MAPACHE – AN INTELLIGENT, REAL-TIME TELEMETRY PLATFORM FOR FORMULA SAE

Bharat Kathi, Andrey Otvagin, Jacob Jurek,
Austin Chan, Colin Bickel
Gaucho Racing
University of California, Santa Barbara

Faculty Advisor:
Kirk Fields
University of California, Santa Barbara

I ABSTRACT

In motorsport, real-time data is key to performance and reliability, but many Formula SAE teams forgo telemetry due to the complexity of traditional systems. We present Mapache, a real-time telemetry and data analytics platform developed by Gaucho Racing, the Society of Automotive Engineers (SAE) chapter at the University of California, Santa Barbara. Mapache eliminates the need for base stations or dedicated personnel, making high-performance telemetry more accessible and resilient. The on-vehicle system decodes CAN messages, streams telemetry via MQTT when connectivity is available, and caches data locally when offline. Upon reconnection, a GPU-accelerated pipeline ensures zero-loss migration of cached data to the cloud. MQTT messages are ingested into a SingleStore database, powering real-time, low-latency analytics. As the first system of its kind in Formula SAE, Mapache sets a new benchmark for intelligent telemetry.

II INTRODUCTION

Formula SAE (FSAE) is a collegiate engineering competition where teams design, build, and race small, high-performance electric vehicles. Success in FSAE is not just about mechanical design and driving skill. Particularly with the onset of new rules and design criteria, data-driven decision-making is a crucial factor in optimizing vehicle performance, diagnosing issues, improving race strategy, and overall competition standing. With race cars equipped with hundreds of sensors, teams must collect, process, and analyze vast amounts of telemetry data in real time to extract actionable insights.

Our goal with Mapache was to develop a fully self-sufficient telemetry system capable of transmitting real-time vehicle data from anywhere, without the need for a track-side base station. By leveraging LTE connectivity, cloud-based data processing, and visualization dashboards, we created a solution that provides continuous real-time data access, enhances debugging efficiency, and streamlines the overall vehicle development and testing process.

Figure 1: Gaucho Racing's 2025 (left) and 2024 (right) vehicles

III SYSTEM ARCHITECTURE

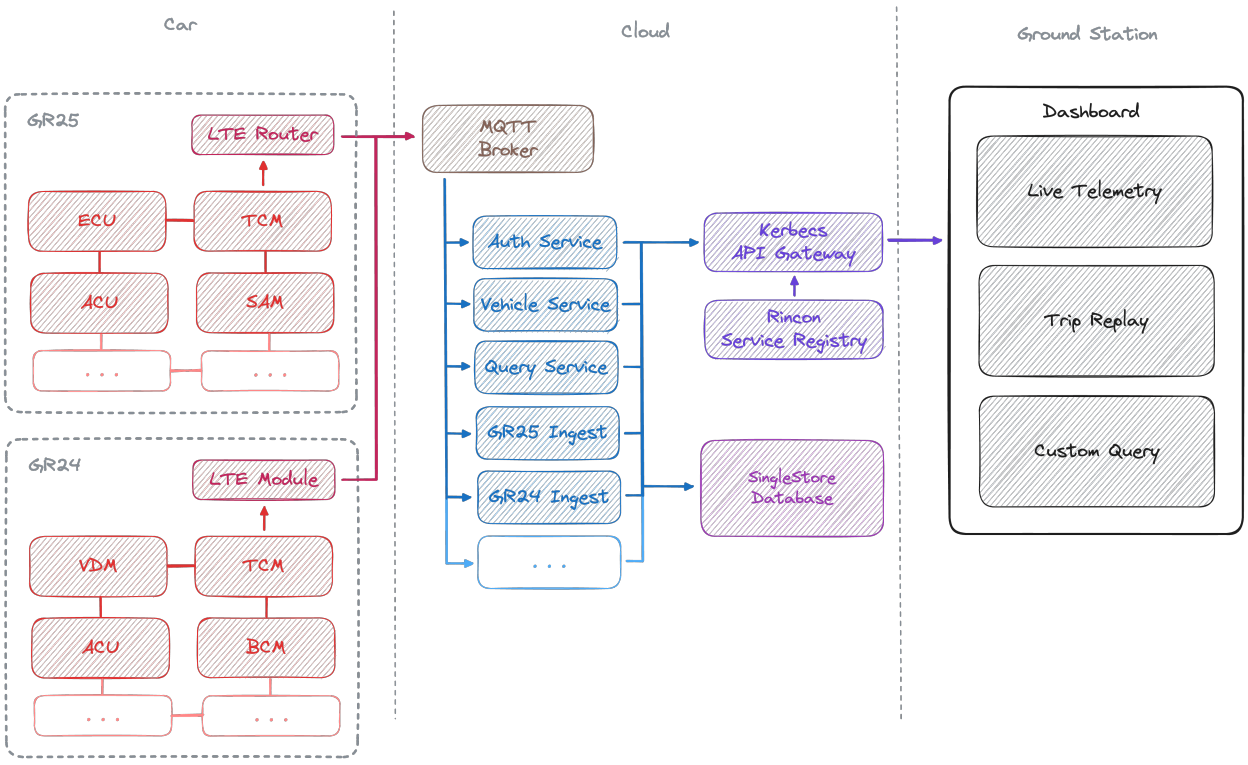

Figure 2: Mapache system architecture diagram

A. *Vehicle Architecture*

Gaucho Racing's 2025 competition vehicle (GR25) was built with a distributed electronic control system based on STM32G4 nodes. Each node was responsible for a specific subsystem of the

car. These nodes communicate over a Controller Area Network (CAN) bus, which is an industry standard protocol initially developed to reduce the wiring complexity and cost associated with communication between nodes in automobiles, but has since been adopted into everything from robotics to agricultural equipment [1].

Each node in GR25 has a unique identifier, embedded within a custom CAN ID format indicating the sender (GR ID), intended receiver (Target ID), and message type (Message ID). For example, if the system wants to limit maximum throttle when battery charge falls below 10%, it listens for specific CAN messages indicating battery status (e.g., Message ID 200), filtering messages by sender node to ensure accuracy. This approach resembles event-based systems such as Apache Kafka.

GR25 has the following nodes:
ACU: Accumulator Control Unit, responsible for interfacing with the BMS boards and controlling the HV hardware of the car
BMS: Battery Management System, located on every cell segment, used to monitor cell voltage and temperature
Dash Panel: Used to display information to the driver and send startup button data
Inverter: Motor Controller, used to supply current and torque commands to the motor
ECU: Electronic Control Unit, hosts the main vehicle state machine that is responsible for driving the car
SAM: Sensor Array Module, module used to communicate with non-critical sensors and transmit their information for the sake of data acquisition
Steering Wheel: controls the buttons and knobs on the physical steering wheel, used for configuring driving modes
TCM: Telemetry Compute Module, responsible for processing and transmitting vehicle data to the cloud in real-time

B. Mapache Architecture

Mapache is built on a microservice architecture, where each service is responsible for a specific part of the data pipeline. This design allows each component to be developed, deployed, and scaled independently, improving maintainability, fault isolation, and overall system performance. To coordinate ingress and inter-service communication, we use Kerbecs, an open-source API gateway that proxies requests to the appropriate service, and Rincon, an open-source service registry that manages service discovery.

All services in the Mapache stack are fully containerized, enabling seamless deployment across a variety of environments, from local development and on-premise servers to cloud-native platforms. While the architecture is cloud-agnostic, we deploy our production environment on Amazon EKS (Elastic Kubernetes Service), which provides horizontal scaling during periods of high traffic or CPU-intensive workloads, such as large data queries. This ensures the system remains responsive and stable even under demanding operational conditions.

The six key services are:
SingleStore: a high-performance distributed SQL database for all telemetry data
NanoMQ: a lightweight MQTT broker for real-time message streaming
Vehicle Service: which tracks metadata for vehicles, trips, and laps
Ingest Service: which decodes MQTT CAN data and writes structured signals to the database
Query Service: which reshapes and filters data for analysis
Dashboard: a web application for real-time and historical visualization.

IV TELEMETRY COMPUTE MODULE

The Telemetry Compute Module (TCM) is a node on GR25 that serves as the bridge between the vehicle's onboard data busses and the cloud, enabling reliable real-time telemetry. GR25 has two CAN buses: a CAN 2.0B bus for primary communication between critical nodes, and a CAN FD bus for high-speed sensor data collection. CAN FD supports up to 64 bytes of data, a significant increase from the 8 bytes normally supported by CAN 2.0. Consequently, CAN FD allows for a higher data rate (up to 8 Mbps) over CAN 2.0 (1 Mbps).

A. Hardware

The TCM hardware stack consists of a Jetson Orin Nano, an Alcatel Link Hub 4G LTE router, and a custom TCM-CAN PCB featuring an STM32G474 microcontroller, each playing a critical role in the system. The Jetson runs the entire software stack, including data acquisition, processing, and synchronization services. The LTE router ensures a stable internet connection in remote environments, allowing the Jetson to upload data when connectivity is available. Meanwhile, the TCM-CAN PCB acts as a dedicated bridge between the Jetson and the car's dual CAN FD networks. The STM32G474 microcontroller was chosen for its dual CAN FD controller support, as well as to maintain hardware consistency across the rest of the vehicle's embedded systems.

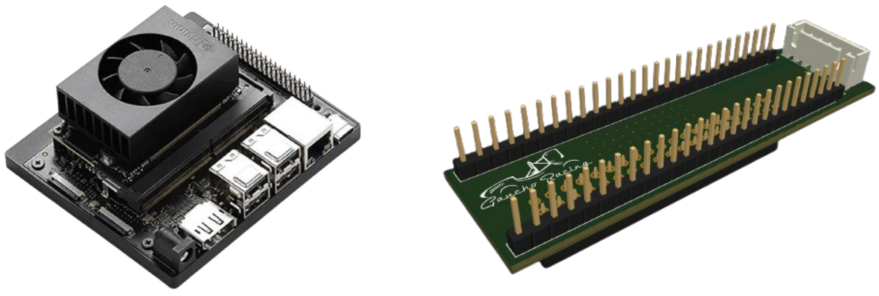

Figure 3: Nvidia Jetson Orin Nano (left) and TCM-CAN PCB (right)

To transfer CAN data from the STM32 to the Jetson efficiently, we chose SPI (Serial Peripheral Interface) as the communication protocol. SPI offers significantly higher theoretical throughput compared to UART or I2C, which is essential for keeping up with the high-frequency, dual-bus CAN traffic.

B. *Software*

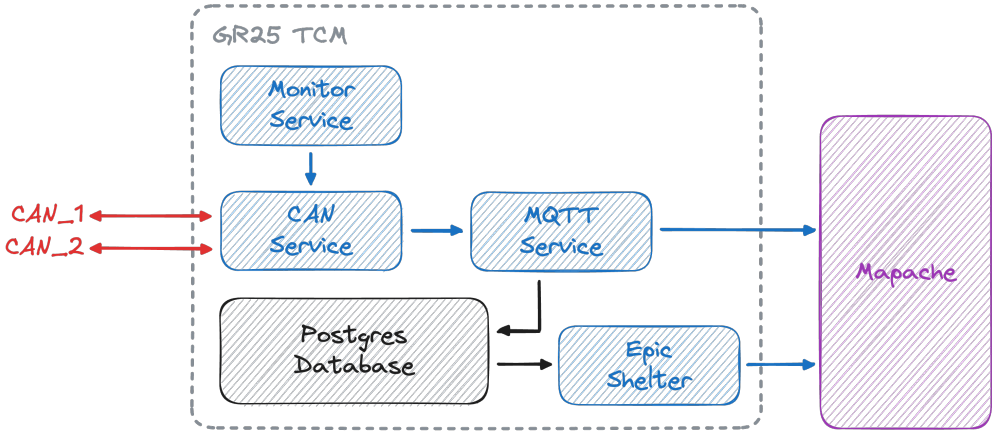

Figure 4: TCM software architecture diagram

The Jetson Orin Nano runs Jetson Linux, a specialized variant of Ubuntu tailored with NVIDIA drivers for GPU acceleration and hardware integration. Running a full-fledged operating system on the TCM allows us to orchestrate multiple independent services, enabling a flexible and modular architecture. These services are containerized and automatically launched on device startup and are designed to be independently maintainable and scalable. There are a total of 5 services running on the TCM.

PostgreSQL Database: A local PostgreSQL instance serves as the primary data store for all TCM services. It logs metadata such as connectivity status and latency metrics, but more critically, it stores a complete backup of all incoming CAN traffic. PostgreSQL was selected for its robustness under high write throughput, offering strong consistency and performance guarantees, even with rapid-fire message ingestion.

CAN Service: The CAN service, written in C, acts as the primary interface between the TCM and the vehicle's CAN bus, communicating with the custom TCM-CAN PCB. It handles all inbound and outbound frames, serving as the central bridge between the software system and other vehicle nodes. Communication occurs over SPI, with a dedicated GPIO interrupt pin used to signal when the microcontroller has one or more messages ready. This interrupt-driven design eliminates polling, reducing CPU load and ensuring timely message transfer. Upon startup, the service also launches a UDP server, broadcasting all received CAN messages to connected clients in binary format and relaying valid incoming UDP messages back onto the CAN bus. By decoupling low-level CAN handling into a standalone service and exposing it via UDP, the architecture enables any external service to access CAN messages without re-implementing any logic.

Monitor Service: The Monitor service, written in Go, tracks internal TCM metrics such as CPU/GPU load, memory usage, disk space, and temperatures using Jetson's `tegrastats` utility. These readings are packaged into CAN-compatible messages and sent to the vehicle's dash panel

for real-time monitoring. It also performs a heartbeat ping to Mapache every 10 seconds, recording connectivity status in the local database. If a ping fails or no response is received, the system flags the TCM as offline, enabling other services to adapt based on network availability.

MQTT Service: The MQTT service, written in Go, transmits CAN data from the TCM to the cloud by bridging the local UDP stream (from the CAN service) to a remote MQTT broker using the lightweight publish/subscribe MQTT protocol. It listens for packed CAN structs and republishes them under a structured topic format: `gr25/<vehicle-id>/<node-name>/<message-id>`, where the vehicle ID is hardcoded, the node name is derived from a lookup map, and the message ID is preserved in hexadecimal for traceability. To optimize for limited bandwidth, the service enforces a 100ms minimum publish interval per unique CAN ID using a hash map of timestamps, with rate-limiting handled in a dedicated goroutine to maintain non-blocking performance. Messages that pass the interval check are published; all messages, regardless of publication, are written to an in-memory circular buffer and flushed to PostgreSQL once per second or after 1 million messages, in batches of 10,000. This architecture enables real-time telemetry while ensuring that all data is stored locally for lossless recovery and later upload by services like Epic Shelter.

Epic Shelter: Epic Shelter is a Python-based service that guarantees long-term durability and cloud synchronization of all telemetry data collected on the TCM, including messages not published live via MQTT. It reads from the `gr25_message` table in the local PostgreSQL database, populated by the MQTT service, where each row contains a raw CAN message, metadata such as vehicle ID, topic, timestamps, node IDs, and a sync flag. Running in the background, Epic Shelter checks for connectivity every 5 minutes and, upon detecting a stable connection, initiates a multi-stage upload pipeline. Unsynced messages are partitioned into time or size based chunks, converted to compressed Parquet format using GPU-accelerated cuDF, and uploaded to Amazon S3 via multipart transfer for reliability [2]. After each successful upload, Epic Shelter publishes an MQTT message to `epic-shelter/gr25` with a signed S3 URL, prompting the Mapache Ingest service to fetch and load the data. This pull-based model ensures reliable, resumable ingestion of high-frequency telemetry even in intermittently connected environments.

V SINGLESTORE

SingleStore is the core database engine behind Mapache. It is a distributed SQL database designed for HTAP (Hybrid Transactional/Analytical Processing) workloads, meaning it can handle low-latency, high-frequency ingestion and point reads/writes, while simultaneously supporting fast analytical queries on large datasets. This dual capability is critical for our use case, where real-time vehicle data is constantly streaming in while people query that same data for performance analysis, debugging, and visualization.

Under the hood, SingleStore uses a shared-nothing, distributed architecture [3]. The system is composed of three layers: aggregators, which handle query parsing and routing; leaf nodes, which execute the actual queries in parallel across data partitions; and a tiered storage engine, which

seamlessly integrates memory, SSD, and remote object storage (like Amazon S3). This architecture allows the database to scale horizontally, maintain low query latency under heavy load, and retain massive amounts of historical data without sacrificing performance.

```sql
SQL
CREATE TABLE `signal` (
  `timestamp` bigint(20),
  `vehicle_id` longtext,
  `name` longtext,
  `value` double,
  `raw_value` bigint(20),
  `produced_at` datetime(6),
  `created_at` datetime(6),
  PRIMARY KEY (`timestamp`,`vehicle_id`,`name`),
  SORT KEY `produced_at` (`produced_at`,`name`,`vehicle_id`),
  SHARD KEY `__SHARDKEY` (`timestamp`,`vehicle_id`,`name`)
);
```

Figure 5: Signal table schema definition

To match our telemetry model, we designed the `signal` table schema for both ingestion speed and efficient querying. Each row represents a decoded signal from a CAN message. The `timestamp` is when the message was received by the TCM, while `produced_at` reflects the in-message production time (if available). `vehicle_id` and name uniquely identify the data source and signal type. value contains the scaled, engineering-unit representation, while `raw_value` stores the original integer payload for debugging or verification.

The composite primary key on `(timestamp, vehicle_id, name)` ensures uniqueness and supports rapid time-based lookups. The sort key on `produced_at` improves performance for chronological queries, and the shard key, which mirrors the primary key, ensures even distribution across all leaf nodes. This layout enables parallel execution of queries, allowing us to scale linearly with cluster size and maintain fast response times even under heavy analytical workloads.

By leveraging SingleStore's HTAP capabilities, we're able to stream telemetry data in real-time at high volume while still supporting sub-second queries for dashboards, time-series graphing, and lap comparisons — all within the same unified storage engine.

VI NANOMQ

NanoMQ is our MQTT broker, serving as the core messaging layer between the TCM and the Ingest service. NanoMQ is a lightweight, high-performance, open-source MQTT broker built in C and designed specifically for edge computing scenarios. Its minimal memory usage, low-latency

architecture, and native support for MQTT 5.0 make it an ideal choice for real-time telemetry pipelines running on embedded hardware like the Jetson Orin Nano.

MQTT (Message Queuing Telemetry Transport) is a lightweight publish/subscribe messaging protocol designed for high-throughput communication over unreliable or constrained networks. In the GR25 system, the TCM publishes CAN-derived messages to MQTT topics, and the Ingest service subscribes to relevant topic patterns to receive and process the data. This decoupling between data producers and consumers simplifies scalability and improves reliability.

VII VEHICLE SERVICE

The Vehicle service manages all metadata related to vehicles, trips, and laps in the Mapache. Each vehicle is uniquely identified and associated with key information such as vehicle type (e.g., gr25), descriptive labels, and a secure upload key used for authentication when ingesting data from the TCM. In addition, the service tracks trips, which represent time-bounded data collection sessions, and laps, which segment those trips into smaller analysis-friendly intervals.

This abstraction enables clean organization of telemetry data and provides an essential layer of context for downstream services. It also enables the creation of test or virtual vehicles, which are used to isolate development and staging data from production telemetry. This ensures that real-time dashboards and analysis tools remain accurate, while still enabling thorough testing of the full data pipeline.

VIII INGEST SERVICE

The GR25 Ingest service is responsible for receiving CAN messages from the car through the MQTT broker, deserializing and scaling the signals, and finally saving them to the database. This service was written in Go due to its speed, low memory footprint, and concurrency capabilities. The system is built for low-latency, high-throughput ingestion and supports concurrent decoding of thousands of telemetry messages per second while remaining flexible and extensible via a centralized message definition library.

A. mapache-go

To standardize message decoding across services, we built `mapache-go` [4], a shared Go library that defines a unified structure and logic for decoding and scaling CAN messages into signals. The structure of a CAN message is represented by a Message object. Each Message is an array of Field objects. A Field defines the byte offset, size, endianness, signedness, and decoding logic for a single value inside a CAN message.

```go
type Message []Field

type Field struct {
	Name             string
	Bytes            []byte
	Size             int
	Sign             SignMode
	Endian           Endian
	Value            int
	ExportSignalFunc ExportSignalFunc
}
```

Figure 6: Message and Field structs

```go
var ECUStatusThree = mp.Message{
	mp.NewField("rr_wheel_rpm", 2, mp.Unsigned, mp.LittleEndian, func(f
mp.Field) []mp.Signal {
		signals := []mp.Signal{}
		signals = append(signals, mp.Signal{
			Name:     "rr_wheel_rpm",
			Value:    float64(f.Value)*0.1 - 3276.8,
			RawValue: f.Value,
		})
		return signals
	}),
	mp.NewField("rl_wheel_rpm", 2, mp.Unsigned, mp.LittleEndian, func(f
mp.Field) []mp.Signal {
		signals := []mp.Signal{}
		signals = append(signals, mp.Signal{
			Name:     "rl_wheel_rpm",
			Value:    float64(f.Value)*0.1 - 3276.8,
			RawValue: f.Value,
		})
		return signals
	}),
}
```

Figure 7: Example message definition

Each Field has an `ExportSignalFunc`, which determines how it maps to one or more Signal objects. The Signal object mirrors the `signal` database table schema from Figure 5, and is the

final transformation of a CAN message.

In the Figure 7 above, you can see how one of the ECU Status CAN messages is represented as a `mapache-go` Message. It contains two Fields for wheel speeds (rpm), each being 2 unsigned bytes in little endian order. Each of those fields then has a custom export function which returns one signal with the value properly scaled.

By centralizing these export functions in `mapache-go`, we maintain a clean separation between how data is defined and how it's processed, allowing the ingest service to remain generic, scalable, and easy to extend.

B. MQTT Subscription

The Ingest service first subscribes to the MQTT topic `gr25/#`, which will consume all messages from topics that begin with `gr25/`. As mentioned previously, the format for valid topics is `gr25/vehicle-id/node-id/message-id`. So for example, `gr25/gr25-main/ecu/0x004` is a valid topic but `gr25/gr25-main/0x004` is not.

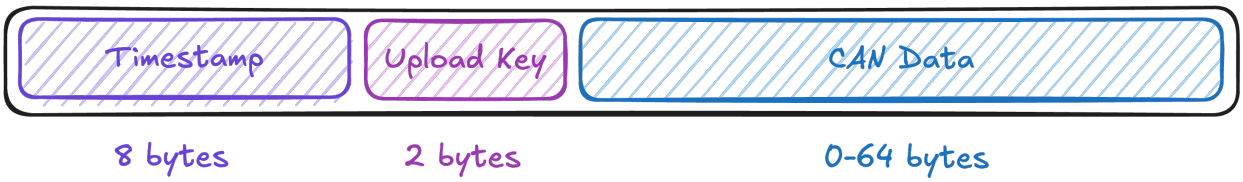

Figure 8: MQTT payload format

When a valid message is received, it gets passed to a message handler in a new Go routine to handle it asynchronously, ensuring high throughput and non-blocking operation. The incoming message payload must be at least 11 bytes: the first 8 bytes encode the message timestamp (UNIX microseconds), the next 2 bytes represent the upload key, and the remaining bytes are the raw CAN message data. Messages with an invalid or missing upload key are ignored to prevent unauthorized ingestion.

C. Signal Processing

In the message handler, first a copy of the Message definition struct is pulled from a hash map that corresponds to the Message ID of the CAN message (pulled from the MQTT topic). This message struct has a `FillFromBytes` function which automatically slices the message payload (which is the data bytes from the CAN message) according to the size of each field, decodes the byte array into integer values using the `Endian` and `Sign` mode, and populates the value of each Field. Once filled, the message is converted into a list of Signal objects via the `ExportSignals` function on the message struct.

```go
messageStruct := model.GetMessage(canID)
messageStruct.FillFromBytes(data)
signals := messageStruct.ExportSignals()

for _, signal := range signals {
	signal.Name = fmt.Sprintf("%s_%s", nodeID, signal.Name)
	signal.Timestamp = int(binary.BigEndian.Uint64(timestamp))
	signal.VehicleID = vehicleID
	signal.ProducedAt = time.UnixMicro(int64(signal.Timestamp))
	signal.CreatedAt = utils.WithPrecision(time.Now())
	CreateSignal(signal)
}
```

Figure 9: Message to Signal transformation

Before the signals are stored, they are enriched with context. The signal `Name` is prefixed with the sending Node ID, resulting in names like `ecu_rr_wheel_rpm`. The VehicleID is assigned from the topic. The original timestamp is applied to both `Timestamp` and `ProducedAt`. The `CreatedAt` value is set to the current system time. Each signal is then finally inserted into the signal table in the database.

D. Websocket Endpoint

In addition to database storage, the ingest service supports real-time streaming of incoming signals via a WebSocket endpoint. This is particularly useful for dashboards and monitoring tools that need to visualize live telemetry as it's being received.

Once the connection is established, the ingest service listens for all decoded signals. Any signal that matches the requested `vehicle_id` and `signal` name list (provided as query parameters when connecting) is streamed to the client in real time as a JSON object. This integration between the ingestion pipeline and the WebSocket layer allows users to stream vehicle data in real-time — enabling customizable dashboards, telemetry alerts, and rapid diagnostics without waiting for data to be queried from storage.

IX QUERY SERVICE

The Query Service is a Python-based microservice that provides fast, flexible access to cloud-stored telemetry data. It serves as the primary interface for exploratory analysis and visualization, transforming raw signals into analysis-ready formats with minimal latency.

Telemetry is stored in a long-format schema: each row is a single timestamped signal reading. This structure is efficient for ingestion and streaming, especially when dealing with hundreds of

signals at varying frequencies. However, analysis and visualization typically require wide-format data, where rows represent timestamps and columns represent signals. SQL-based transformation is difficult and slow due to complex joins across non-uniform timebases.

To address this, each signal is queried independently via simple SQL statements using SQLAlchemy, then loaded into memory as pandas DataFrames. Merging, resampling, and interpolation are performed with vectorized pandas operations, enabling fast and readable logic that scales well for most short to mid-range analysis tasks [5]. Though this method is more memory-intensive than pure SQL, most queries are scoped to specific trips or laps, keeping workloads small and responsive.

The service exposes a single endpoint at `/query/signals`, returning results in wide JSON format. Supported parameters include:

- `vehicle_id`: target vehicle to query
- `signals`: list of signal names to include in the result
- `trip_id`: if provided, automatically fetches start and end timestamps from the Vehicle service
- `start`, `end`: manual time bounds for the query
- `merge`: merge strategy to align signals (e.g., `smallest`, `largest`)
- `fill`: missing value fill strategy (`none`, `forward`, `backward`, `linear`)
- `tolerance`: time window for fill operations

Each signal is queried and merged according to the chosen strategy. Fill settings let users control how gaps are handled, tailoring results to specific analysis needs. Results can be returned as JSON, CSV, or Parquet, enabling integration with tools like Python, Excel, MATLAB, and Jupyter. By abstracting away SQL and schema details, the Query Service streamlines the path from raw data to insight—whether users are debugging signals, comparing laps, or running statistical models.

X DASHBOARD

The Dashboard is a web-based application that provides team members with immediate visibility into vehicle telemetry, supporting real-time monitoring, historical analysis, and rapid debugging across both development and competition settings. Built with React and TypeScript, it runs in any modern browser, making it easily accessible from virtually anywhere.

A. Live Dashboard

The live dashboard page displays incoming data from the selected vehicle. A grid based widget system allows users to customize what data they see. The widgets receive their specified signals

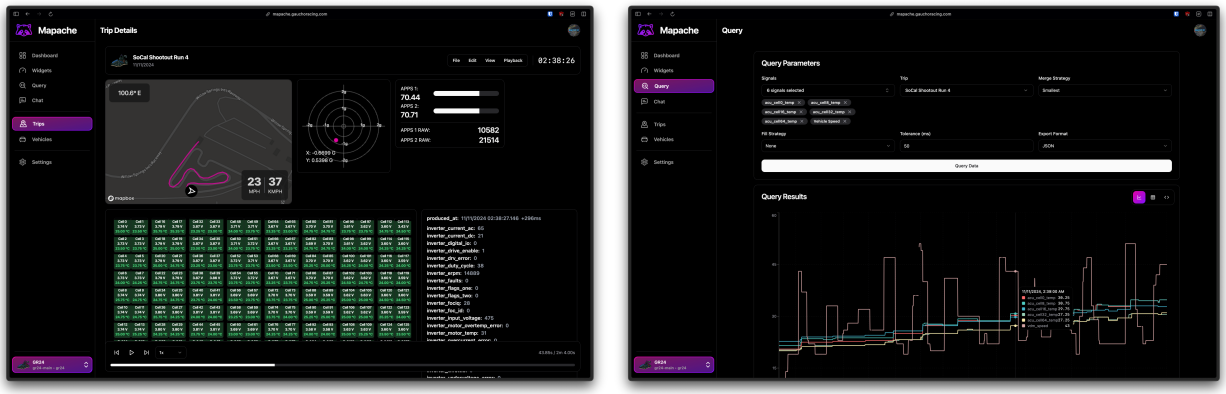

Figure 10: Trip replay system (left) and Query page (right)

from the web socket endpoint exposed by the Ingest service. Each widget keeps a rolling window of latest signals depending on what they are visualizing. For example, a live tire temperature graph may keep the last 20 signals, while an accelerometer display may only keep the latest one.

B. Trip Replay

The Trips page features a comprehensive replay system that allows users to visualize past telemetry sessions in full context. After selecting a vehicle and trip, users are presented with the same customizable, grid-based widget system used in the live view. However, instead of real-time data, each widget retrieves its signals for the entire trip duration from the Query Service. A global playback cursor lets users scrub through the trip timeline, with all widgets synchronized to display the closest data to the current timestamp. This coordinated playback enables frame-by-frame debugging, trend analysis, and post-run reviews without writing a single line of code.

C. Custom Queries

The Query page enables users to run custom analyses by selecting specific signals and visualizing them for any vehicle over a defined time range. Parameters are passed to the Query Service via an API call, and the returned data can be explored in multiple formats including interactive charts, tables, or raw JSON. For added convenience, users can select from predefined trips and laps retrieved from the Trip Service, eliminating the need to manually input timestamps. Whether used for diagnosing a subsystem or comparing performance across laps, the Query page provides a flexible and intuitive interface for extracting insights from raw telemetry.

XI PERFORMANCE AND RELIABILITY

Mapache was evaluated during full-vehicle runs at both the regional SoCal Shootout event and the FSAE Michigan EV competition, where it handled continuous telemetry across multiple events and testing sessions. These real-world deployments provided insight into its latency, throughput, and reliability under sustained load and variable LTE network conditions.

Latency: Under optimal connectivity conditions, end-to-end latency from CAN message broadcast to dashboard rendering consistently remained below 250 milliseconds. Latency was primarily distributed across MQTT transmission (80ms average), ingest service processing (10ms), and frontend rendering (30–50ms). Throughout spotty connectivity, end-to-end latency remained below 650 milliseconds.

Throughput: The TCM received and buffered over 2,000 CAN frames per second across its dual CAN interfaces. In production, the Jetson's SPI interface sustained 10+ Mbps, ensuring fast, lossless transfer of telemetry data to both the MQTT pipeline and local database.

Backpressure & Recovery: Epic Shelter successfully buffered incoming messages locally to prevent data loss. The Jetson sustained write performance while storing over 10 million messages in its local PostgreSQL database. Upon reconnection, Epic Shelter synced 128MB batches in under 5 seconds each, enabling full recovery of multi-hour sessions within minutes.

Dashboard Performance: The dashboard remained stable under load, streaming over 50 distinct signal channels at 100 Hz without dropped messages. Internal tests confirmed consistent delivery and low latency, even with many frontend widgets requesting simultaneous updates.

System Resilience: Throughout the year, including at FSAE Michigan and SoCal Shootout, Mapache maintained zero critical telemetry failures. Its modular architecture allowed isolated service restarts and horizontal scaling without interrupting data collection, significantly improving uptime, debug turnaround, and overall vehicle reliability.

XII CONCLUSION

Mapache represents a transformative shift in how Formula SAE teams approach data-driven development. By replacing legacy RF-based systems with a cloud-native, LTE-powered telemetry platform, teams can now access live vehicle data from anywhere, whether at competition, in remote test locations, or during debugging at the lab.

Looking ahead, Mapache opens new possibilities beyond telemetry. Features like real-time anomaly detection, predictive diagnostics, and driver feedback systems can be built atop its existing foundation. As vehicle complexity grows, the importance of intelligent data infrastructure will only increase, and Mapache is well-positioned to meet that challenge.

XIII ACKNOWLEDGMENTS

Mapache would not have been possible without the huge amount of support from our corporate sponsors SingleStore and Amazon Web Services (AWS).

Figure 11: Gaucho Racing at the 2025 Michigan Formula SAE EV Competition

We would also like to give a shoutout to the numerous people on Gaucho Racing who provided support for us to realize this vision: Thomas Yu, Raaghav Thirumalai, Anirudh Kumar, Shehan Seneviratne, Nikunj Parasar, Ryan Nguyen, William Ni, Vin Shin, just to name a few.

REFERENCES

[1] M. Bozdal, M. Samie, and I. Jennions, "A survey on can bus protocol: Attacks, challenges, and potential solutions," in *2018 International Conference on Computing, Electronics Communications Engineering (iCCECE)*, pp. 201–205, 2018.

[2] P. Narayanan, *GPU Driven Data Wrangling Using CuDF*, pp. 133–161. Springer Nature, 09 2024.

[3] A. Prout, S.-P. Wang, J. Victor, Z. Sun, Y. Li, J. Chen, E. Bergeron, E. Hanson, R. Walzer, R. Gomes, and N. Shamgunov, "Cloud-native transactions and analytics in singlestore," in *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*, (New York, NY, USA), pp. 1–13, Association for Computing Machinery, June 2022.

[4] B. Kathi, "mapache-go: A shared go library for handy mapache abstractions.." `https://github.com/Gaucho-Racing/mapache-go`, 2025.

[5] E. Baumann, C. Hsu, H. Buba, and T. Cox, "An introductory approach to time-series data preparation and analysis," *Annual Conference of the PHM Society*, vol. 15, 10 2023.