SAMFMS

Software Requirement Specification

Team Firewall Five:

Team Member	Student number
Herman Engelbrecht	u22512374
Morné van Heerden	u21482153
Laird Glanville	u22541332
Stefan Jansen van Rensburg	u22550055
Nicolaas Johan Jansen van Rensburg	u22592732

Contents

1. Introduction

1.1 Purpose

SAMFMS is a modular fleet management platform designed for small to medium-sized businesses. It allows organizations to track, monitor, and manage fleets through a customizable, extensible system using plugin-style modules called SBlocks.

Problem Statement:

Small to medium-sized businesses (SMBs) struggle to adopt Fleet Management Systems (FMS) due to:

1. High Costs: Traditional FMS solutions are expensive and often bundle unnecessary features.
2. Lack of Flexibility: Most systems are monolithic, forcing businesses to pay for unused functionalities.
3. Complexity: Overwhelming interfaces and steep learning curves deter SMBs with limited technical resources.
4. Scalability Issues: Growing businesses need modular systems that adapt to their evolving needs without costly migrations.

Without an affordable, modular, and user-friendly FMS, SMBs face inefficiencies in fleet tracking, maintenance, and route optimization, leading to higher operational costs and reduced competitiveness.

How will we solve this?

SAMFMS addresses these challenges by offering:

1. Modular Core (MCORE**)**: A lightweight base system with essential FMS features.
2. Plug-and-Play SBlocks: Businesses pay only for needed features (e.g., GPS tracking, maintenance scheduling).
3. Customizable Dashboard: Tailored views for different user roles (e.g., fleet managers, drivers).
4. Scalability: Seamless addition of SBlocks as business needs grow.
5. Cost-Effectiveness: Affordable entry-level pricing with optional premium modules.

2. User Stories & User Characteristics

2.1. User Characteristics:

User Type	**Role**	**Needs**
Fleet Manager	Oversees fleet operations.	Needs real-time tracking, maintenance alerts, and driver analytics.
Driver	Operates vehicles.	Requires clear route instructions and trip logs.
Admin/IT Staff	Manages system configuration.	Needs to be able to manage staff, adding or removing them according to company needs.

User Story 1: Administrator

As an administrator:

1. I want to add and remove vehicles.
2. I want to add and remove drivers.
3. I want to be able to do what a fleet manager does.
4. I want to be able to add or remove fleet managers as they join the company or leave the company to ensure security of our system.

User Story 2: Fleet Manager

As a fleet manager:

1. As a fleet manager I want to schedule regular maintenance that could be based on mileage or time as a preventative measure, so that vehicle breakdowns are minimized.
2. I want to assign drivers to available vehicles for trips, so that tasks are carried out efficiently and on schedule and avoid traffic issues.
3. I want to see which vehicles are currently available or in use, so that I can make informed scheduling decisions.
4. I want to be able to keep track of maintenance and record maintenance of my vehicles.
5. I want to track real-time GPS locations of all vehicles on a map, so that I can monitor positions and respond quickly to delays or emergencies.
6. I want to be notified of any delivery delays or route issues, so that I can take corrective action and inform stakeholders.
7. I want to track the certifications and licenses of each driver, so that I ensure only qualified drivers operate specific vehicles.
8. I want to view historical trip data (routes, stops, distance, fuel consumption), so that I can analyze past performance.
9. I want to set up geofence alerts, so that I receive notifications when vehicles enter or leave designated zones.
10. I want to see speed violations highlighted, so that I can address reckless driving.
11. I want to view live traffic data, so that I can reroute vehicles to avoid delays.

12. I want to see vehicle health status (maintenance due), so that I can prevent breakdowns.

User Story 3: Driver

As a driver:

1. I want to view the vehicle and route assigned to me for the day, so that I know what vehicle to drive and which deliveries to make.
2. I want to check in when I start and end a trip, so that the fleet manager knows my driving schedule and hours.
3. I want to receive real-time updates or changes to my assigned route, so that I can avoid delays or reroute as needed.
4. I want to see my past trips and performance metrics, so that I can monitor and improve my own driving.
5. I want to take my vehicle to the service center when scheduled or instructed, so that the vehicle remains in safe and optimal condition.

3. Functional Requirements

3.1 Subsystems

- MCore (Modular Core / Core Gateway)
- User Interface
- Security SBlock
- GPS SBlock
- Trip Planning SBlock
- Vehicle Maintenance SBlock
- Vehicle Management SBlock
- Management SBlock

3.2 Requirements

SAMFMS will:

- Provide role-based login and authentication [UI, Security, MCore]
- Provide a way to add or remove vehicles in the fleet [UI, Vehicle Management, MCore] (Admin)
- Provide a way to add or remove drivers [UI, Vehicle Management, MCore] (Admin)
- Provide a way to manage users and assign roles (Admin/Fleet Manager) [UI, Security Service, User Management]
- Provide a way to plan trips and routes for vehicles [Trip Planning SBlock] (Fleet Manager)

- Provide a way to schedule and record maintenance [Vehicle Maintenance SBlock] (Fleet Manager/Driver)

- Provide a way to track the vehicles with GPS [GPS SBlock] (Fleet Manager)

- Provide real-time trip visibility and instructions [Trip Planning, Driver Portal] (Driver)

- Provide analytics and reporting (utilization, mileage, performance, fuel, etc.) [Management Service, Dashboard]

- Provide notifications for trips, maintenance, geofencing, and assignments [Core Notifications Service]

3.3 Functional Requirements

1. MCORE (Modular Core / Core Gateway)

User Authentication & Authorization
1.1 The system shall support role-based access (Admin, Fleet Manager, Driver).
1.2 Users shall log in via email/password.

Data Interoperability
2.1 MCORE shall provide a standardized API for SBlocks to share data (vehicle IDs, driver details, trip IDs).
2.2 All SBlocks shall store data in a central database with unified schemas.

Dashboard Framework
3.1 Users shall customize dashboard widgets (drag-and-drop).
3.2 The dashboard shall aggregate data from all active SBlocks in real time.

Notification System
4.1 MCORE shall route alerts from SBlocks to users via in-app notifications.

2. User Interface

Responsive Design
5.1 The UI shall adapt to desktop, tablet, and mobile screens.
5.2 Dark/light mode shall be user-selectable.

Account Management
6.1 Users shall update profile details and upload a profile picture.
6.2 Users shall change their password via a secure modal.
6.3 Admins shall manually create accounts.
6.4 System shall prevent direct signup if users already exist.

Driver Portal
7.1 Drivers shall view assigned trips and tasks.
7.2 Drivers shall see performance analytics

Help
8.3 Help section shall provide FAQs, tutorials, and contact support.

3. GPS SBlock

Real-Time Tracking
9.1 The system shall display vehicle locations on a live map.

Geofencing
10.1 Users shall draw and manage geofences.
10.2 Entry/exit alerts shall notify fleet managers.

4. Trip Planning SBlock

Route Optimization
11.1 The system shall calculate optimal routes (fuel/time) with constraints.
11.2 The system shall allow trip constraints (avoid tolls, POIs, etc.).

Driver Instructions
12.1 Navigation shall be pushable to drivers' apps.

Trip Analytics
13.1 The system shall provide trip history stats (distance, time, completion rate).

5. Vehicle Maintenance SBlock

Maintenance Scheduling

14.1 Fleet managers shall record service logs and costs.

6. Vehicle & Driver Management (Management Service)

Vehicle Management

15.1 Admins shall add, edit, search, filter, and delete vehicles.

15.2 Vehicles shall display status, mileage, service dates, insurance.

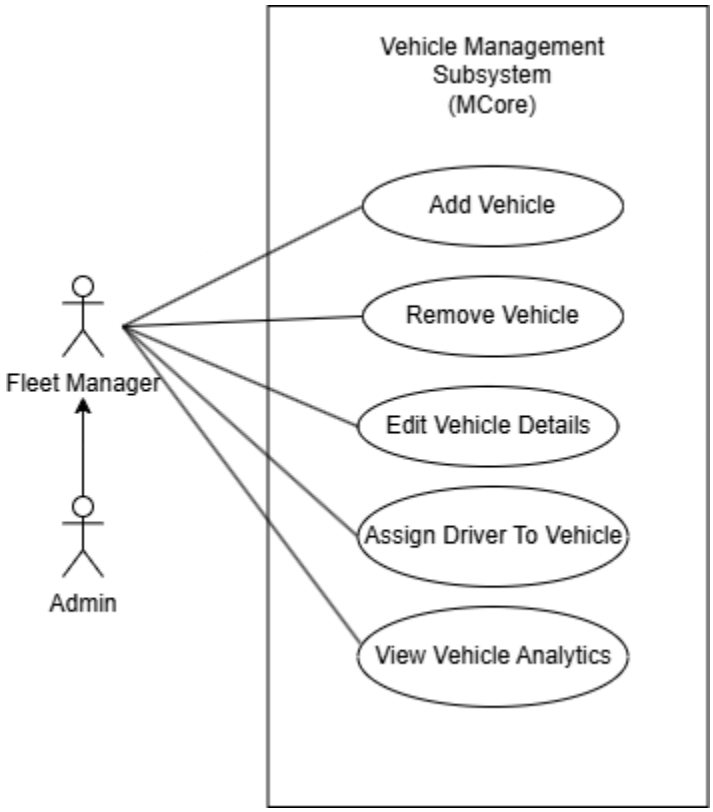
Driver Management

16.1 Admins shall add, edit, search, filter, and delete drivers.

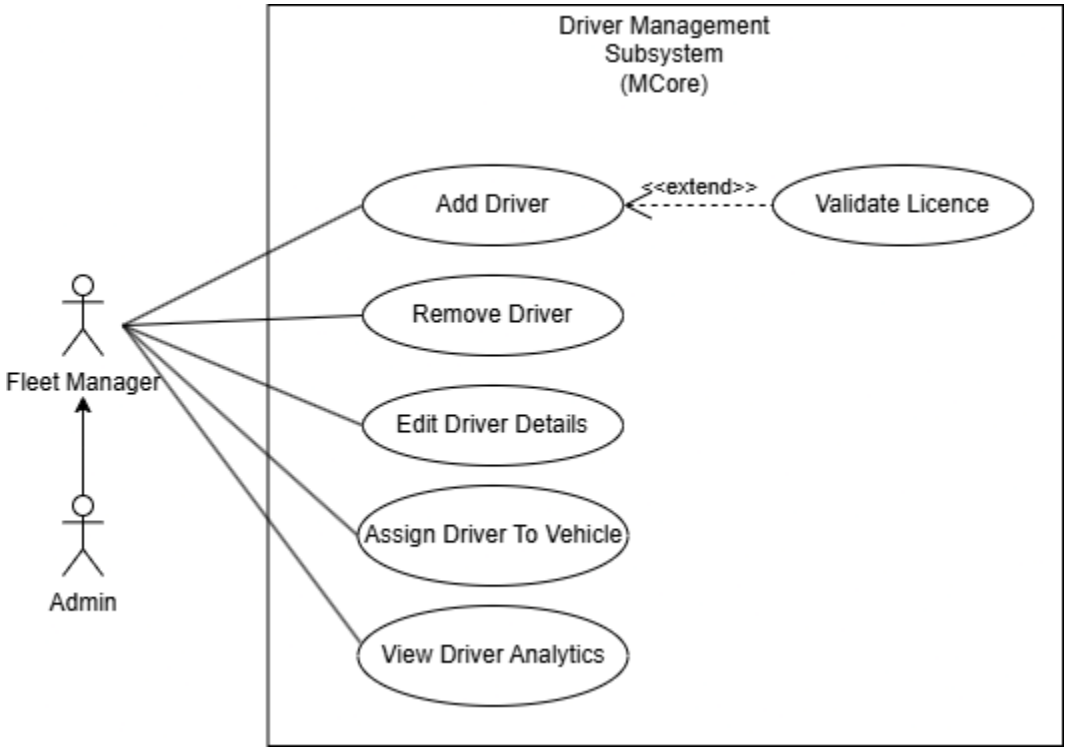
16.2 Drivers shall have license and availability tracking.

3.4 Use Cases

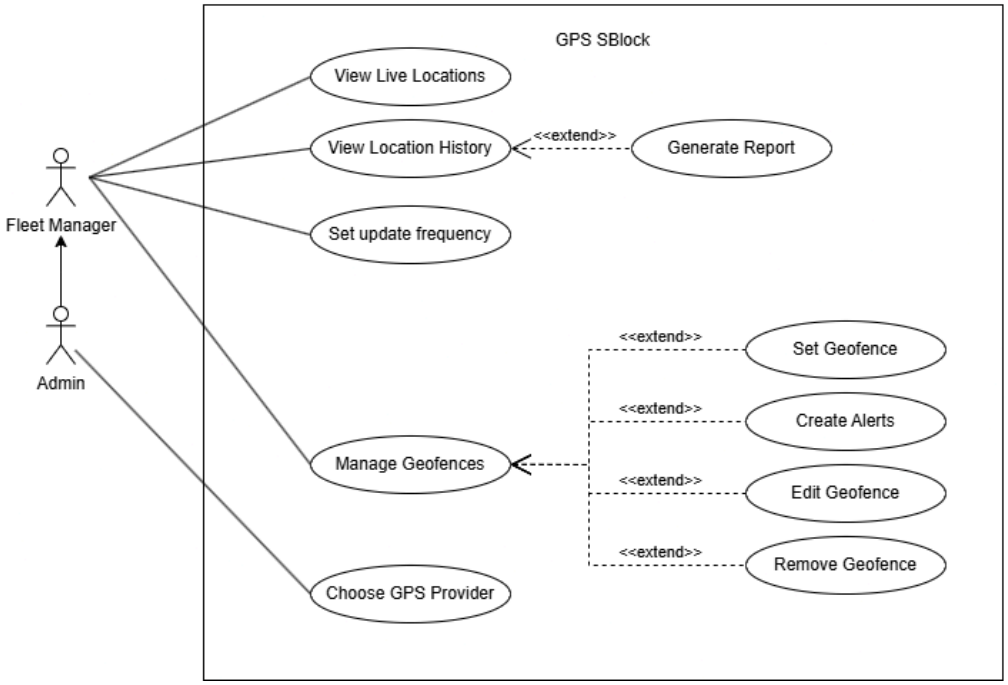
1. Use Case: Manage Vehicles



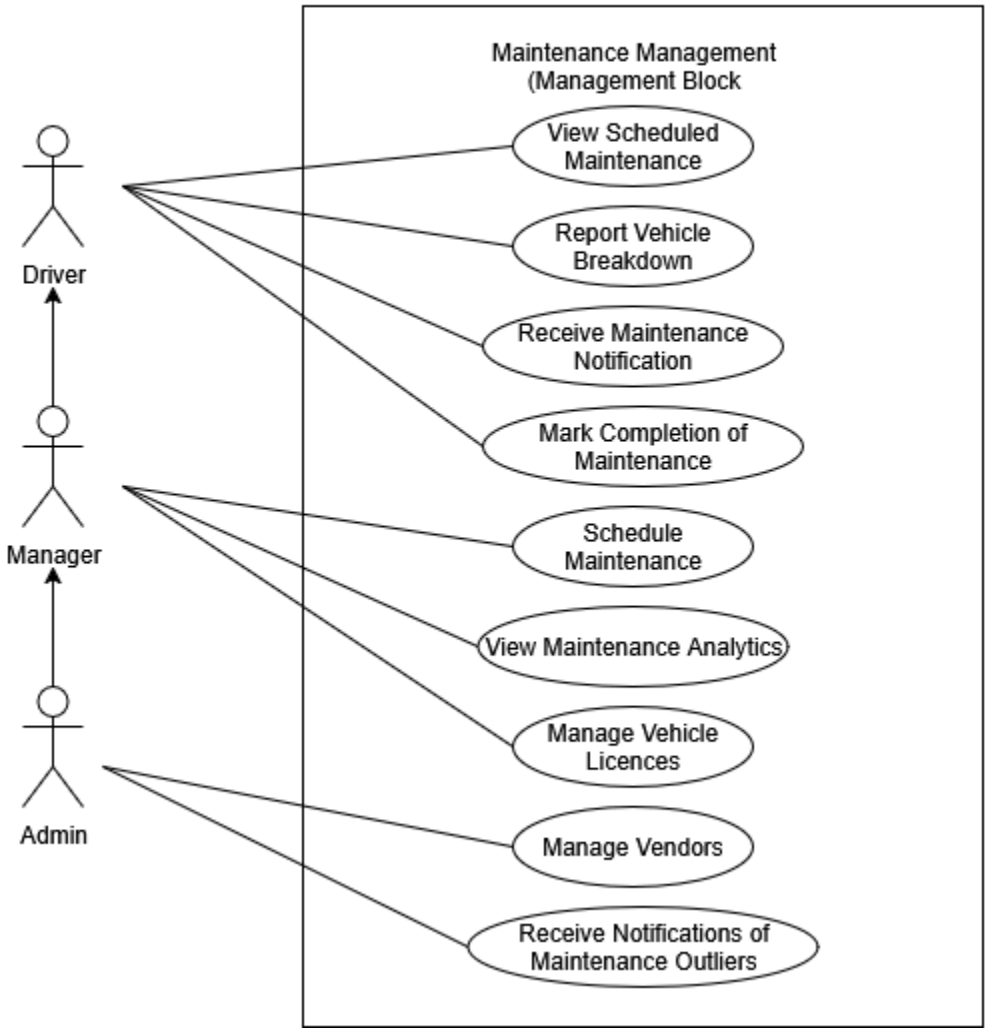
2. Use Case: Driver Management (Management SBlock)



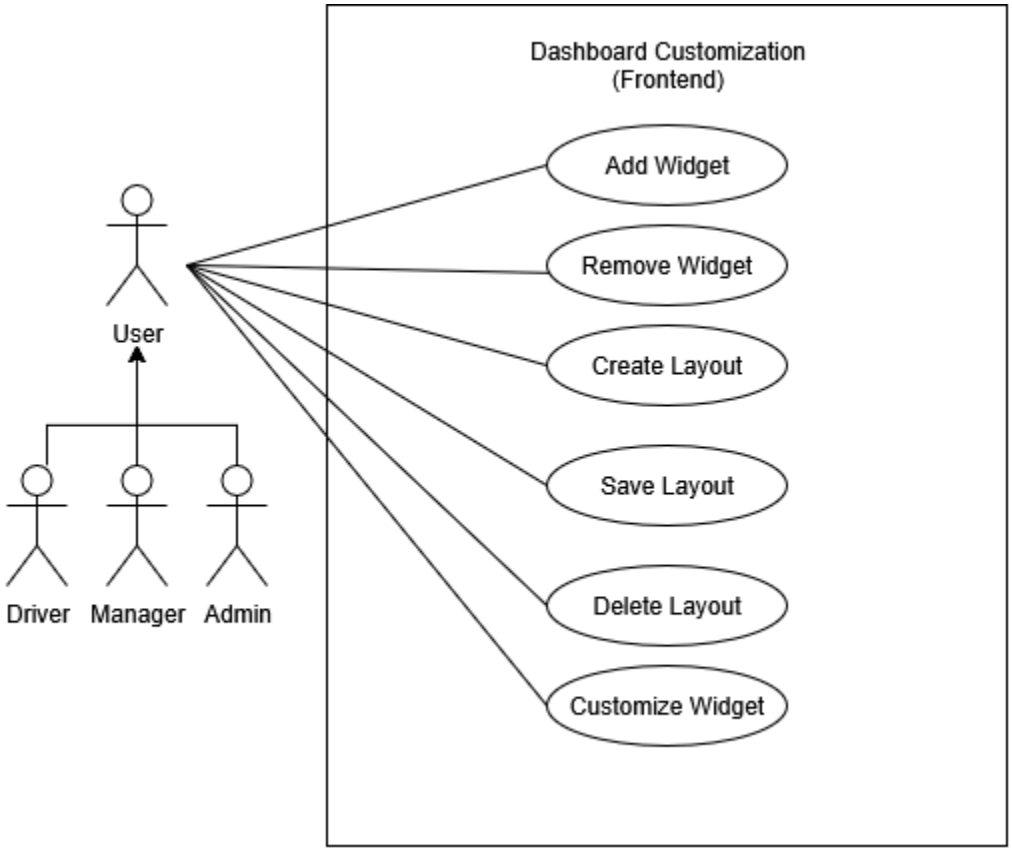
3. Use Case: Manage and view vehicle travel (GPS SBlock)



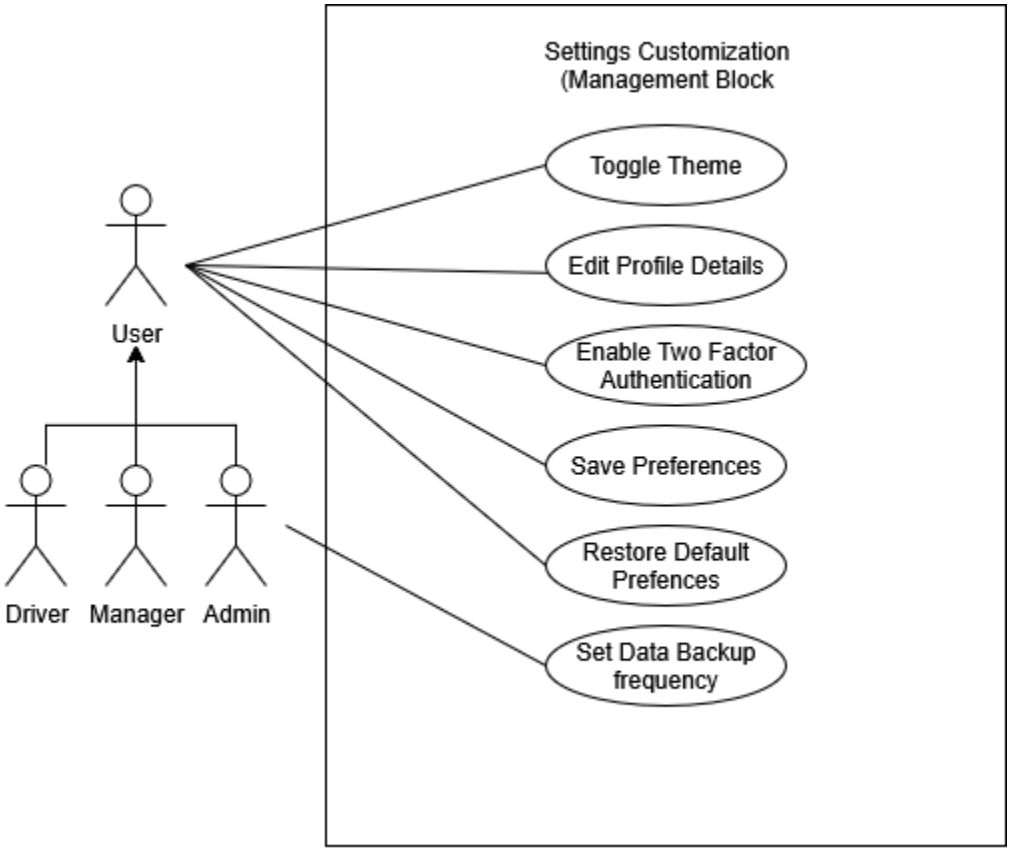
4. Use Case: Maintenance Management (Maintenance SBlock)



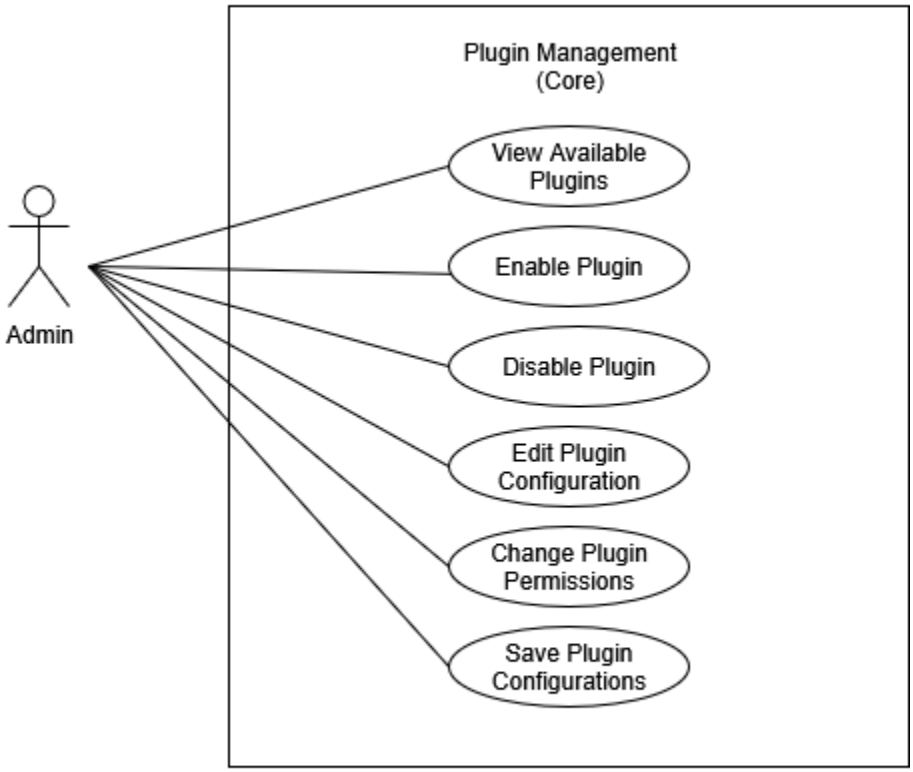
5. Use Case: Dashboard Management (Frontend UI)



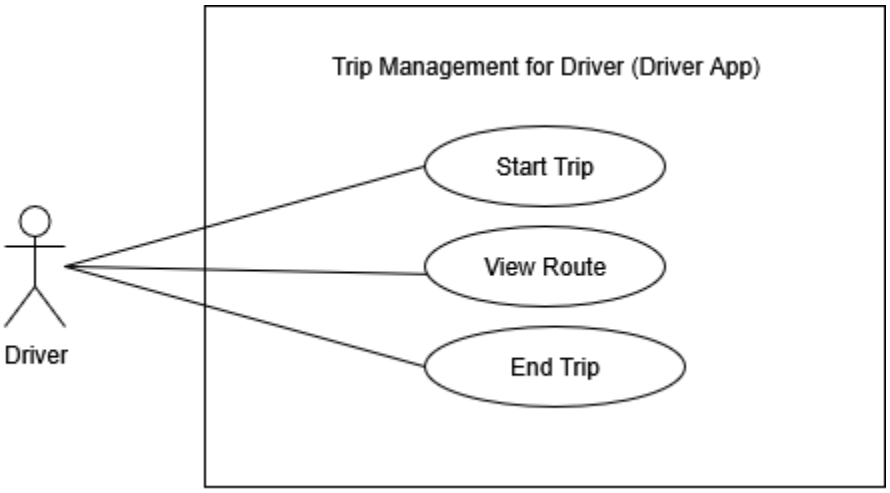
6. Use Case: Preference Management (Management SBlock)



7. Use Case: Plugin Management (Core)



8. Use Case: Trip Management for Driver (Trip Planning SBlock)



4. Quality Requirements

1. Availability and resilience

Target: 99.5 percent monthly uptime on the core path (gateway to message queue to service blocks). Every cross service call has a timeout so the app does not hang, and Core returns a clear timeout message if something is slow or unreachable. This is covered by automated integration tests that bring containers up cleanly and block deploys when tests fail.

2. Performance

Targets: GPS updates appear in the UI within about 1 second. Key dashboard widgets render in about 0.5 seconds, with heavier summaries allowed a few seconds more. Core adds no more than about 200 ms when routing. Analytics and routes return in about 1 seconds.

Measured: Load tests at 250 requests per second matched the baseline at 1 request per second with zero errors and no added latency, confirming stable performance under high concurrency.

3. Security

Targets: Token based auth with permission checks at each service boundary. Account lock after 5 failed sign ins (configurable). Passwords stored with strong one way hashing. HTTPS for all client to server traffic. Service to service calls are authenticated. Critical actions are logged with user id, timestamp, and a support token.

Measured: Auth and permission checks are covered by unit and integration tests that run automatically on pushes to main and development.

4. Reliability and data safety

Targets: Safe retries with idempotency and de duplication. Correlation IDs on requests and messages so we can trace issues end to end.

Measured: Driver behavior detection was evaluated on about 9.4 million samples of driving data and achieved about 85 percent accuracy for aggressive braking and acceleration alerts, helping keep false alerts down.

5. Maintainability and operability
Targets: Independent deploys for each service block. Consistent response shapes for success and errors. Health and readiness checks for Core and service blocks. Shared coding and API guidelines.

Measured: Unit tests target 65 percent coverage per service block, focusing on core logic. Unit tests run automatically on pushes to main and development. Integration tests tear down and restart all containers and block merges or deploys if anything fails.

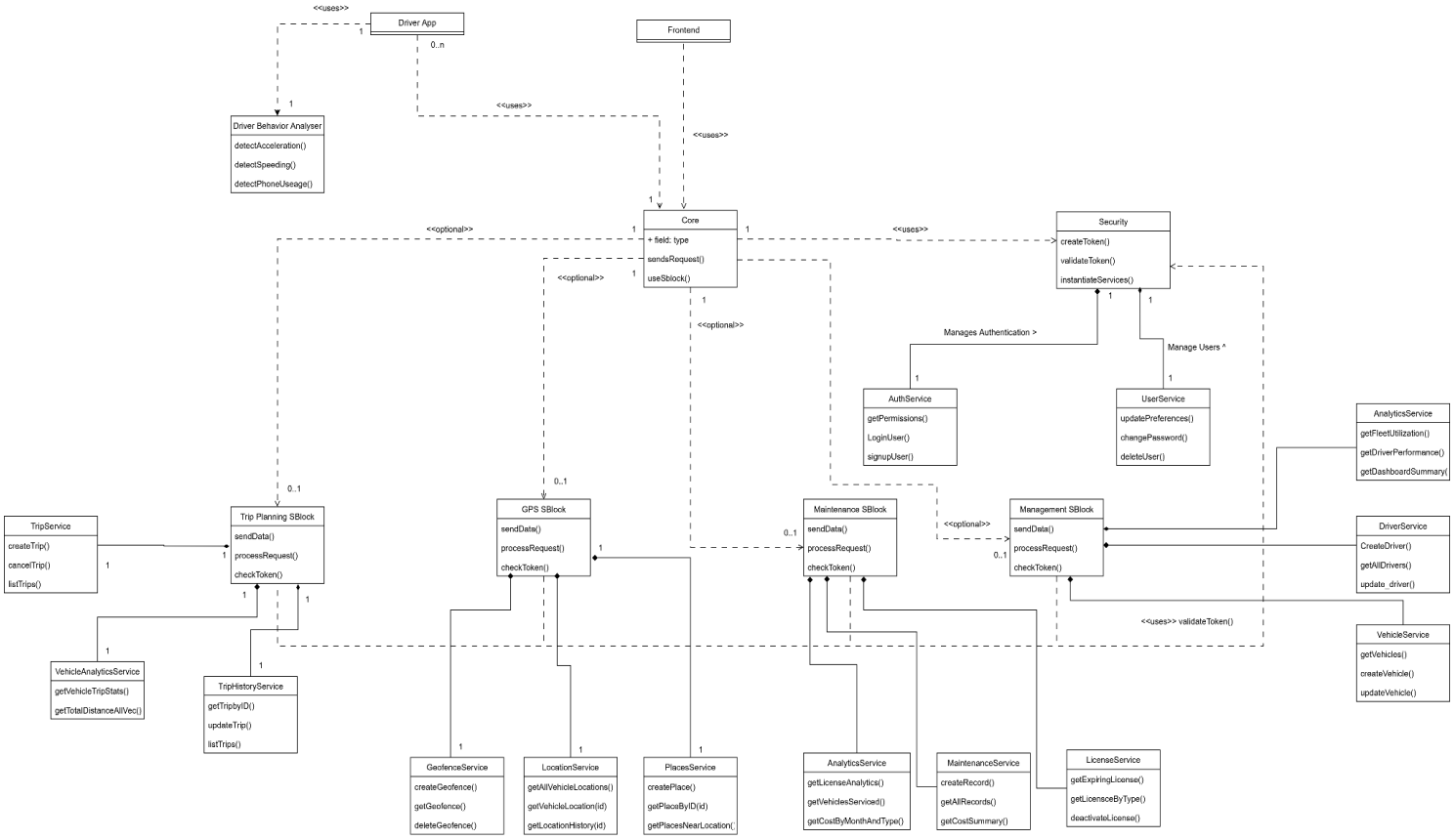
7. Usability
Targets: Clear, human readable errors and timeouts that include the support token. A responsive web UI that remembers preferences such as light or dark theme. Customizable dashboard layout so people can put key metrics up front.

Measured: Short task based usability sessions used a UEQ style questionnaire on a 1 to 7 scale. Results were generally positive, with feedback to improve signposting and perceived performance, which we fed into UI tweaks.

8. Capacity and scalability
Targets: Horizontal scaling for Core and service blocks, independently.

Measured: The system is aimed at small and medium fleets (up to about 50 vehicles). Load testing at 250 rps showed the backend handled this level of demand without errors and with latency matching the 1 rps baseline, giving headroom for growth.

5. Domain Model



6. Technology Choices

1. Backend: Python with FastAPI

We use Python because it is readable and has many ready made libraries, which helps us ship without tricky code. FastAPI lets the server handle many requests at once without blocking on database or queue calls. It also provides automatic API docs you can use in the browser, which makes testing and maintenance easier.

How this supports our targets: it keeps Core routing work light, helps us keep response shapes consistent, and makes it straightforward to test endpoints and error paths.

Trade offs: Python is not the fastest at heavy number crunching. We keep heavy work focused on input and output, or run it in background consumers. When needed we move hot paths to workers better suited for the job.

2. Message queue: RabbitMQ

The queue sits between services so they do not have to be up at the same time. If one service slows down or restarts, others can keep going. The queue smooths traffic so short bursts do not overwhelm workers. We can direct messages to the right worker groups with simple labels.

How this supports our targets: it allows graceful degradation when a service block is down, and keeps publish and consume latency low under normal load while absorbing spikes.

Trade offs: a queue is another thing to run and watch. We keep this simple with health checks, clear timeouts, and a clean path for messages that need a second try. A message is removed only after a worker confirms it finished, so retries do not create duplicate work.

3. Database: MongoDB

Our data changes shape over time. Storing records as flexible documents lets us evolve without heavy migrations. Vehicle to trip to telemetry is a natural fit for nested data. Location data is first class, so storing points, routes, and geofences and running map queries is straightforward.

How this supports our targets: it handles high write rates for telemetry and logs, and keeps read latency low for dashboards by letting us fetch what we need in one go.

Trade offs: writes that touch many documents at once are harder than in a relational setup. We design data so most writes are per document and atomic. We protect integrity with unique indexes and clear checks in the application.

4. Deployment: Docker and Docker Compose

Containers give us the same environment in development, test, and production. Each service block runs in its own container, which matches our modular design. Compose files describe the full stack so we can bring it up quickly for local work and for repeatable test runs in CI.

How this supports our targets: faster rollouts and rollbacks help uptime. Repeatable environments reduce surprises and improve reliability.

5. Web frontend: React

React encourages small, testable components we can reuse. The ecosystem is strong and hiring is straightforward. Patterns for data flow are well known, which helps keep screens predictable.

How this supports our targets: the UI stays responsive and can reflect back end events quickly, like live GPS updates and notifications. Clear patterns make the client easier to maintain and test.

6. Frontend styling: Tailwind CSS

Tailwind lets us build and adjust layouts quickly with small class names. Shared spacing, type, and color choices keep the look consistent. The CSS surface stays small, which reduces churn.

How this supports our targets: faster UI changes improve time to fix and help us keep the dashboard focused on what matters first.

7. Mapping: Leaflet with React Leaflet

Leaflet is lightweight and open source. It handles live markers, routes, geofences, and grouping many points on screen. It fits well with our location data model.

How this supports our targets: smooth map updates help us meet the real time feel for GPS, while keeping the page quick to load and interact with.

8. Driver app: React Native with Gradle

The driver app is built with React Native, so we can share ideas and some code patterns with the web. Gradle drives the Android build and release steps.

How this supports our targets: a shared way of building screens speeds up fixes and features. The app stays responsive, updates quickly in response to events, and remains simple to test and release.

Trade offs: mobile performance can vary by device.

9. Mobile styling: NativeWind

NativeWind lets us style React Native screens with a class name approach similar to Tailwind on the web. That means familiar spacing, type, and color tokens across products.

How this supports our targets: consistent design speeds up delivery and reduces mistakes. It makes it easier to keep the driver app clear and fast without one off styles.

Trade offs: style classes can drift if not reviewed.

7. Architectural structural design & requirements

Quality Attribute Mapping

Attribute	Design Strategy
Scalability	Microservices + async queues allow dynamic SBlock scaling
Usability	MVVM ensures modular UI + JS-Lib decouples logic
Security	Token-based auth + Role-Based Access Control (RBAC)
Performance	Async queues enable parallel processing; Redis caching planned
Portability	Containerized services = platform independence

Architectural Design Summary for SAMFMS

Microservices

- **Independent Deployability:** Each SBlock can be updated, deployed, or scaled without affecting the others, ensuring minimal downtime.
- **Fault Isolation:** Failure in one microservice (e.g., Maintenance SBlock) does not cascade to the whole system.
- **Flexibility:** SMBs can add or remove SBlocks based on their needs, aligning with our plug-and-play approach.

Event-Driven

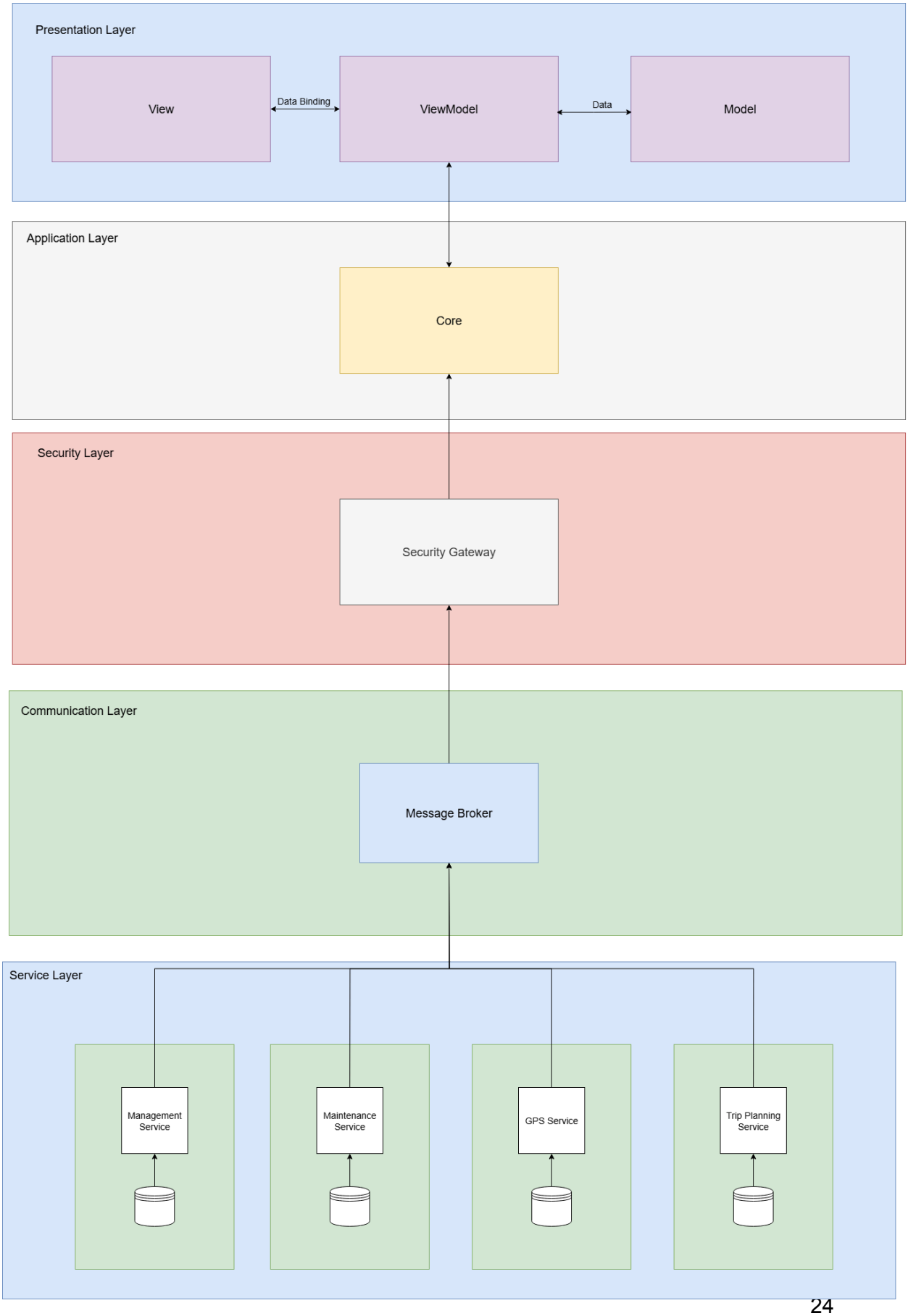
SAMFMS requires decoupled communication between MCore and SBlocks while maintaining near-real-time updates. Event-driven architecture is ideal because:

- **Asynchronous Communication:** MCore sends events to queues; SBlocks subscribe and react independently.

- **Loose Coupling:** Services do not wait for each other, reducing bottlenecks and increasing responsiveness.
- **Extensibility:** New services can subscribe to relevant events without modifying MCore or other SBlocks.

MVVM

- **Separation of Concerns:**
    - **Model:** SBlocks handles the backend logic
    - **ViewModel:** The MCore acts as the intermediary, handling business logic and API calls.
    - **View:** React components render UI, responding automatically to ViewModel changes.
- **Maintainability:** Changes in logic or UI do not tightly couple, reducing regression risk.
- **Testability:** ViewModels can be tested independently of the UI, improving quality assurance.


Presentation Layer
View
Data Binding
ViewModel
Data
Model
Application Layer
Core
Security Layer
Security Gateway
Communication Layer
Message Broker
Service Layer
Management Service
Maintenance Service
GPS Service
Trip Planning Service

MVVM (Model-View-ViewModel) Pattern

View (Frontend)

- The Frontend represents the View because it's the user interface layer that displays data and captures user interactions. It renders the visual elements and handles UI events without containing business logic.

ViewModel (Core)

- The Core acts as the ViewModel because it serves as an intermediary between the View and Model. It manages the presentation logic, handles data binding, exposes data in a format suitable for the View, and processes UI commands.

Model (SBlocks + Databases)

- The SBlocks with their databases represent the Model because they contain the business logic, data validation, and data persistence. Each SBlock (Management, Maintenance, GPS, Trip Planning) encapsulates specific domain logic and manages its own data storage.

Event-Driven Architecture Pattern

Producer (SBlocks)

- The SBlocks are Producers because they generate events when business operations occur (like trip updates, maintenance alerts, GPS location changes). They publish these events to notify other parts of the system about state changes.

Broker (Message Queue)

- The Message Queue acts as the Broker by receiving events from producers and routing them to appropriate consumers. It decouples the event publishers from subscribers and ensures reliable event delivery.

Consumer (Core)

- The Core functions as a Consumer because it subscribes to and processes events from the Message Queue. It reacts to domain events and coordinates responses across the system, updating the ViewModel state accordingly.

Microservices Architecture Pattern

Services (SBlocks)

- Each SBlock (Management, Maintenance, GPS, Trip Planning) represents an individual microservice because:
    - They are independently deployable units with their own databases
    - Each focuses on a specific business domain (single responsibility)

- They can be developed, scaled, and maintained separately
- Each has its own data storage, ensuring data isolation