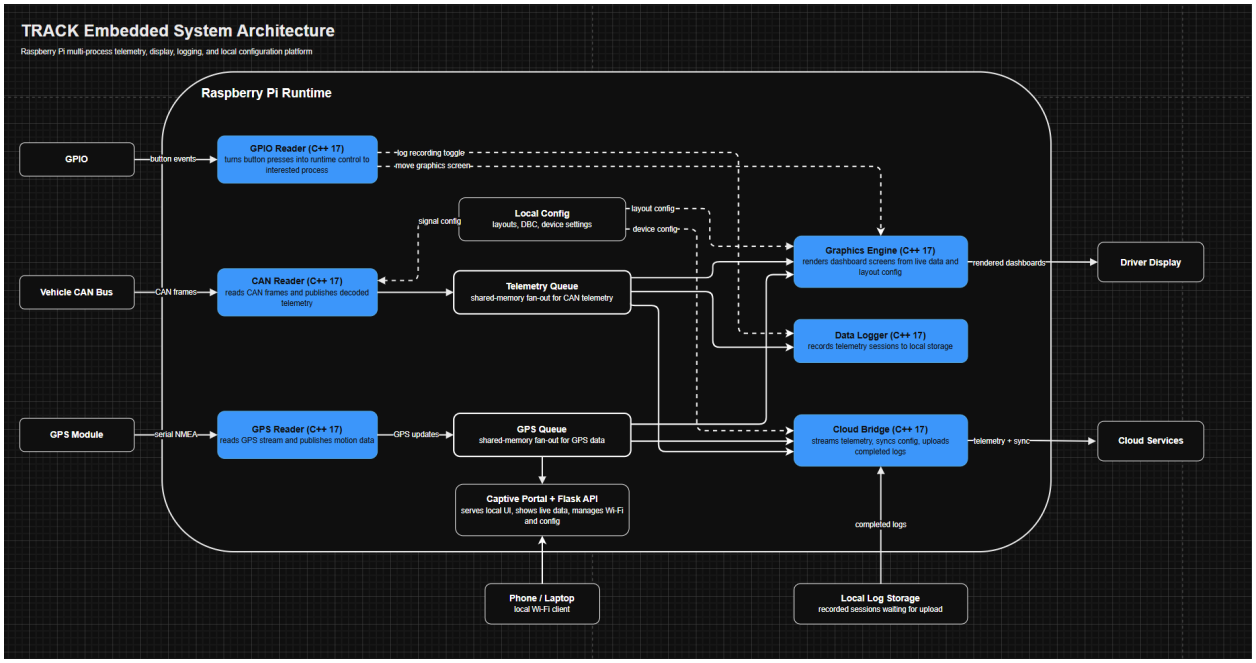
The second addition is a full cloud migration. The original proposal described a web-based configuration interface that wrote JSON config files to local disk and triggered a SIGHUP to reload running services. While functional as a proof of concept, this model had meaningful operational limitations. Config was tied to the physical machine, there was no user authentication, and multiple engineers could not work from separate devices without manual file transfers. During development, Firebase was identified as a well-supported, zero-cost-at-our-scale platform that resolved all of these issues. Firebase Authentication provides Google OAuth out of the box, and Firestore provides per-user cloud persistence with no infrastructure to operate. The migration from disk-based config to Firestore was implemented without changing the embedded side. The cloud-bridge process handles the boundary between the cloud-connected web system and the on-car pipeline, receiving config_update messages over /ws/pi, writing them to /opt/track/config/graphics.json or display.dbc, and sending SIGHUP to the consumer. Pi-side edits flow back up via graphics_upload and dbc_upload messages on the same socket. This shifts the product from an MVP tied to a single machine to a genuinely deployable tool that any team member can access from any device, with cross-tab realtime sync layered on top via /ws/client.

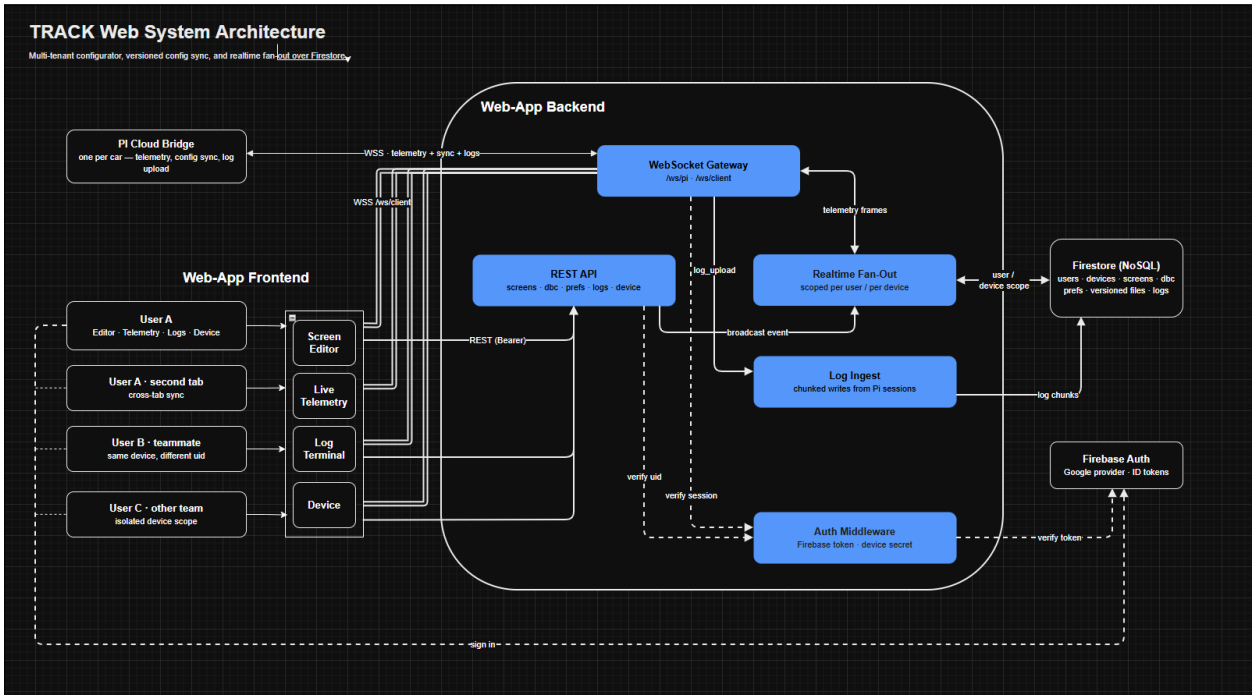
For cloud persistence, a self-hosted PostgreSQL or SQLite database was considered alongside Firebase. A self-hosted database would have eliminated any dependency on a third-party platform, but it introduces operational overhead. The team would need to maintain a server, handle backups, and manage uptime, which is poorly matched to a student team without dedicated infrastructure. Firebase's free tier covers the usage levels this project will ever reach, requires no server administration, and includes authentication and real-time listeners as first-class features.

For GPS, an alternative was to source position and speed from the vehicle's existing CAN network if the BMS or inverter ECU published those signals. This was ruled out because GPS availability on the CAN bus depends on vehicle configuration decisions made by other subteams, which introduces an external dependency outside our control. A dedicated GPS module is self-contained, costs under $30, and produces standard NMEA sentences that are well-documented and easy to parse. For real-time pit telemetry, a polling REST API was considered as an alternative to WebSocket streaming, but polling introduces unnecessary latency at any interval short enough to be useful during a live run.

The original itemized budget totaled $239. Two additions affect this figure. A u-blox-compatible serial GPS module was added at an estimated cost of $25, bringing the hardware total to $264. Firebase infrastructure carries no additional cost under the free tier given the team's usage profile. Authentication, Firestore reads and writes, and WebSocket relay through the Express backend (hosted on Fly.io's free allotment) all fall well within the no-cost quotas for a project of this scale. No other budget line items have changed.

2.2 System description





The system description consists of high-level block diagrams and a functional description of the different parts and interfaces. The canonical diagrams for this section are maintained as drawio files: one for the on-car embedded pipeline and one for the cloud and client side of the system.

From a high level, the system breaks into three main components: the driver display, the backend web server, and the client application.

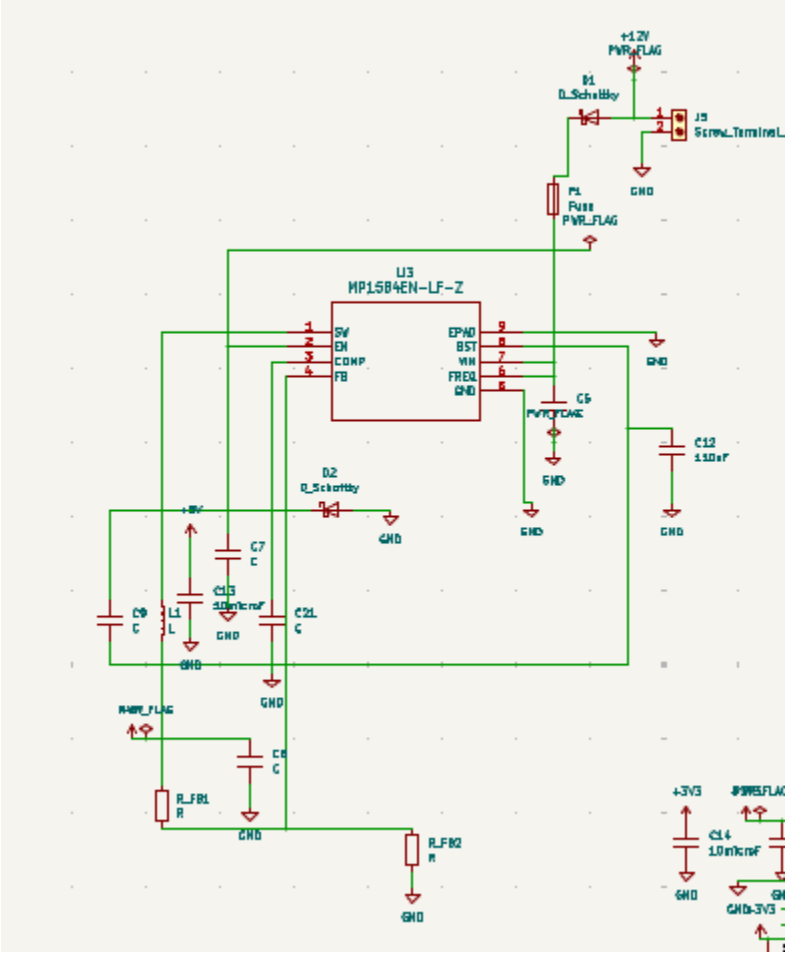
The driver display is mounted directly on the vehicle and is responsible for reading real-time data and visualizing it during a race. It is built around a Raspberry Pi connected to our custom PCB and daughter board, with the rendered output shown on an 800-by-480 HDMI display. The Pi runs five isolated processes that communicate through shared-memory queues, so a crash in any one process cannot bring down the others. The CAN Reader ingests raw CAN frames from the vehicle bus and decodes them according to the loaded DBC file. The Graphics Engine reads decoded telemetry from shared memory and renders the configured widget layout at 60 FPS on a 10-by-6 grid, supporting gauge, bar, number, indicator, and graph widget types. The Data Logger batches telemetry to the SD card so the team can run post-race analysis. The Cloud Bridge maintains a WebSocket connection to the backend to stream live telemetry, push uploaded logs after a session ends, receive configuration updates, write the new graphics or DBC config to disk, and signal the consumer process to reload the file. The GPS Reader publishes NMEA position and speed into its own shared-memory queue, which the Cloud Bridge forwards to the cloud. A Python captive portal is also served from the Pi for first-time setup, which covers Wi-Fi credentials and team-member email registration.

The Backend Web Server is responsible for managing all communication between the driver display and the clients. It runs as an Express application deployed to the cloud and exposes two WebSocket endpoints. The first carries traffic to and from the Pi, authenticated by a device ID and secret pair, and carries telemetry, GPS, log uploads, and Pi-initiated config edits. The second is for the browser, authenticated by a Firebase ID token, and carries live telemetry along with cross-tab editor events. It also hosts REST endpoints so the client can manage screen layouts, DBC content, log history, per-user tab preferences, and team-member access. Telemetry frames received from the Pi are fanned out to every connected browser session whose user is a team member of that device, so multiple users can see live data at the same time. The backend also pushes config in the other direction: every screen save rebuilds the full layout payload and sends it to the Pi over the WebSocket. All persistent data lives in Firestore, including users, devices, and their team-member lists, screens, DBC content, screen preferences, and chunked logs. There is no separate database to operate.

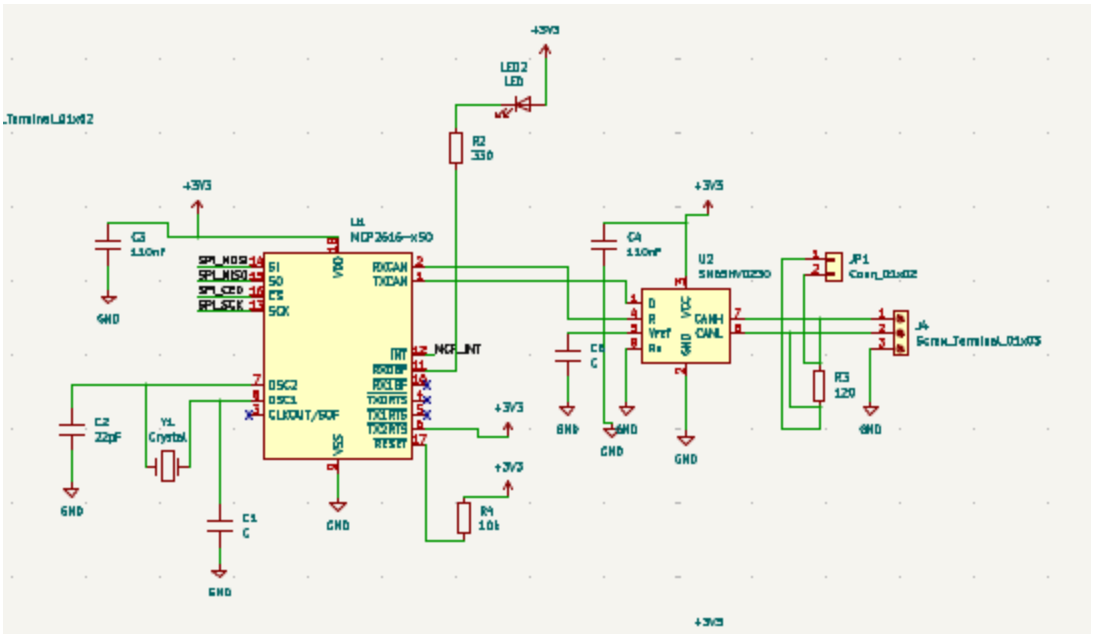
The Client Application is a React web app deployed as a separate cloud service. Pit crew members sign in with Google through Firebase Authentication, after which they are auto-linked to a Pi by matching their email against the device's team-member list. The app is organized into five pages reachable from a top-bar navigation. Screen Editor is the drag-and-drop layout builder on the same 10 by 6 grid that the on-car renderer uses, with widget property configuration and CAN signal binding through an inline DBC frame editor. Live Telemetry renders one sparkline card per incoming signal in a dynamic grid. GPS Mapping plots the live track in real time with naive lap detection. Log Terminal is a split-panel viewer with a day-level history accordion, session grouping, CSV export, and a live feed. Device shows the linked device ID, Pi online status, and a team-member email editor. Multi-screen support lets users create, save, rename, pin, drag-to-reorder, and bulk-delete display profiles, with per-user pin and order preferences persisted to the backend and synced across open tabs in real time. The application is intuitive, with a drag-and-drop layout for creating screens and customizable configuration so teams can easily control what data they wish to see.

2.3 Complete module-wise specifications

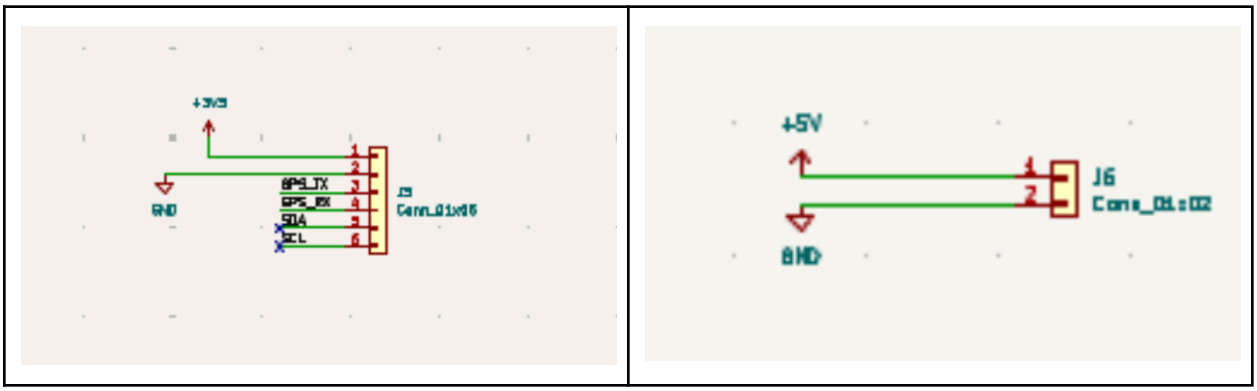
The power supply converts the car's 12V nominal supply to a regulated 5.0V input using the MP1584EN synchronous step-down converter. The input path consists of a Schottky diode for reverse polarity protection, followed by a 3A fuse for overload protection. A 22 µF capacitor provides local energy storage and suppresses input voltage transients from the car's electrical environment. The MP1584EN operates at a fixed 1.5MHz switching frequency. Key components connected to the converter are a 33 µHz/3A output inductor, a 100 nF bootstrap capacitor between the BST and SW pins, a 22 µF output capacitor, and a 100 nF decoupling capacitor on the internal VCC pin. Another Schottky diode serves as the freewheeling diode for the system.



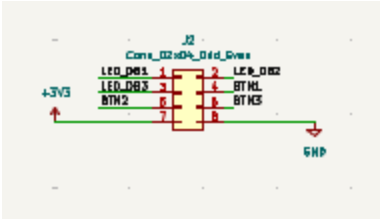
The CAN interface consists of two ICs: the MCP2515-I/SO SPI-to-CAN controller and the SN65HVD230DR 3.3V CAN transceiver. The MCP2515 communicates with the Pi over SPI at up to 10 MHz. An 8 MHz crystal with two 22 pF load capacitors provide the oscillator clock. A 10 kΩ pullup resistor holds the active-low RESET pin high during normal operation. The interrupt output connects to GPIO25 to be configured as a falling-edge interrupt input. The SN65HVD230DR converts TTL-level signals from the MCP2515 TXCAN and RXCAN outputs to differential CAN bus levels. The VCC is a 3.3V input with a 100 nF decoupling capacitor, and the Rs pin is tied to GND for high-speed mode, which supports bit rates up to 1 Mbps. The last Vref pin is left floating.

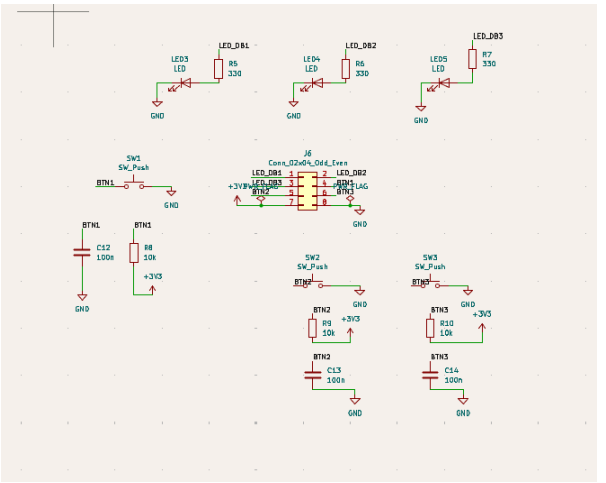


The SparkFun NEO-M9N breakout connects to the main PCB via a 6-pin female header, although only four of the six pins are active (GND, 3v3 power, TX, and RX). The TX pin is where the GPS transmits NMEA data, and the RX pin is used by the Pi to send UBX configuration commands. SDA and SCL pins are broken out but reserved for future I2C use. The Digiwise display connects to the system through three interfaces: HDMI video (connected to the main Pi board, not the PCB), power (5V connected to the main PCB through a two-pin power and GND header), and capacitive touch input (connected from the display's five-pin terminal to one of the Pi's USB ports).



The daughter board connects to the main PCB through an 8-pin Molex Micro-Fit 3.0 connector. On the daughter board, there are three distinct LEDs and buttons. Each button circuit uses a 10 kΩ pullup resistor connected to the 3v3 power and a 100 nF debounce capacitor. Each LED circuit uses a 330 Ω resistor to limit the current, yielding approximately 4 mA per LED.





	Quantity	Amount per Unit	Cost per Unit	Total Amount ($)
3D printing filament	1	1	13.99	13.99
Sparkfun GPS Breakout	1	1	74.95	74.95
Raspberry Pi 4b	1	1	75.00	75.00
CAN Hat	2	1	16.31	32.62
4.3 inch display	1	1	85.00	85.00
Main PCB	1	1	259.00	259.00
LED and Button Daughterboard	1	1	49.61	49.61
Total Cost				590.17

3 Final Design

Roles and responsibilities:

Jack Williams (Team Leader):

Jack Williams as team leader has been responsible for managing the team throughout the project. Below is a list of responsibilities he has been in charge of related to project management.

1. Organizing and leading team meetings. This happens every Monday virtually over discord.
2. Submitting purchase order forms and communicating with Ankitha about said purchases
3. Preparing weekly report forms.

Outside of project management responsibilities, Jack has been helping with other parts of the project. He has done extensive development on the graphics engine portion of the project. Notably the individual widgets (gauge, bar graph, indicator light, etc) that can be placed as components on the screen. In addition to this, he has also contributed to the graphics engine portion of the project by allowing for multi-page support, allowing the driver to be able to toggle multiple different display profiles. He has also assisted in testing of multiple systems throughout the development process. Lastly, he is in charge of developing the enclosure for T.R.A.C.K., which is still in progress.

Alonso Peralta Espinoza:

Alonso Peralta Espinoza served as one of the two full-stack developers on the web app, with the bulk of his work concentrated on the frontend. On the frontend, he delivered the drag-and-drop widget editor and 10x6 grid canvas using @dnd-kit/core and @dnd-kit/sortable, the full widget palette (gauge, bar, number, indicator, and graph), and widget property configuration with CAN signal binding wired through the inline frame editor. He built out the multi-screen system, which lets users create, save, rename, pin, drag-to-reorder, and bulk-delete display profiles, with per-user pin and order preferences persisted through the prefs API and synced across open tabs in real time. He developed the four non-editor pages: Live Telemetry (WebSocket-driven dynamic grid of sparkline cards), GPS Mapping (live track plotting with equirectangular projection, scroll-zoom, click-pan, and naive lap detection), Log Terminal (split-panel viewer with day-level accordion, session grouping, and CSV export), and Device (device ID, Pi online status, and team-member email editor).

On the cloud side, he owned the frontend integration with Firebase. This included the auth gate state machine in AppWithAuth.tsx, Google sign-in, the authFetch helper that attaches a Firebase ID token to every backend call, the onboarding flow for unlinked users, and the user document write to users/{uid} on first sign-in. He also stood up the TelemetryProvider WebSocket context, which manages the /ws/client connection with auto-reconnect, fans out telemetry and GPS frames, and dispatches the cross-tab editor events that keep sibling tabs in sync.

His backend contributions were targeted rather than primary. The backend module, including the REST API, the Firestore data layer, and the WebSocket gateway, was owned by his partner. Alonso stepped into the backend when the frontend needed an adjusted broadcast payload or a missing field on an API response, with the agreed module boundary keeping those edits clean.

Brayden Bailey:

Brayden Bailey acted as the lead developer for the backend web server. He was responsible for designing and implementing a centralized system that would enable real time communication between the driver display and client applications. He developed a modular backend that uses WebSockets to support low latency, bidirectional data flow allowing telemetry to be broadcast to multiple clients while also allowing users to push configuration updates back to the display. He developed the API for managing display layouts, CAN signal definitions, and user to vehicle mapping. Additionally he focused on system performance and reliability by handling connection management, supporting multiple concurrent clients, and enabling dynamic updates without interrupting operations to ensure a seamless user experience.

Cameron Stone:

Cameron acted as the lead embedded software engineer and system architect for T.R.A.C.K., designing the overall distributed architecture and authoring most of the embedded codebase (33 source files across 6 subsystems). He architected the system as a multi-process pipeline on Raspberry Pi 4, where independent C++ services communicate through a custom lock-free shared memory broadcast queue. He designed and implemented the core embedded services that feed the rest of the system. The CAN reader daemon opens a raw Linux CAN socket, captures live CAN frames, and decodes signals using a regex-based DBC parser he wrote to handle the industry-standard Vector DBC format - extracting message and signal definitions, bit layout, and byte ordering. Decoded telemetry is broadcast to shared memory for consumption by downstream services. The data logger writes compact 24-byte binary log entries to a timestamped file. The cloud bridge aggregates telemetry at 20 Hz into JSON and streams it to the backend, while also listening for remote configuration updates and applying them locally through atomic file writes and SIGHUP signaling to downstream services. He built the GPS reader's UART serial interface for NMEA capture from the NEO-M9N module, and the captive portal - a Flask server providing WiFi provisioning via nmcli, DBC file management, captive portal detection, serving both the landing page and a React-based display editor SPA. Beyond the application code, he authored the common libraries used by the entire system: the lock-free broadcast queue with acquire/release memory semantics, POSIX shared memory wrappers, and typed telemetry and GPS message queues.

Campbell Wright:

Campbell leads hardware design and PCB development. He designed the complete schematic for both the Main PCB and Daughter Board in KiCad 9.0, including the MP1584EN buck converter power supply, MCP2515/SN65HVD230 CAN bus interface, SparkFun NEO-M9N GPS integration, Raspberry Pi 40-pin header pin assignment, display power and USB passthrough, and daughter board connector wiring. He validated the power supply feedback divider and component selection using TI WEBENCH Power Designer. He completed the PCB layout as a Pi 4B hat form factor with the GPIO header on B.Cu and all components on F.Cu, and submitted both boards to JLCPCB for fabrication. He will assemble the prototype boards, perform power bench testing, and lead hardware integration testing including CAN loopback verification, GPS lock testing, and display power validation.

Justin Busker:

Justin Busker acted as an embedded software engineer with a primary focus on the display graphics engine. Justin helped design and create the graphics pipeline, including: reading display configurations from a given config file, assigning widgets to respective CAN ID's for value output, design of the configurations JSON structure, creation of websocket structure for sending log files over the internet, and connecting the datalogger to the cloudbridge. He has also done extensive testing and debugging on these parts through testing the graphics engine using VCAN to simulate live CAN bus messages, and ensuring that websockets sent with data log chunks are sending information cleanly with great stability.

Management:

The management approach for T.R.A.C.K. has emphasized autonomy, accountability, and consistent communication. As team leader, the strategy has been intentionally hands-off in day-to-day execution, while maintaining structured checkpoints to ensure alignment and progress. Rather than micromanaging individual contributors, team members are given ownership over their respective subsystems, with the expectation that they independently drive development forward.

To maintain coordination, the team holds weekly meetings every Monday over Discord. These meetings serve three primary purposes:

- Review progress from the previous week
- Identify blockers or integration issues
- Assign or adjust tasks for the upcoming week

This cadence has proven effective in balancing independence with accountability. Team members have clear expectations but retain flexibility in how they meet them.

This approach aligns closely with concepts from *The Five Dysfunctions of a Team*, particularly in addressing:

- **Absence of Trust:** The team has built strong trust by consistently delivering on assigned tasks and openly communicating challenges. Members feel comfortable asking for help or admitting when something is not working.
- **Fear of Conflict:** Technical disagreements such as cloud integration have been resolved through open discussion rather than avoidance. This has led to stronger design decisions.
- **Lack of Commitment:** Weekly meetings ensure that all members leave with clearly defined responsibilities, reducing ambiguity and increasing commitment to deliverables.
- **Avoidance of Accountability:** Progress is visible during weekly check ins, creating a natural accountability mechanism without requiring heavy oversight.
- **Inattention to Results:** The team has maintained a strong focus on the end goal of a fully integrated, race-ready telemetry system rather than individual contributions in isolation.

Overall, this management style has worked well for a technically capable and self-motivated team. The project is currently ahead of schedule, indicating that the balance between autonomy and structure is effective.

Effectiveness and Improvements

The current management approach has been effective, as evidenced by:

- The team being ahead of schedule
- Successful parallel development across subsystems
- Minimal rework due to early design alignment
- Strong team cohesion and communication

However, some minor improvements have been identified:

- **More explicit task tracking:** While the current system works, introducing a lightweight task board (e.g., Trello or GitHub Projects) could improve visibility of in-progress work.
- **Earlier integration checkpoints:** As the project moves into full system integration, more frequent intermediate integration tests will reduce risk.
- **Documentation standardization:** Ensuring consistent documentation across subsystems will improve maintainability for future team members.

These adjustments are being implemented as the project transitions from development to integration and validation.

Summary

The T.R.A.C.K. team has adopted a management approach centered on trust, ownership, and structured communication. By combining a hands off leadership style with consistent weekly alignment, the team has avoided common pitfalls outlined in *The Five Dysfunctions of a Team* and maintained strong momentum throughout development. This has resulted in a cohesive, productive team environment and a project that is currently ahead of schedule and on track for successful integration and validation.

3.1 Updated implementation schedule

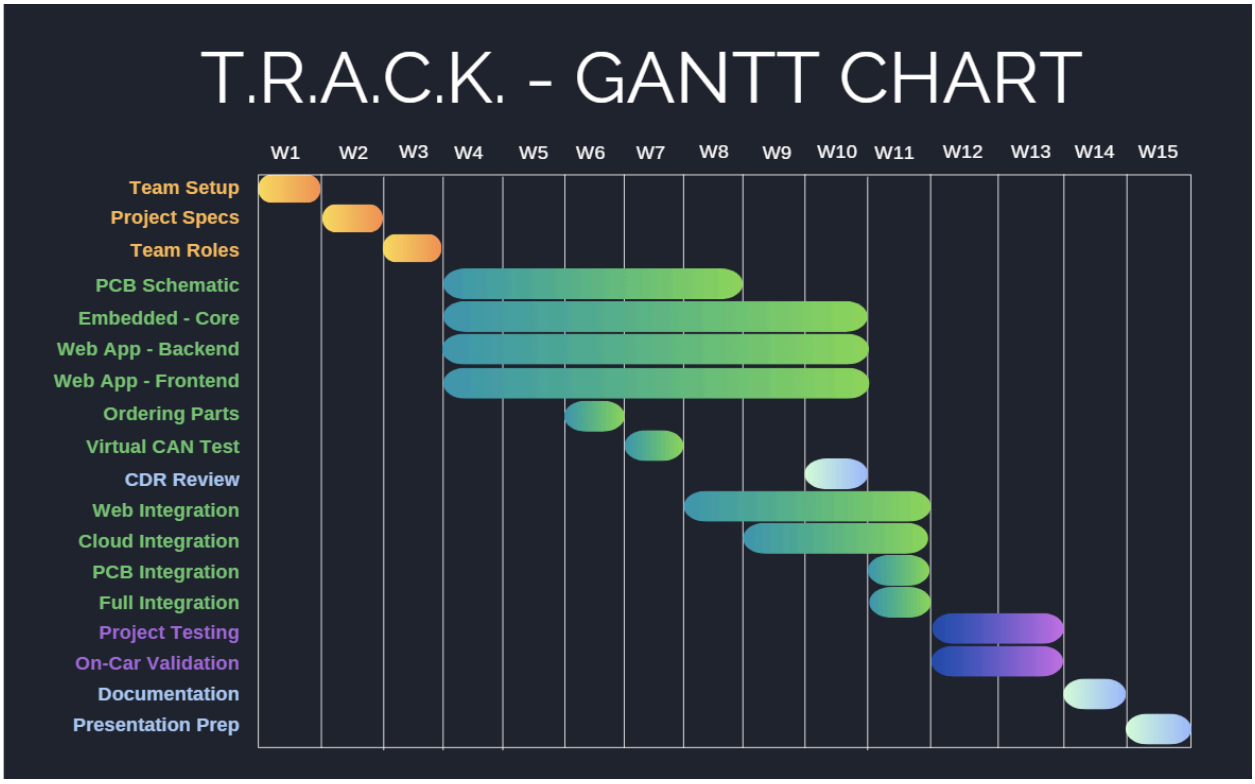


Figure 1: Up-to-date Gantt Chart for T.R.A.C.K. – we started design in parallel between PCB, Embedded, Backend, and Frontend. We were ahead of schedule leading to our decision to include our two stretch goals, GPS module and Cloud Integration, with high ROI taking our project from a proof-of-concept to a product.

Implementation was broken into four sequential phases. The first phase established project coordination and requirements. The second phase ran three parallel development tracks: embedded core processes, web backend and frontend, and PCB schematic design. Parallel execution was possible here because the embedded pipeline and web system share a clean interface boundary (the /ws/pi config-sync contract), which let teams develop independently against agreed contracts. The third phase was integration, where the parallel tracks converged. Web integration, cloud integration, and PCB integration all completed before full system integration closed the phase. The fourth phase covered bench testing, on-car validation, documentation, and presentation.

All software dependencies have been resolved. The only remaining hardware dependency is the 3D-printed enclosure, which is gating final PCB soldering. Full integration, meaning the tethering between the web application and the Raspberry Pi via the cloud bridge, is complete on the software side and exercised end-to-end against a commercial CAN interface.

The critical path ran through embedded core development, cloud integration, and full system integration. The primary schedule risk identified at CDR was PCB bring-up. The planned mitigation was

to run software integration against a commercial CAN interface in parallel, and that mitigation was executed, which ensured the software pipeline was fully validated regardless of board status. Testing and on-car validation overlapped intentionally, since findings from vehicle testing drove targeted bench re-tests.

3.2 Updated validation and testing procedures

Validation of our driver display will focus on the operational capabilities of the prototype to meet the needs of both the SAE Electric team's vehicle, and the individual engineers and drivers. While we would enjoy getting to install our prototype on the vehicle, it may not be realistic in our given timeframe for this project due to SAE's schedule. CAN bus testing will have to be simulated through VCAN. We will maintain the integrity of the real car system by having a separate Raspberry Pi generating CAN signals, and our driver display Pi receiving and operating on these signals.

For system validation, our primary benchmark is ensuring that the system delivers accurate, low-latency telemetry to both the driver and pit crew while maintaining robustness and usability. We will test these through four key process indicators: latency, accuracy, reliability, and usability. Latency will be measured across the time from a sent CAN message to the driver display (<50ms) and the pit dashboard visibility. The pit dashboard is more lenient on display of telemetry. The Raspberry Pi may not always have internet connection while driving, so we will focus more on ensuring eventual propagation of data when the Raspberry Pi is connected. Accuracy will be measured across the correct decoding and representation of CAN signals and GPS data. Reliability is tested through stability of the Pi under high bus load / operation. While they are racing the car, we don't want the Pi to overheat or crash. Usability will be determined based on ease of configuration and the clarity of display information. Can a non-technical user easily edit display configs, view telemetry, and understand what they are looking at better as compared to PLEX software.

An end-to-end telemetry flow test can be conducted where CAN messages are generated from another Pi and transmitted through the entire pipeline: from the CAN bus → embedded system → display rendering → logging → WebSocket → pit dashboard. From this we can measure the latency from message generation to display update and to the pit's dashboard, verifying that all the signals match their expected values from the DBC dataset. We will know that this test is successful if we are able to accurately display latency <50 ms, display telemetry in the pit's dashboard <250 ms, and have a high (99%) signal decoding accuracy.

For display stability, we will render the driver display under high-frequency CAN traffic to ensure smoothness and readability on outputs. From simulating peak bus loads using VCAN, we are able to observe the frame rate stability (shooting for 60 fps), and UI responsiveness. A successful display test will be determined under the following conditions: no dropped or frozen frames during the operation, stable rendering at the target refresh rate (60 fps), and no visual artifacts under load.

Data logging and retrieval is arguably the most important part of our entire system. To properly test the system's ability to log telemetry for post-run analysis the following must be done: record telemetry during a simulated run (VCAN), properly upload logs to the cloud pipeline, and verify data integrity by comparing logged values to source CAN data (ensuring all data was retrieved and no data corrupted).

The web-based configurator will be tested on its ability to provide dynamic modifications to the systems with no restarts. This test will be verified through modifying multiple display layouts and bindings through the web interface, measuring the propagation time to the Raspberry Pi to update the display. A successful run will look like: configurations applied in a short amount of time (<3 seconds from upload to

actual display on car), no system restart required for updating the display, and no disruption of other processes like telemetry or data logging.

Finally, the GPS will be validated as a part of the full system. In order to properly test the GPS, we will conduct the following tests to ensure accuracy on: GPS speed and position as compared to truth (position is accurately logged allowing ~1.5 meters of error), and synchronization with our existing CAN telemetry and data logging software.

Overall, our validation focuses on end-to-end system behavior, ensuring all subsystems work individually and together to produce verifiable and accurate outputs. Through quantitative metrics, our testing approach ensures that the system not only works but satisfies the accuracy requirements for the SAE team.

3.3 Updated division of labor and responsibilities

Provide a detailed list of deliverables (and due dates) for each team member.

Campbell Wright Deliverables

Deliverable	**Due Date**	**Status**
Research & Design	Feb. 23	COMPLETE
Full Product Specifications Check	Feb. 25	COMPLETE
Main PCB schematic	Mar. 9	COMPLETE
Daughter board schematic	Mar. 9	COMPLETE
PCB layout + wiring	Mar. 12	COMPLETE
Fabrication order submission	Mar. 15	COMPLETE
PCB assembly	Apr. 4	COMPLETE
Power bench test	Apr. 4	COMPLETE
CAN+GPS test	Apr. 7	COMPLETE
Display+daughter board functionality test	Apr. 7	COMPLETE
Full hardware integration with software	Apr. 14	COMPLETE
3D printed enclosure + final	Apr. 20	COMPLETE

wiring		

4 Implementation Notes

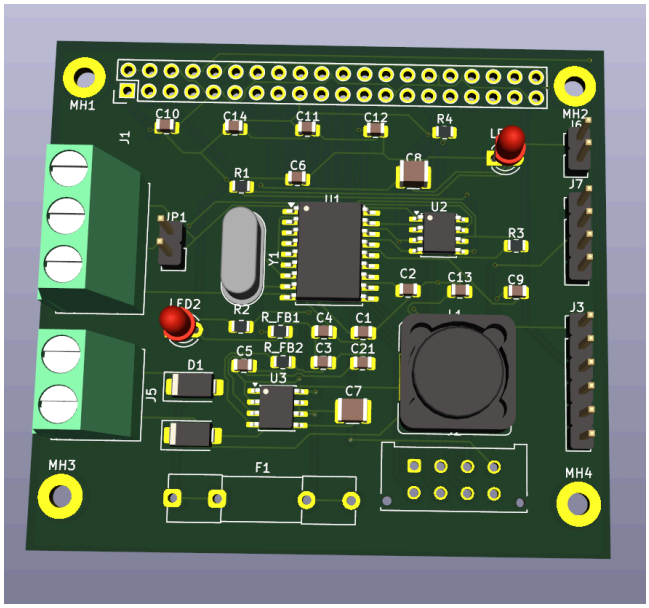
Completed items include the following:

- PCB Schematic + Layout
- Embedded Code
- Web App - Frontend
- Graphics engine
- Data Logger

PCB Schematic:

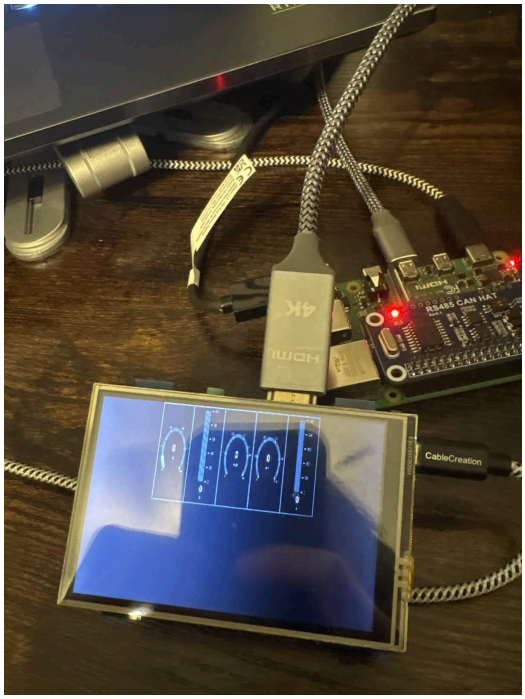
Daughter board with buttons and LEDs completed.

Main PCB completed.



Embedded Code:

Code reacting to virtual CAN signal inputs.



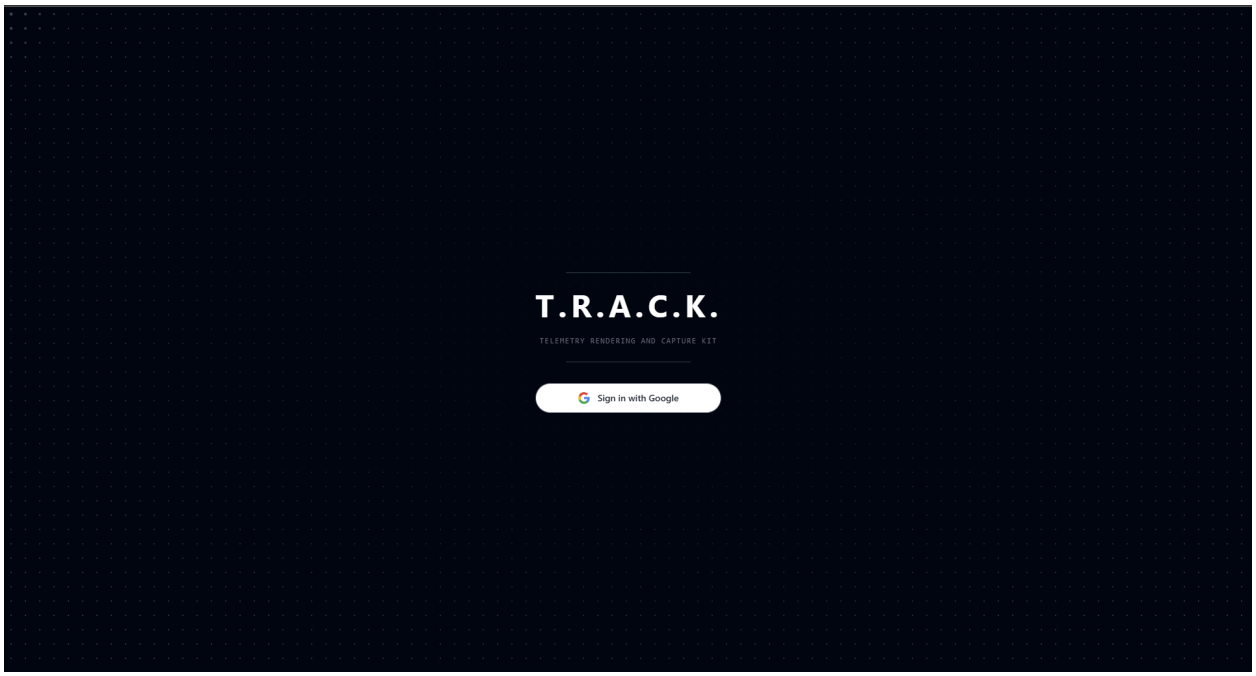
Another thing to note on the embedded side is that the data logger has been completed, although it still requires testing.

Web UI - Backend:

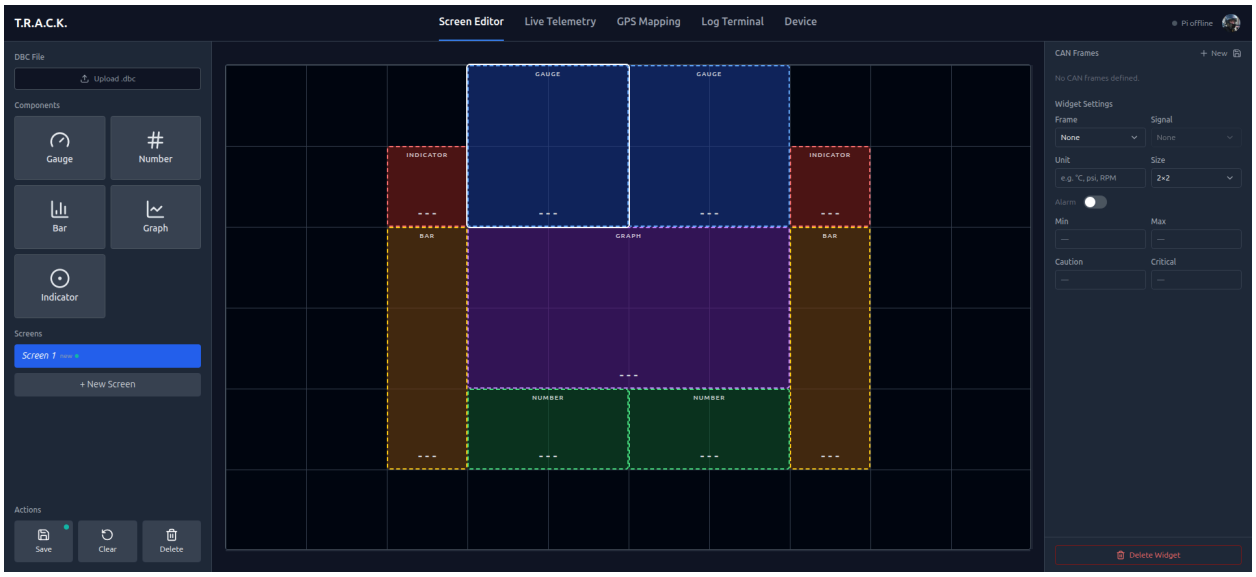
The backend is complete and deployed. Graphics screen configuration, DBC management, per-user screen preferences, log ingest and read, and device registration with team-member access are all live. Pi mapping is finalized via the /ws/pi WebSocket, authenticated by x-device-id and x-device-secret headers, with the Pi auto-linked to a user's account by matching the signed-in email against the device's team-member list. Every screen write rebuilds the full { screens } payload and pushes it in-band to the Pi as a config_update WSS message. Pi-initiated edits flow back up the same socket as graphics_upload or dbc_upload. All cross-tab sync, including screen updates, deletes, and preferences, is handled through the /ws/client broadcast channel. The backend runs as the track-web Fly app and auto-deploys on push to main via the Fly GitHub integration.

Web UI - Frontend:

Login Page

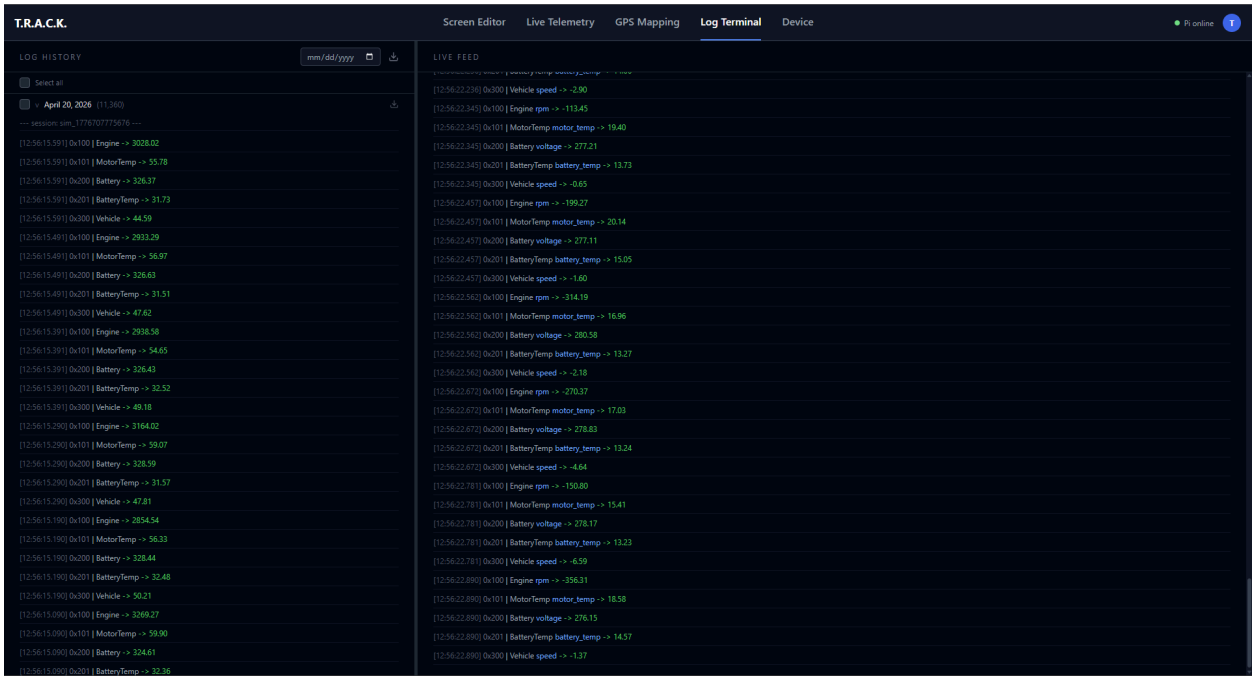


Screen Configurator

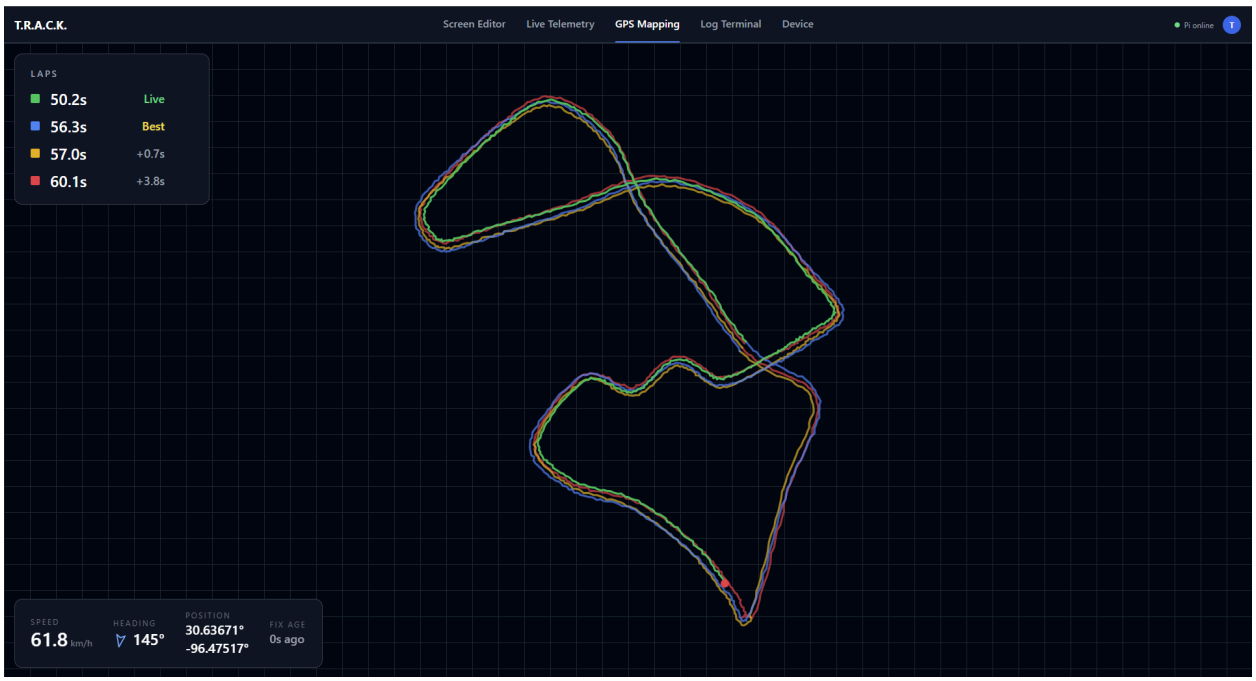


Telemetry Viewer

Log Terminal

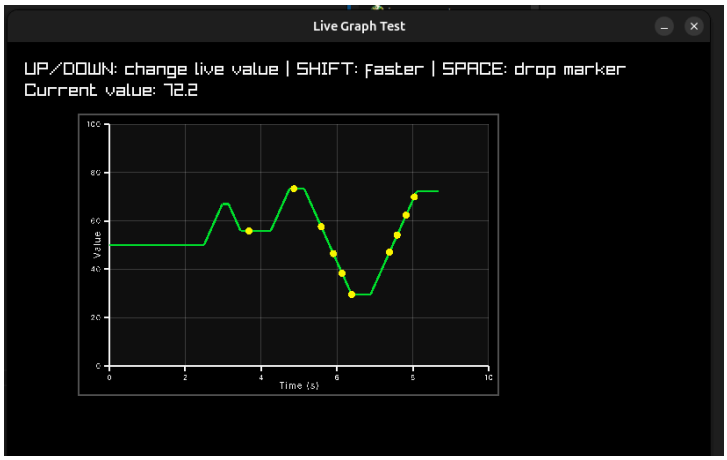


Gps Mapping



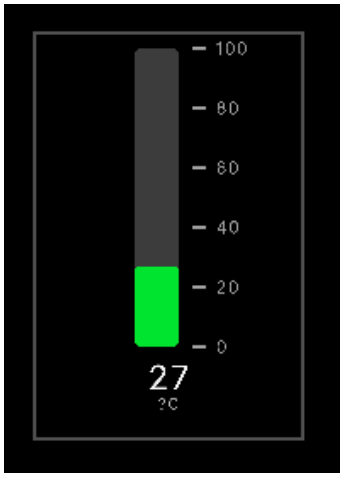
Graphics engine:

Graph widget

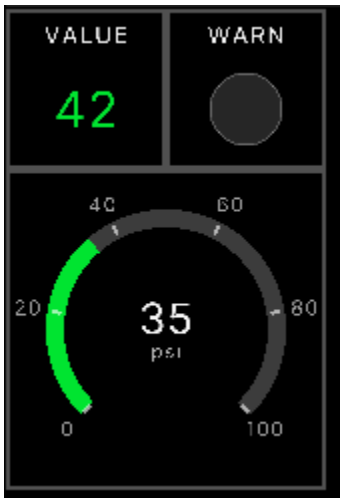


Number widget

Bar graph widget



Gauge, indicator, and number widgets



5 Experimental Results

Introduction

System validation was conducted using a two-Pi setup bench: the main pi running the full T.R.A.C.K. embedded stack and a second Pi generating synthetic CAN traffic through a can-utils script. A virtual CAN interface was used for latency testing so frames could be timestamped accurately at both ends. The web application was tested from a laptop and a phone simultaneously, and GPS functionality was verified outdoors. All results below reflect bench testing against simulated CAN traffic.

Requirement	Specification	Result	Status
Real-time CAN display	Signal on screen within 50 ms, 60 FPS	Passed with comfortable margin; display update consistently well under the limit on bench hardware	PASSED
Configurable layout	Non-developer reconfigures display in under 2 minutes from any browser	3 team members each completed the task in under 2 minutes on first attempt	PASSED
No proprietary tools	All configuration from a standard browser, any OS/device	Verified on Mac (Safari), Windows (Chrome), iPhone, and iPad	PASSED
Cloud connectivity	WebSocket heartbeat within 1 second; telemetry forwarded at 20 Hz	Connection established in under 1 second; telemetry forwarded at 20 Hz steady-state	PASSED
GPS tracking	Position and speed logged alongside CAN data	GPS data present in log files alongside CAN entries; live track visible in web app	PASSED

Continuous data logging	All CAN telemetry logged to disk for full session, compressed binary format	Extended simulated sessions logged without data loss; log integrity confirmed by playback	PASSED
Web-based display editor	5 widget types on a 10x6 grid; save/load via REST APIs	All five widget types functional in editor and rendered correctly on the Pi display	PASSED
Hot reconfiguration	Layout changes apply without restarting any process	New layouts pushed from browser and live on display within a few seconds; no process restarts observed	PASSED

CAN Display Latency and Frame Rate

The CAN-to-display path was validated by having the second Pi transmit frames on a known message ID at a sustained rate while the graphics engine was running. We verified visually and via timestamps that signal values updated on the screen well within the 50 ms budget. The 60 FPS target was confirmed by monitoring the raylib frame timer output over a sustained run, including under elevated bus traffic on the virtual CAN interface. The shared-memory architecture kept the graphics engine fully decoupled from CAN bus load, and the frame rate did not degrade when the simulated bus was saturated.

Configurable Display Layout

Three team members who had not seen the web application before were each given the same task: open the Screen Editor, add a gauge, a number display, a graph, and an indicator, bind each to a CAN signal from an uploaded DBC file, and save the screen. All three finished the task in under two minutes, and each gave feedback on the drag-and-drop grid as well as the signal binding with a pre-loaded DBC file.

No Proprietary Tools

The configuration workflow was completed on four different devices without installing any software. The devices included a Macbook Pro running Safari, a Windows laptop running Chrome, an iPhone running Safari, and an iPad running Chrome. Each device successfully opened the web app, modified a screen layout, and confirmed the update was visible on the Pi display. This directly satisfied the requirement that no vendor-specific tools or operating system are required for configuration.

Cloud Connectivity

The WebSocket connection between the Pi cloud bridge and the backend server was established in under one second on each test. Telemetry frames were forwarded to connected browser clients at the configured 20 Hz rate and were visible updating in real time on the Live Telemetry page. We also verified the reverse direction: saving a new screen layout in browser caused the Pi display to update within a few seconds, with no process restart and no interruption to ongoing logging.

GPS Tracking

The SparkFun NEO-M9N was tested outdoors and was able to acquire a fix within about a minute of startup. Position data appeared in the session log files timestamped alongside CAN entries, and the GPS

Mapping page in the web app rendered a live track outline as the module was driven around a parking lot. Lap detection triggered correctly each time the starting position was re-entered. The module's rated positional accuracy is approximately 1.5m, which is adequate for track mapping and lap delta purposes.

Data Logging

Extended simulated sessions were run with the virtual CAN interface generating continuous CAN traffic. The data-logger process wrote entries to disk without interruption for the full session duration. After the session, a Python script relayed the binary log and decoded each entry, validating that all entries matched the injected frame sequence with no corruption. Completed log files were uploaded to the backend over the WebSocket cloud bridge in chunked transfers and were retrievable from the Log Terminal page in the web application.

Display Editor and Hot Reconfiguration

All five widget types were placed on the 10x6 editor grid, saved via the REST API, and verified to render correctly on the Pi display. Screen layouts were also loaded back from the API and restored without loss. Hot reconfiguration was tested by saving a new layout from the browser while the system was fully running. All six embedded processes stayed alive, telemetry continued logging without a gap, and the new layout appeared on the Pi display within a second of the browser save click.

6 Users manual

6.1 Introduction

This manual covers everything a team member needs to install, configure, and operate the T.R.A.C.K. (Telemetry Rendering And Capture Kit) system. T.R.A.C.K. is a race-day telemetry and display platform built for the Texas A&M Formula SAE Electric vehicle. It shows real-time vehicle data to the driver on a 7-inch dashboard display, records that data to an SD card for post-session analysis, and streams live telemetry to pit crew members over the internet.

Intended Users

This manual is written for the following users. No software engineering background is required to operate the system, but experience in this realm will help:

- Pit engineers who need to view live telemetry and adjust the display layout during a competition day.
- Drivers who need to understand what is shown on the display and how to navigate screens.
- Team members responsible for installing or re-deploying T.R.A.C.K. between seasons.
- New team members taking over the system for the first time.

System Overview

T.R.A.C.K. consists of two parts that work together:

- The on-car unit: a Raspberry Pi 4 running on the vehicle, connected to the CAN bus and a 7-inch HDMI display.

- The web app: a browser-based tool hosted in the cloud that lets pit crew view live telemetry, configure the driver display, and download session logs.

The on-car unit reads live data from the vehicle (speed, motor current, battery voltage, etc.) and displays it on the dashboard. At the same time, it forwards that data to the cloud so the pit crew can see it on their phones or laptops. If the Pi loses internet connectivity mid-run, it will continue displaying data and logging locally, and will upload the session data when the connection is restored.

6.2 Hardware Installation

What You Need

Before installing T.R.A.C.K. on the vehicle, gather the following items:

Item	Details
Raspberry Pi 4B	Pre-flashed with Raspberry Pi OS (64-bit)
T.R.A.C.K. Main PCB	Hat-form-factor board that mounts on the Pi GPIO pins
LED/Button Daughter Board	Connects to Main PCB via an 8-pin Molex Micro-Fit 3.0 connector
HDMI Display	800x480 resolution; HDMI and USB connections to the Pi
SparkFun NEO-M9N GPS Breakout	Connects via 4-pin header (GND, 3V3, TX, RX)
MicroSD Card (32GB+)	Stores the OS, software, and session log files
Vehicle 12V power (via PCB)	PCB regulates down to 5V for the Pi and peripherals
3D-printed enclosure	Houses all electronics
CAN bus wiring harness	Vehicle-side connector to the PCB CAN screw terminals

PCB Assembly and Mounting

The T.R.A.C.K. Main PCB plugs directly onto the Raspberry Pi 4 GPIO header. All 40 GPIO pins align with the Pi hat form factor.

- Ensure the Pi is powered OFF before installing any boards.
- Align the 40-pin female header on the underside of the Main PCB with the 40-pin GPIO header on the Pi. Press down firmly and evenly until fully seated.
- Connect the Daughter Board cable (8-pin Molex Micro-Fit 3.0) to the matching connector on the Main PCB. The connector is keyed and cannot be inserted backwards.
- Mount the assembled Pi + PCB stack inside the 3D-printed enclosure. Secure with M2.5 screws where provided.

Display Connection

- Connect the display HDMI cable to the Pi's micro-HDMI port (the port closest to the USB-C power port).
- Connect the display USB touch cable to any of the Pi's USB-A ports.
- Connect the 5V display power cable to the two-pin header on the Main PCB labeled DISP_PWR (GND and 5V).

  NOTE: *The display draws power from the PCB, not from the HDMI cable. Both the HDMI data cable and the 2-pin power cable must be connected before powering on.*

GPS Module Connection

The SparkFun NEO-M9N GPS module connects to the Main PCB via a 4-pin female header. Only four of the six available pins are used:

GPS Pin	PCB Label	Function
GND	GND	Ground reference
3V3	3V3	3.3V power supply from PCB
TX (GPS transmits)	GPS_RX (Pi receives)	NMEA sentence output from GPS module
RX (GPS receives)	GPS_TX (Pi transmits)	UBX configuration commands to GPS

NOTE: *The SDA and SCL pins on the GPS module are not connected. Do not use them. Mount the GPS module with an unobstructed view of the sky for best satellite lock.*

CAN Bus Wiring

- Locate the two-terminal CAN connector on the Main PCB (labeled CANH and CANL).
- Connect CANH to the vehicle's CAN high wire (the dominant wire in the twisted pair).
- Connect CANL to the vehicle's CAN low wire.
- Verify that 120-ohm termination resistors are present at both ends of the CAN bus. T.R.A.C.K. does not add its own termination; the vehicle harness must be properly terminated.

Power Input

The Main PCB accepts 12V from the vehicle supply and uses the onboard MP1584EN buck converter to regulate it down to 5V for the Pi and peripherals.

- Connect the vehicle 12V supply to the PCB's input terminals. Observe polarity; the board includes reverse-polarity protection (Schottky diode) and a 3A fuse.
- Verify the 3A fuse is installed before first power-on.
- Do not bypass the fuse or connect a supply rated above 15V.

NOTE: *If the Pi does not power on, check the fuse first; it is the most common fault during initial installation.*

/boot/config.txt Configuration

The following lines must be present in /boot/config.txt on the Pi's SD card before first boot. Edit this file from another computer or via SSH after initial boot:

```
dtoverlay=vc4-kms-v3d
gpu_mem=128
dtoverlay=mcp2515-can0,oscillator=8000000,interrupt=25
```

The first two lines enable the DRM graphics driver and allocate GPU memory. The third line enables the MCP2515 CAN controller using the PCB's 8 MHz crystal and GPIO25 interrupt.

6.3 Software Installation

Overview

T.R.A.C.K. runs as a set of system services on the Raspberry Pi. Installation is handled by two scripts: setup-pi.sh (run once to install dependencies and build the code) and deploy.sh (run after any code update to install binaries and enable services).

Prerequisites

- Raspberry Pi OS (64-bit, Debian Bookworm) installed on the SD card.
- Internet access on the Pi (for downloading packages and building dependencies).
- Node.js 18 or newer installed on the Pi (for building the web UI).
- Git installed on the Pi.

To install Node.js on the Pi if not already present:

```
curl -fsSL https://deb.nodesource.com/setup_18.x | sudo -E bash -
sudo apt install -y nodejs
```

Cloning the Repositories

T.R.A.C.K. uses two repositories that must be cloned side by side into the same parent directory:

```
mkdir -p ~/track && cd ~/track
git clone https://github.com/evilspoon13/track-embedded.git fsae-display
git clone https://github.com/evilspoon13/track-web.git track-web
```

Running the Pi Setup Script

The setup script installs all system dependencies, builds the raylib graphics library from source (configured for the Pi DRM display driver), builds the ixwebsocket library, creates the Python virtual environment for the captive portal, and compiles all T.R.A.C.K. C++ binaries.

```
cd ~/track/fsae-display
```

```
./scripts/setup-pi.sh
```

This script will take 15–30 minutes on a fresh Pi because it compiles raylib and ixwebsocket from source. Do not interrupt it. When complete it will print next steps.

> **NOTE:** *If the script fails partway through, it is safe to re-run. Already-installed libraries are detected and skipped on subsequent runs.*

Building the Web UI for the Pi

The React web UI must be compiled specifically for the Pi (disabling Firebase authentication and using local config files). Run:

```
cd ~/track/fsae-display
./scripts/build-ui.sh
```

This produces a static build in captive-portal/ui/ that the Flask captive portal serves when users connect to the Pi's Wi-Fi access point.

Deploying the Software

The deploy script copies all binaries and configuration files to /opt/track/ and registers all services with systemd so they start automatically on every boot:

```
cd ~/track/fsae-display
sudo ./scripts/deploy.sh
sudo reboot
```

On reboot, all T.R.A.C.K. services start automatically in the correct order. The display should show data within 10–15 seconds of boot.

Systemd Services

Service	What It Does
track-setup	Creates required directories in /tmp at boot
track-can-interface	Sets the CAN bus bitrate (default 500,000 bps)
track-can-reader	Reads CAN frames and decodes signals into shared memory
track-gps-reader	Reads GPS NMEA data into shared memory
track-gpio	Reads physical buttons on the daughter board
track-graphics	Renders dashboard widgets on the HDMI display at 60 FPS
track-logger	Writes telemetry to binary log files on the SD card

Service	What It Does
track-cloud-bridge	Streams telemetry to cloud; receives config updates
track-portal	Runs the Flask Wi-Fi setup portal on port 80

To check service status after reboot:

```
sudo systemctl status track-can-reader track-graphics track-logger
track-cloud-bridge
```

CAN Interface Configuration

CAN settings live in /opt/track/config/track.env. The default uses the hardware CAN interface at 500 kbps:

```
CAN_IFACE=can0   CAN_MODE=can   CAN_BITRATE=500000
```

For testing without a vehicle, switch to the virtual CAN interface (vcan0) by editing track.env and restarting services:

```
sudo systemctl restart track-can-interface track-can-reader track-graphics
track-logger track-cloud-bridge
```

Updating the Software

```
cd ~/track/fsae-display && git pull

make clean && make PLATFORM=DRM        # if embedded code changed

./scripts/build-ui.sh                   # if frontend code changed

sudo ./scripts/deploy.sh

sudo systemctl restart track-can-reader track-graphics track-logger
track-cloud-bridge track-portal
```

6.4 Operation

Powering On

T.R.A.C.K. starts automatically when the vehicle 12V supply is connected. No manual interaction is required. The display shows the dashboard within approximately 45–60 seconds of power-on. If the display remains blank after 90 seconds, see Section 6.5 (Troubleshooting).

Navigating Screens on the Driver Display

The driver can cycle through configured screen layouts using the physical buttons on the daughter board:

Button	Action
Button 1 (left)	Cycle to the next screen layout
Button 2 (middle)	Reserved for future use

Button	Action
Button 3 (right)	Reserved for future use

NOTE: *The left/right arrow keys on a USB keyboard connected to the Pi also cycle screens, which is useful during testing.*

Using the Web App (Pit Crew)

The web app runs in any modern browser on a phone, tablet, or laptop. No local installation is needed.

- Open the web app URL in your browser (https://track-web-frontend.fly.dev/).
- Click Sign in with Google and use your Google account.
- On first sign-in, if you see an Unlinked status, ask your team leader to add your Google email to the Pi's team-member list via the Device page.
- Once linked, your device's live data appears automatically.

Screen Editor

The Screen Editor lets you build the layout shown on the driver display. Changes are pushed to the Pi in real time over the internet, no restart required.

- Navigate to the Screen Editor page.
- Drag widgets from the left palette onto the 10-by-6 grid canvas.
- Click a placed widget to open its configuration panel and set the CAN signal, display range, alarm thresholds, and units.
- Click Save. The configuration propagates to the Pi in under 3 seconds.

Available widget types:

Widget	Best Used For
Gauge	Single analog value with a sweeping arc (e.g., motor speed)
Bar	Single value as a vertical fill bar (e.g., state of charge)
Horizontal Bar	Same as Bar but oriented horizontally
Number	Large numeric readout (e.g., lap time)
Indicator Light	On/off status light (e.g., fault flags)
Graph	Time-series or XY plot (e.g., speed vs. time)

Viewing Live Telemetry

Navigate to the Live Telemetry page. Each CAN signal the Pi is receiving appears as a card showing the signal name, current value, and a sparkline of recent history. The grid updates in real time. If the Pi is offline, cards will indicate a stale connection.

GPS Mapping

Navigate to the GPS Mapping page. The map traces the vehicle's live position updated in real time. A naive lap detection algorithm marks each lap as the vehicle passes close to its starting position. The map supports scroll-to-zoom and click-drag panning.

Log Terminal

- The left panel shows a history of sessions grouped by day. Click a session to load it.
- The right panel shows the raw telemetry log for the selected session.
- Click Export CSV to download session data as a spreadsheet-compatible file.
- A Live Feed at the bottom shows incoming telemetry in real time during an active session.

First-Time Wi-Fi Setup (Captive Portal)

When the Pi boots without a configured Wi-Fi connection, it creates its own access point so you can configure it:

- On your phone or laptop, connect to the Wi-Fi network named TRACK (or the name configured in config/hostapd.conf).
- Your device may show a "sign in to network" prompt: tap it, or open any browser and navigate to any URL.
- Enter the venue Wi-Fi credentials (SSID and password).
- Submit the form. The Pi will connect to the internet and the TRACK access point will disappear.

Testing Without a Vehicle (VCAN)

For bench testing without vehicle hardware, use the virtual CAN interface to simulate CAN traffic:

```
./scripts/setup-vcan.sh
```

Then update /opt/track/config/track.env to use vcan0 and restart services. Use ./scripts/test-can.sh or cangen vcan0 to inject sample frames.

6.5 Troubleshooting

Symptom	Likely Cause	Fix
Display blank after boot	HDMI cable in wrong port, or gpu_mem not set	Verify HDMI is in the micro-HDMI port closest to USB-C. Check gpu_mem=128 in /boot/config.txt.
No data on display	CAN reader not running, or wrong bitrate	Run: sudo systemctl status track-can-reader. Verify CANH/CANL are wired and bitrate matches 500000 in track.env.
All values show 0 or frozen	CAN bus not active, or wrong DBC file	Run: candump can0. Verify display.dbc matches the vehicle signal definitions.
Pi not visible in web app	Cloud bridge not connected, or no internet	Run: sudo systemctl status track-cloud-bridge. Check Pi internet connectivity.
GPS page shows no track	No satellite fix yet	Give the GPS module a clear sky view and wait 1-2 minutes for a first fix.
Web app shows Unlinked	Your email not in the device team-member list	Ask the team leader to add your Google email on the Device page.
Display flickers or drops frames	CPU overload or thermal throttling	Check temperature: vcgencmd measure_temp. Ensure adequate airflow inside the enclosure. Target is 60 FPS.
Config changes don’t appear on display	Cloud bridge not connected	Check: sudo systemctl status track-cloud-bridge track-graphics.
Pi does not power on	Blown fuse	Check and replace the 3A fuse on the Main PCB input path.

6.6 Quick Reference Card

On-Car

Task	Action
Power on	Connect 12V to PCB; Pi boots automatically
Cycle display screens	Press Button 1 (left button on daughter board)
Check service status (SSH)	sudo systemctl status track-graphics track-can-reader

Task	Action
Restart all services (SSH)	sudo systemctl restart track-can-reader track-graphics track-logger track-cloud-bridge
Switch to VCAN for testing	Edit /opt/track/config/track.env then restart services

Web App

Task	Where
View live telemetry	Live Telemetry page
Change driver display layout	Screen Editor: drag widgets, click Save
Add a team member	Device page: Team Members section
View GPS track	GPS Mapping page
Download session log as CSV	Log Terminal: select session, click Export CSV

7 Course Debriefing

7.1 What Went Well

The most important decision we made as a team was agreeing on the overall system architecture at the beginning of our time together. This made the implementation schedule move more quickly and smoothly than expected and allowed us to develop the code and hardware to work under different circumstances and roadblocks. The ability to develop parallel subsystems, each using their own defined interfaces, allowed us to stay ahead of schedule all the way until our CDR and also allowed us to add two bonus features from our stretch goals, which included the GPS integration and a complete migration to Firebase.

Coordination between the team was handled with a weekly Monday meeting on Discord with fixed agenda items that included what work we had done so far, what issues had arisen, and what we will complete in the coming week. The meeting structure was lightweight enough to give us flexibility on assigning tasks and overall communication, but also consistent enough that nothing fell through the cracks for more than a week. Giving each member ownership over their own subsystem, and trusting them to move forward independently, worked effectively for the team as everyone had the technical depth to do so. Having hardware and software experience on the team also helped as well, since questions regarding the PCB and embedded code could be resolved in a conversation rather than across a handoff.

7.2 What Could Have Gone Better

The main PCB arrived much later than expected due to fabrication and shipping lead times, which compressed the hardware integration window near the end of the semester. Running the software stack against the off-the-shelf CAN HAT while waiting for the main PCB was effective, but it meant the custom board was not tested with the full embedded pipeline until late in the project. A future note on this would

be to allow four weeks of additional headroom for PCB manufacturing and fabrication and treat the order date as a hard deadline rather than a target.

As a result of this, the first full end-to-end integration test involving a CAN message traveling through the bus to the cloud via the embedded stack did not happen until closer to the end of the semester. Reflecting on this, a simpler or slimmer version of this test could have been performed much earlier in the semester to expose any issues that would need to be addressed and give enough time to address them. The team also could not perform any tests in the actual FSAE vehicle due to conflicting schedules between both teams. All testing was conducted against synthetic data from a second Pi to validate software correctness, but was not able to detect the noise and grounding issues that only appear in a real vehicle environment.

7.3 Lessons Learned & Course Reflections

The milestone structure of the course instilled an element of discipline that included writing what the system needs to do and how success will be measured prior to the beginning of implementation. Since all the requirements for the project had been specified from the first few weeks, the execution of this project was straightforward to produce and validate. This course structure made it imperative to define requirements before building rather than trying to build before even constructing a success criteria.

Collaborating with a real external stakeholder changed how the project developed. The team's input on the initial prototyping phase helped shape the choices of many internal and external parts of the project, including the widget categories, the use of the DBC configuration approach, and the decision to focus on browser-based configuration rather than device-based. Taking this feedback into account led to a product that served the needs of the group better than the original proposal would have on its own. The requirement for a true needs statement, followed by a full analysis and traceability to a specification, gave the team a shared understanding to make them more productive.

8 Budgets

8.1 Budget Narrative

The T.R.A.C.K. prototype budget covers all physical hardware required to build and demonstrate the system. Software infrastructure, including the Firebase backend, Fly.io hosting for the Express server, and all development tooling, runs entirely on free tiers and incurs no cost at the usage levels of this project. The original proposal estimated a total hardware budget of $239. During development, two scope additions affected this figure: a SparkFun NEO-M9N GPS breakout module was added for independent position and speed acquisition, and two CAN hats were procured for development and testing purposes. The final prototype cost came to $590.17, driven primarily by the custom PCB fabrication order through JLCPCB ($259.00) and the display unit ($85.00). The Raspberry Pi 4B ($75.00) and SparkFun GPS module ($74.95) are the next largest individual line items.

The budget overrun relative to the original $239 estimate is attributable to three factors: (1) the GPS module was added as a scope extension after the original proposal, (2) two CAN hats were purchased rather than one to support parallel development on separate machines, and (3) the custom PCB fabrication cost was higher than the initial estimate once final board dimensions and layer count were determined. All purchases were approved through the standard TAMU CSCE 483 purchase order process.

8.2 Detailed Cost Breakdown

Component	Qty	Unit Cost ($)	Total ($)	Source
Raspberry Pi 4B	1	75.00	75.00	Approved vendor
SparkFun NEO-M9N GPS Breakout	1	74.95	74.95	SparkFun Electronics
4.3-inch HDMI Display (Digiwise)	1	85.00	85.00	Amazon
CAN Hat (MCP2515-based)	2	16.31	32.62	Amazon
Main PCB (fabricated via JLCPCB)	5	51.80	259.00	JLCPCB
LED and Button Daughterboard (JLCPCB)	5	9.92	49.61	JLCPCB
3D Printing Filament (PLA)	1	13.99	13.99	Amazon
Total			**590.17**	

Note: The Main PCB cost of $259.00 includes both the primary hat-form-factor board and the LED/button daughterboard in a single JLCPCB order, though they are listed separately above as reported in the CDR. The daughterboard cost of $49.61 is the individually itemized JLCPCB line.

8.3 Comparison to Proposed Budget

Item	Proposed ($)	Actual ($)	Variance ($)	Notes
Raspberry Pi 4B	75.00	75.00	0.00	On budget
Display	85.00	85.00	0.00	On budget
CAN Hat	16.31	32.62	+16.31	2 units purchased vs 1 proposed
Main PCB	~50.00	259.00	+209.00	Full custom fabrication; estimate was low

Item	Proposed ($)	Actual ($)	Variance ($)	Notes
LED/Button Daughterboard	Not in proposal	49.61	+49.61	Added during design phase
GPS Module	Not in proposal	74.95	+74.95	Scope addition post-proposal
3D Printing Filament	13.99	13.99	0.00	On budget
Firebase / Fly.io hosting	0.00	0.00	0.00	Free tier; no cost incurred
Total	**~$239.00**	**$590.17**	**+$351.17**	

8.4 Mass Production Cost Estimate

The current prototype cost reflects one-off pricing for custom PCB fabrication and individual component purchases. At volume, the per-unit cost would decrease substantially. The table below estimates cost at a hypothetical production run of 50 units, representative of a small motorsport equipment supplier serving multiple FSAE teams.

Component	Prototype Unit Cost ($)	Est. Volume Unit Cost ($)	Notes
Raspberry Pi 4B (or CM4)	75.00	55.00–65.00	Modest discount at volume; CM4 is cheaper for embedded builds
Display	85.00	60.00–70.00	OEM pricing available at volume from display manufacturers
CAN Transceiver (MCP2515 + SN65HVD230)	16.31	8.00–12.00	Standard ICs; significant discount at 50+ units
Custom PCB (Main + Daughterboard)	308.61	40.00–60.00	PCB cost drops sharply with volume; BOM assembly included
GPS Module (NEO-M9N)	74.95	25.00–35.00	u-blox modules cheaper direct from distributor at volume

Component	Prototype Unit Cost ($)	Est. Volume Unit Cost ($)	Notes
Enclosure (injection mold vs 3D print)	13.99	15.00–25.00	3D print replaced by injection mold; tooling amortized over run
Assembly labor	0.00	20.00–40.00	Not applicable to prototype; significant at production scale
Estimated total per unit	**$590.17**	**$223–$307**	**~50% reduction vs prototype**

At volume, the dominant cost shifts from PCB fabrication to assembly labor and the Raspberry Pi itself. A further cost reduction path exists by replacing the Pi 4B with a custom carrier board using the Raspberry Pi Compute Module 4, which retails around $35 in quantity and eliminates several redundant connectors. The web infrastructure (Firebase, Fly.io) remains free at the usage levels of a small team sport deployment and would only incur cost at very large scale.

9 References

[1] PLEX Tuning, “PLEX SDM / Driver Display System,” *PLEX Tuning*, 2024. [Online]. Available: https://www.plextuning.com. Accessed: Feb. 2026.