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/Pain Points
Fleet Manager	**Oversees fleet operations.**	**Needs real-time tracking, maintenance alerts, and driver analytics.**
Driver	**Operates vehicles.**	**Requires clear route instructions, trip logs, and emergency alerts.**
Admin/IT Staff	**Manages system configuration.**	**Requires ability to add and remove SBlock plugins. Also needs to be able to manage Fleet Managers, 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 add and remove functionality.
4. I want to be able to do what a fleet manager does.
5. 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, 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 monitor the fuel consumption of each vehicle, so that I can identify inefficiencies and reduce fuel costs.
5. I want to receive alerts when a vehicle is due for service, so that I can take timely action and avoid unscheduled downtime.
6. I want to generate weekly and monthly reports on vehicle usage, so that I can analyze performance and optimize fleet operations.

7. I want to monitor how many hours each driver has worked, so that I can comply with labor regulations and prevent overworking.
8. 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.
9. I want to be notified of any delivery delays or route issues, so that I can take corrective action and inform stakeholders.
10. I want to track the certifications and licenses of each driver, so that I ensure only qualified drivers operate specific vehicles.
11. I want to record any accidents or breakdowns, so that I can investigate causes and improve preventative procedures.
12. I want to view the complete history of each vehicle (maintenance, usage, incidents), so that I can make informed decisions about repairs or replacements.
13. I want to filter vehicles by status (moving, idle, offline), so that I can quickly identify issues.
14. I want to view historical trip data (routes, stops, distance, fuel consumption), so that I can analyze past performance.
15. I want to set up geofence alerts, so that I receive notifications when vehicles enter or leave designated zones.
16. I want to see speed violations highlighted, so that I can address reckless driving.
17. I want to export trip reports (PDF/CSV), so that I can share data with stakeholders.
18. I want to click on a vehicle to see driver details (name, contact info), so that I can communicate quickly.
19. I want to view live traffic data, so that I can reroute vehicles to avoid delays.
20. I want to set custom alerts (e.g., for unauthorized stops), so that I can prevent misuse.
21. I want to see vehicle health status (engine faults, maintenance due), so that I can prevent breakdowns.
22. I want to receive emergency alerts (e.g., accidents), so that I can respond immediately.

User Story 3: Driver

As a driver:

1. I want to view the truck 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 the status and condition of my assigned vehicle, so that I can report any issues before starting the trip.

3. I want to check in when I start and end a trip, so that the fleet manager knows my driving schedule and hours.
4. I want to log any vehicle problems during or after a trip, so that maintenance can be scheduled promptly.
5. I want to receive real-time updates or changes to my assigned route, so that I can avoid delays or reroute as needed.
6. I want to log fuel refills, mileage, and location, so that the company can track expenses and schedule maintenance accurately.
7. I want to submit feedback after completing my trip, so that I can report delays, road conditions, or concerns.
8. I want to upload proof of delivery (e.g., signature or photo), so that there is evidence that the delivery was completed successfully.
9. I want to see my past trips, hours driven, and performance metrics, so that I can monitor and improve my own driving.
10. I want to receive automatic maintenance reminders, so that I know when my vehicle needs service.
11. I want to acknowledge alerts, so that my fleet manager knows I acknowledge them.
12. I want to upload photos of vehicle issues, so that mechanics can diagnose problems faster.
13. I want to take my vehicle to the service center when scheduled or instructed, so that the vehicle remains in safe and optimal condition.
14. I want to see urgent vs. routine alerts, so that I can prioritize repairs.
15. I want to receive notifications for recalls, so that I can take action if needed.
16. I want to contact support directly from the alert, so that I can get help fast.

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 functionality (SBlock/Plugins) [UI, MCore] (Admin)
- 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 or SSO (Google/Microsoft).

SBlock & Plugin Management
2.1 The Core orchestrates the SBlocks and allows activation/deactivation.
2.2 The system shall validate plugin compatibility before installation.
2.3 Admins shall refresh and test plugin connectivity.

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

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

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

2. User Interface

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

Account Management
7.1 Users shall update profile details and upload a profile picture.
7.2 Users shall change their password via a secure modal.
7.3 Admins shall invite users or manually create accounts.
7.4 System shall prevent direct signup if users already exist.

Driver Portal
8.1 Drivers shall view assigned trips and tasks.
8.2 Drivers shall send emergency alerts with GPS coordinates **(future)**.
8.3 Drivers shall see performance analytics (Driver ScoreCard).

Reporting & Help
9.1 Users shall generate prebuilt reports (fuel usage, mileage, idle time).
9.2 Reports shall export to PDF/Excel.
9.3 Help section shall provide FAQs, tutorials, and contact support.

3. GPS SBlock

Real-Time Tracking
10.1 The system shall display vehicle locations on a live map.
10.2 Historical routes shall be replayable with annotations.

Geofencing
11.1 Users shall draw and manage geofences.
11.2 Entry/exit alerts shall notify fleet managers.

Telemetry Data
12.1 The system shall capture and store speed, mileage, heading.
12.2 Data shall be accessible via API for third-party integrations.

4. Trip Planning SBlock

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

Multi-Stop Planning
14.1 Fleet managers shall drag-and-drop stops to reorder.
14.2 Stops shall auto-group for efficiency.

Driver Instructions
15.1 Navigation shall be pushable to drivers' portal.
15.2 Trips shall include fuel stations and rest areas.

Trip Notifications
16.1 Fleet managers shall be notified when trips start or complete.
16.2 Drivers shall receive trip start/end reminders.

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

5. Vehicle Maintenance SBlock

Maintenance Scheduling
18.1 The system shall trigger alerts based on mileage/time/OEM guidelines.
18.2 Fleet managers shall assign tasks to garages/mechanics.
18.3 Drivers shall log maintenance completion.
18.4 Fleet managers shall record service logs and costs.
18.5 Predictive maintenance shall forecast service needs.

Diagnostics Integration
19.1 OBD-II data shall be ingested to predict engine issues.
19.2 Mechanics shall annotate records with notes/photos.

Parts Inventory
20.1 The system shall track spare parts with stock alerts.
20.2 Barcode scanning shall log part installations.

6. Vehicle & Driver Management (Management Service)

Vehicle Management
21.1 Admins shall add, edit, search, filter, and delete vehicles.
21.2 Vehicles shall display status, mileage, service dates, insurance.
21.3 Vehicles shall be exportable in bulk.

Driver Management
22.1 Admins shall add, edit, search, filter, and delete drivers.
22.2 Drivers shall be assigned/unassigned to vehicles.
22.3 Drivers shall have license and availability tracking.

Assignments, Mileage & Fuel
23.1 Drivers shall be assigned to vehicles with start/end times.
23.2 Fleet managers shall update mileage records.
23.3 Fuel records shall be logged with liters, cost, and odometer readings.

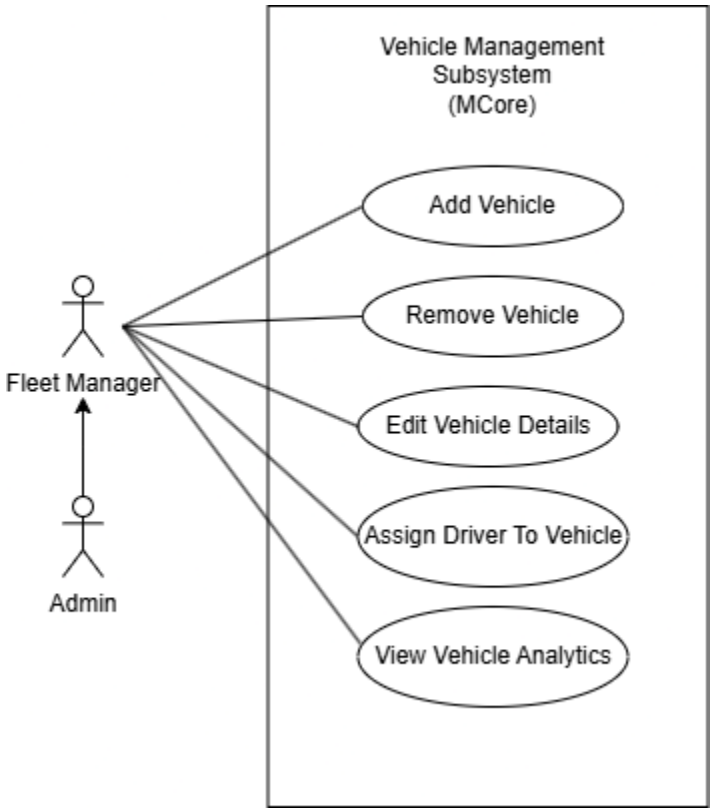
Analytics & Notifications

24.1 Fleet managers shall see driver utilization and performance stats.

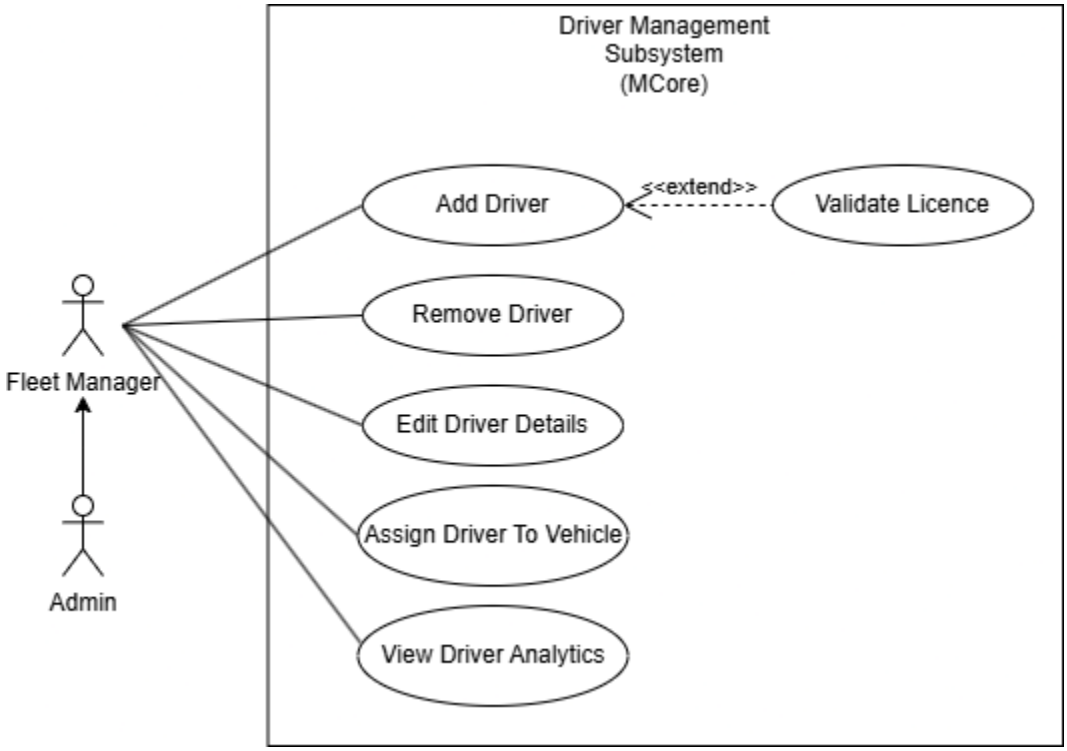
24.2 Notifications shall include system, assignment, and maintenance alerts.

3.4 Use Cases

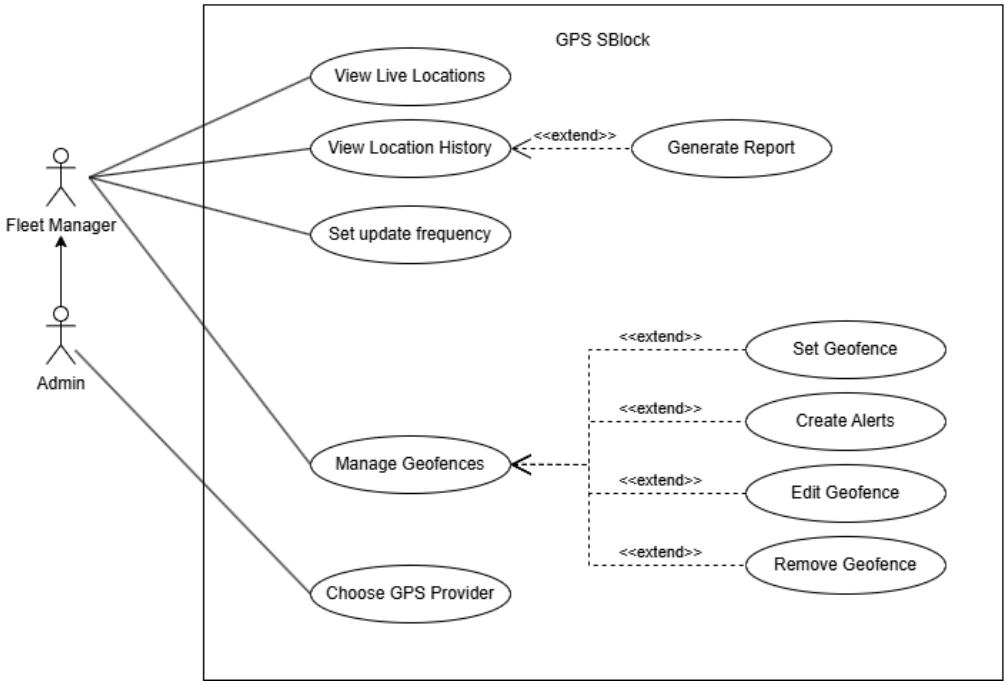
1. Use Case: Manage Vehicles



2. Use Case: Track Fleet (GPS SBlock)



3. Use Case: Schedule Maintenance (Maintenance SBlock)



4. Quality Requirements

1) Availability & Resilience

- Uptime target: 99.5% monthly for the Core path (gateway <-> message queue <-> SBlocks).
  Why: This balances reliability with our current deployment footprint and what our circuit breakers, retries, and queue-based decoupling already support. 99.5% keeps the system “generally available” without over promising to prospective clients.

- Graceful degradation when a service block (SBlock) is down.
  Why: The Core is designed to keep healthy services usable even if one SBlock is unavailable, so operations don’t come to a stop on the whole system.

- Bounded waits and fail-fast behavior.
  Why: Every cross-service request has a timeout to prevent “hung” requests and to keep the UI responsive. Core returns a timeout message when an SBlock is slow or unreachable.

2) Performance

- GPS: updates reach the UI within ≤2 seconds under normal conditions.
  Why: “Feels real-time” is essential for tracking decisions; our lightweight write-path and queue fan-out make this realistic.

- Dashboard load: key widgets render in ≤3 seconds; secondary data can take up to 5 seconds.
  Why: Users need a usable screen fast; we prioritize above-the-fold metrics and let heavier summaries arrive just after.

- Core proxy overhead: ≤200 ms added latency for requests routed through Core.
  Why: The Core should be a thin router; this keeps the gateway from becoming a bottleneck.

- Analytics/route calculations: return results in ≤5 seconds.
  Why: These are compute/heavy I/O paths; 5s keeps them practical while we cache safe aggregates to speed repeat views.

- Message queue delivery: publish -> consume latency ≤1 second within the same region.
  Why: We use asynchronous messaging for decoupling but still expect near-real-time behavior for most flows.

3) Security

- Authentication & authorization with tokens and permissions enforced at service boundaries.
  Why: Least-privilege access is already built in, and enforcement at each service limits e

- Account protection: lockout after 5 failed sign-ins (configurable), short temporary lock.
  Why: Thwarts brute-force attempts without creating support headaches.

- Password handling: stored using strong one-way hashing.
  Why: Prevents credential disclosure even if a user table is exposed.

- Transport security: all client <-> server traffic over HTTPS; service-to-service calls authenticated.
  Why: Protects data in transit and prevents unauthenticated internal calls.

- Auditing: critical actions (e.g., assignments, maintenance changes, user/admin changes) are logged with user ID, timestamp, and correlation ID.
  Why: Supports investigations, compliance, and reliable rollbacks of human error.

- Abuse throttling where applicable (e.g., invitations/OTP verifications).
  Why: Limits automated abuse and accidental hammering of public flows.

4) Reliability & Data Safety

- Idempotency & de-duplication for requests.
  Why: Retries are inevitable with distributed systems; de-duplication avoids double-writes and inconsistent state.

- Correlation IDs across requests and messages.
  Why: Lets us trace a problem from the UI through Core and into an SBlock in seconds.

5) Maintainability & Operability

- Independent deployability of SBlocks (no global downtime for a local change).
  Why: Teams can ship fixes and features without scheduling system-wide outages.

- Consistent response shape for success and errors.
  Why: Uniform handling in the UI and Core reduces boilerplate and mistakes.

- Health and readiness signals for Core and SBlocks.
  Why: Operators and automated probes can detect issues early and route traffic away

from unhealthy parts.

- Coding and API conventions documented and applied.
  Why: Lowers onboarding time and prevents drift across services.

6) Integration & Interoperability

- All cross-service communication via the message queue with request/response pairing.
  Why: Decouples services, balances load more efficiently and allows safe retries without tight coupling.

- JSON for requests, responses, and messages.
  Why: Simple, widely supported, and already in use across our services.

7) Usability

- Clear, human-readable errors and timeouts that include a support token (correlation ID).
  Why: Users get clear, understandable messages; support can locate the exact trace fast.

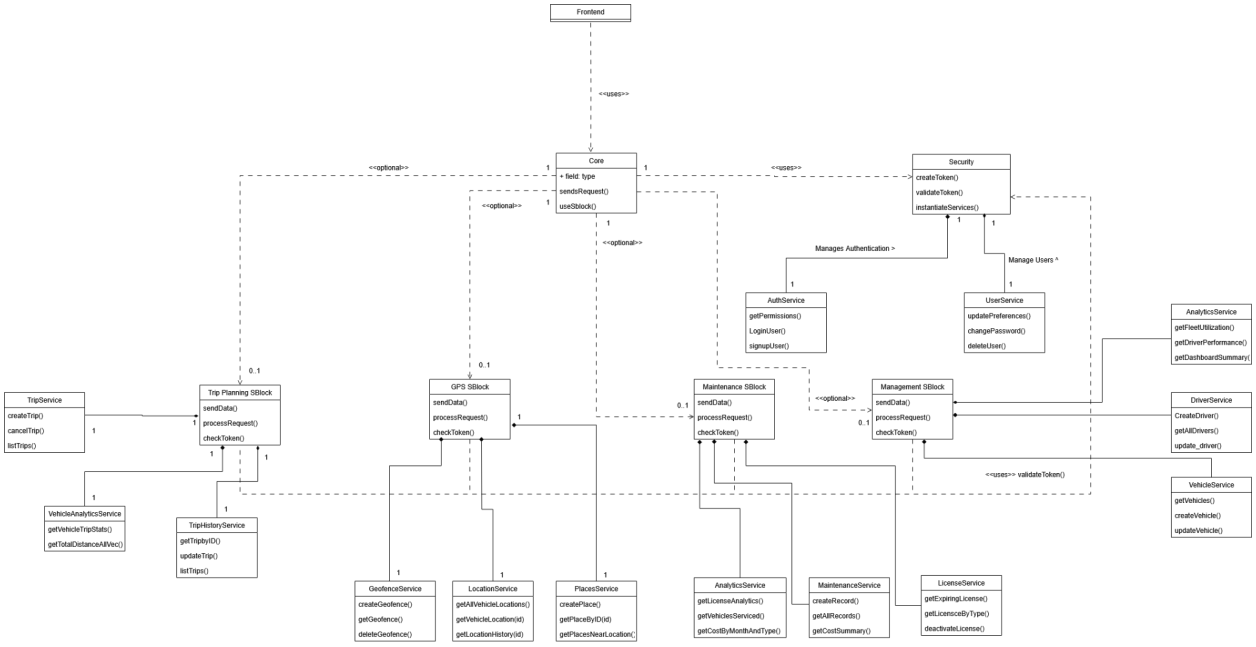
- Responsive web UI; remembers user preferences such as theme (light/dark).
  Why: Works well on common desktop and mobile browsers and feels tailored to the user.

- Customizable dashboard composition.
  Why: Different roles care about different numbers; letting users arrange tiles reduces hunting for information.

8) Capacity & Scalability

- Horizontal scaling: Core and SBlocks can scale out independently.
  Why: We add instances only where load exists; costs stay low while capacity grows.

5. Domain Model



6. Technology Choices

Backend: Python + FastAPI

Why?

- Productivity and clarity. Python's readability and library ecosystem let us move fast without cryptic code.
- Async by design. FastAPI's async support works well with our I/O-heavy flows (database, message queue), keeping latency low.
- Built-in API docs. Automatic OpenAPI/Swagger keeps the interfaces self-documenting and easier to test and maintain.

How it supports our quality targets

- Helps keep Core routing overhead low by handling concurrent requests efficiently.
- Encourages consistent request/response shapes, which improves operability and maintainability.

Trade-offs & mitigations

- Python isn't the fastest at raw CPU work; for heavier jobs we keep the work I/O-bound or move it to async/background consumers.

Message Queue: RabbitMQ

Why do we need a queue at all?

- Decoupling. Services don't need to be up at the same time. The queue sits in the middle so one slow or restarting service doesn't stall the rest.
- Smoothing traffic. Bursts get buffered. Producers can keep publishing while consumers catch up, which improves availability during spikes.
- Targeted routing. We can route messages to the right service (or a specific worker group) using simple keys, which fits our plug-in SBlock model.
- Retries without duplicates. Ack/nack semantics and correlation IDs let us retry safely and keep state consistent.

How it supports our quality targets

- Enables graceful degradation: if one SBlock is down, others continue.
- Keeps publish→consume latency low in normal conditions, while still giving us back-pressure and durability under load.

Trade-offs & mitigations

- A queue is another piece to run and monitor; we mitigate with health checks, sensible timeouts, and clear dead-letter practices.

Database: MongoDB

Why?

- Flexible documents. Our data changes over time (e.g., GPS snapshots, trip details, geofences, maintenance records). A flexible document model lets us evolve without heavy migrations.
- Natural fit for hierarchies. Vehicle → trip → telemetry snapshots and similar structures map cleanly to nested documents.
  Geospatial support. Storing points and shapes (locations, routes, geofences) and running geo queries is built-in and efficient.

How it supports our quality targets

- Handles high-volume writes (telemetry, logs) without complex schema changes.
- Keeps read latency low for dashboards by reading whole documents without cross-table joins.

Trade-offs & mitigations

- Cross-document transactions are limited compared to relational setups; we model aggregates so most writes are atomic per document, and enforce integrity with unique indexes and application checks.

Deployment: Docker & Docker-Compose

Why?

- Same environment everywhere. Containers make dev, test, and prod behave consistently.
- Service isolation. Each SBlock runs in its own container, which aligns with our modular design.
- Simple orchestration for our scale. Compose files define the full stack (Core, SBlocks, queue, database) for fast spin-up and repeatable CI runs.

How it supports our quality targets

- Faster rollouts and rollbacks → better uptime.
- Repeatable environments → better reliability and fewer “it works on my machine” surprises.

Trade-offs & mitigations

- Compose isn’t a full cluster orchestrator; we keep replicas sensible and monitor container health to avoid surprises.

Frontend: React

Why?

- Component model. Encourages small, testable pieces that we can reuse across screens.
- Ecosystem and skills. Easy to hire for, lots of stable libraries.
- Mobile path. Concepts carry over to React Native when we build the mobile app.

How it supports our quality targets

- Responsive UI and quick updates to match back-end events (e.g., GPS updates, notifications).
- Clear state management patterns support reliability and maintainability on the client.

Frontend Styling: Tailwind CSS

Why?

- Speed. Utility classes let us build and adjust layouts quickly.
- Consistency. Shared spacing, typography, and color tokens reduce design drift.
- Smaller CSS surface. Fewer one-off styles to maintain.

How it supports our quality targets

- Faster UI changes improve time-to-fix and keep the dashboard load focused on essentials.

Mapping Library: Leaflet (with React-Leaflet)

Why?

- Lightweight and open. No vendor lock-in; efficient for live markers and polylines.
- Feature coverage. Good support for geofences, routes, clustering, and custom overlays.

How it supports our quality targets

- Smooth rendering of live GPS positions helps us hit “feels real-time” expectations.
- Works well with our document+geo data model from MongoDB.

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

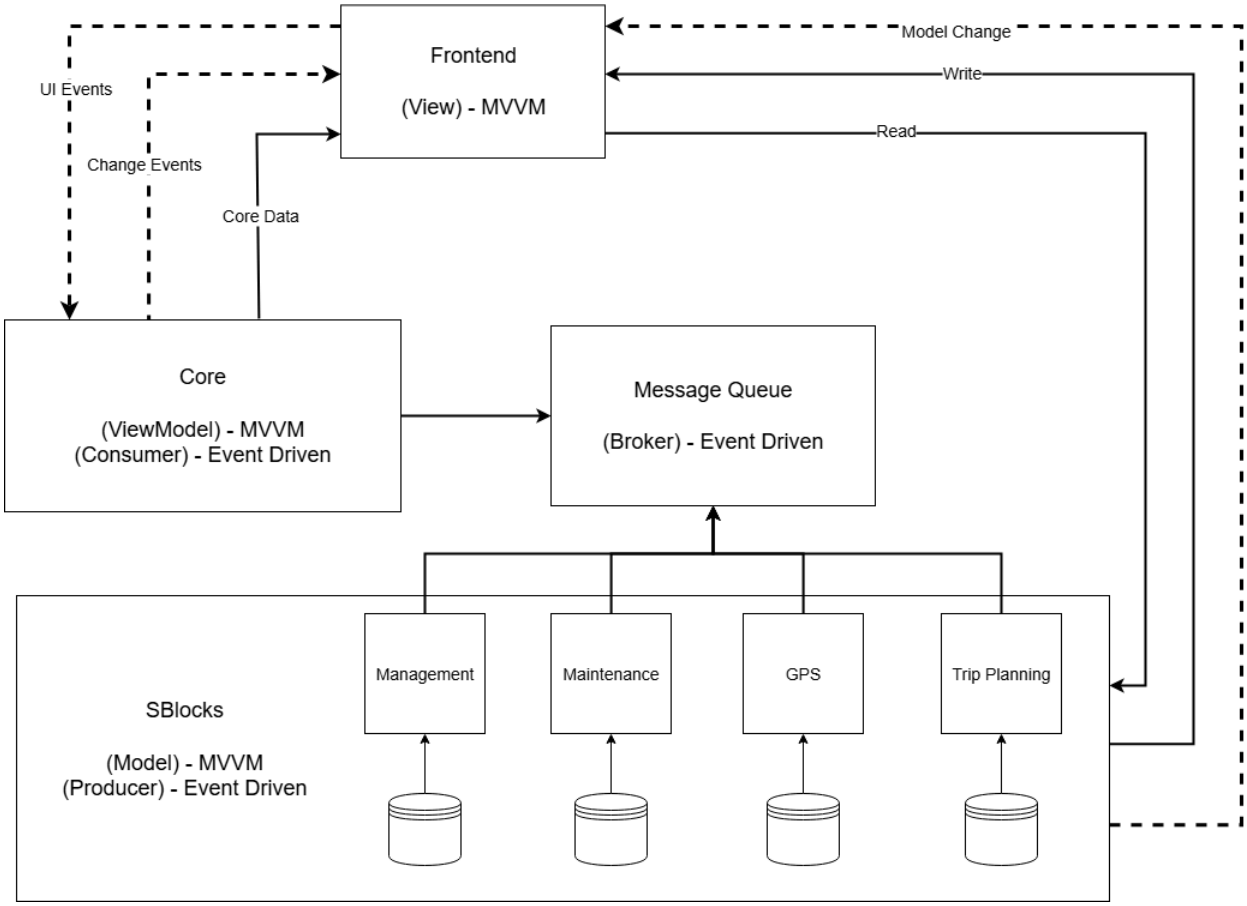
- **Separation of Concerns:**
    - **Model:** SBlocks handles the backend logic

    - **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.



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