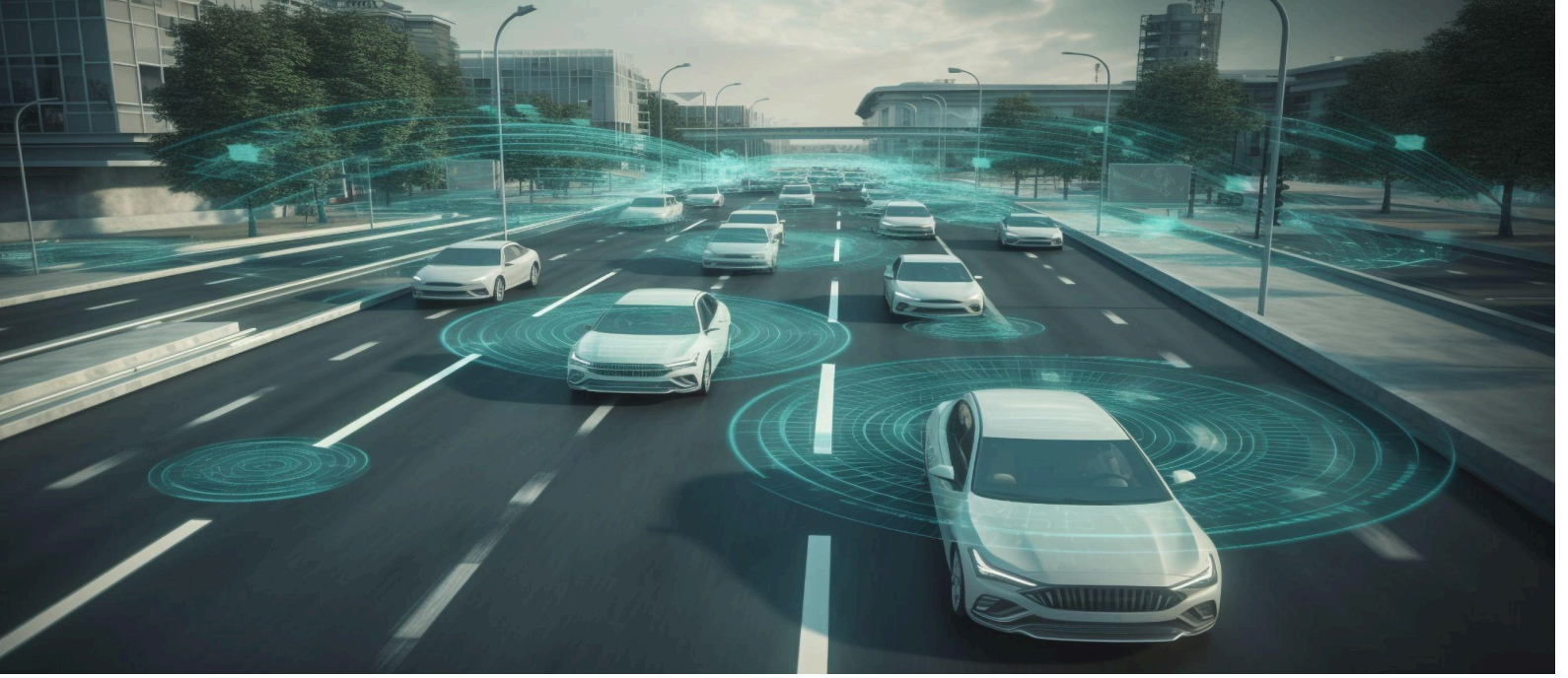


SAMFMS DEPLOYMENT STRATEGY

Firewall Five

1. Introduction

This document provides a comprehensive description of the deployment architecture and operational strategy for the **Safe and Modular Fleet Management System (SAMFMS)**. It serves as a detailed blueprint for deploying, managing, and scaling the system in a production environment. The architecture is meticulously designed to embody the core principles of a modern, micro-services application—specifically, high availability, elastic scalability, robust security, and effortless maintainability. This model directly translates the system's critical quality attributes into a concrete, operational reality, ensuring that the technical infrastructure fully supports the business objectives of reliable and efficient fleet management.

2. Deployment Objectives & Quality Attributes

The deployment strategy is the engineering embodiment of the system's non-functional requirements. It is designed not just to host the application but to guarantee its performance and integrity under load and over time.

- Scalability: The system must elastically accommodate growth in vehicles, users, and high-frequency telemetry data. The microservices architecture is important here, allowing for horizontal scaling of specific components. For instance, the GPS service can be scaled out independently of the User Management service during periods of heavy data transmission. Docker Compose's scaling features, combined with the stateless design of most services, make this operationalizable.
- Reliability & Availability: Containerization is a key enabler, providing immutable, consistent runtime environments from development to production, eliminating the "it works on my machine" problem. The fully automated deployment process drastically reduces the potential for human error, a leading cause of outages. Service health checks and automatic container restart policies, managed by Docker, further enhance resilience.
- Maintainability: The modular design decouples service lifecycles. A bug fix or feature update to one microservice can be built, tested, and deployed without necessitating a full system-wide redeployment or downtime. The integrated CI/CD pipeline provides a streamlined, auditable, and reversible process for applying changes, enabling rapid iteration and simplifying root cause analysis and rollback procedures.
- Security (Defense-in-Depth): Security is layered throughout the deployment:
    - Network Segmentation: All backend application and data tier components reside within a isolated private docker network segment, inaccessible from the public internet.
    - Secure Gateway: A reverse proxy acts as the single, hardened entry point, providing SSL/TLS termination, HTTP-to-HTTPS redirection, and basic rate limiting.

- Secure Artifact Chain: The use of a private Docker registry ensures that all deployed container images are vetted and stored internally, eliminating supply chain risks associated with public repositories and providing a clear audit trail of what is deployed in production.

3. Target Environment

SAMFMS will be deployed on a Production Server provided by the client.. The environment is considered a single, self-contained production environment.

- Operating System: Ubuntu Server 24.02 LTS
- Provisioning: The server will be manually provisioned with the necessary base software: Docker Engine, Docker Compose, and the SAMDS application.

4. Deployment Topology

The system follows a containerized microservices architecture. The application is decomposed into discrete, loosely coupled services, each running in its own Docker container. This promotes modularity, independent scaling, and easier development.
The topology consists of three main tiers:

1. Client Tier: Served by the CADDY reverse proxy, this delivers the React.js frontend assets to users' web browsers.
2. Application Tier (Microservices): Comprises multiple backend services (Core API, Security, and various service blocks) that contain the business logic. They communicate with each other via RESTful APIs.
3. Data Tier: A single MongoDB instance hosting multiple logical databases, one for each service block to ensure data encapsulation.

5. Tools & Technologies

- Containerization: Docker
- Orchestration: Docker Compose (for single-node management)
- CI/CD: GitHub Actions + Custom SAMDS Flask API
- Web Server / Reverse Proxy: NGINX
- Runtime: Node.js (for React build and services), Python (for FastAPI services and SAMDS)
- Database: MongoDB
- Artifact Storage: Private Docker Registry (hosted on the production server)

6. Deployment Process

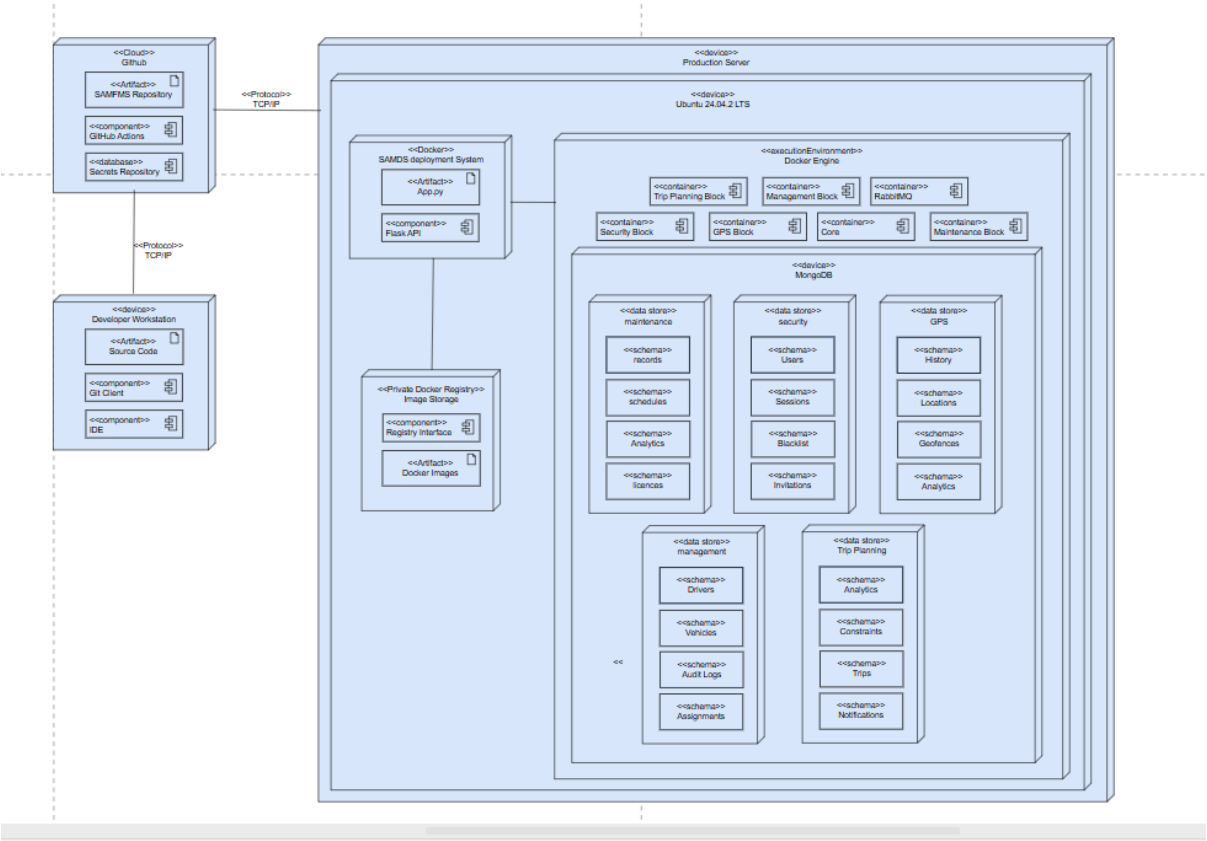
The deployment process is a fully automated, event-driven pipeline designed for speed, reliability, and consistency.

1. Trigger: A developer merges a approved feature branch or a hotfix via a Pull Request into the main branch of a service's GitHub repository.
2. Continuous Integration (CI - GitHub Actions):
    - The commit to main triggers the pre-configured GitHub Actions workflow.
    - The workflow checks out the code, sets up the required environment (e.g., Node.js, Python), and installs dependencies.

    - It executes the automated test suite (unit tests, integration tests). The deployment will abort at this stage if any test fails.
    - Upon successful testing, the workflow authenticates with the production server's SAMDS API using a secure secret (API key or token stored in GitHub Secrets) and sends a POST request to a specific endpoint (e.g., https://<server-ip>:5000/deploy/vehicle-service).
3. Continuous Deployment (CD - SAMDS API):
    - The SAMDS API listens on port 21049. It receives the POST request.
    - It performs authentication and validation (e.g., verifying the provided deployment token matches the one stored in its configuration).
    - Once authenticated, it shells out to execute a pre-written deployment script (e.g., a Bash script deploy.sh).
4. Script Execution (deploy.sh):
    - Pull Latest Code: The script navigates to the correct checked-out repository on the server and executes git pull origin main to fetch the latest code.
    - Build Docker Image: It runs docker build -t <service-name>:latest . to create a new Docker image from the updated codebase.
    - Tag and Push Image: The new image is tagged for the private registry (docker tag <service-name>:latest localhost:5000/<service-name>:latest) and pushed to the local private registry (docker push localhost:5000/<service-name>:latest).
    - Orchestrate Update: The script finally executes the core orchestration command: docker-compose up -d --no-deps <service-name>. This command is efficient; it pulls the latest image for the specific service from the registry (if not present) and recreates only that service's container with zero-downtime (if configured correctly), leaving all other running services completely unaffected.

7. Deployment Diagram

The diagram below illustrates the infrastructure setup and communication flow between the components.



8. Configuration and Data Management

Robust configuration and data management are critical for security and reliability.

- Environment Configuration: All service configuration is externalized using environment variables. These are defined in the docker-compose.yml file under the environment key for each service. This allows for the same container image to be run in different environments (dev, staging, prod) with different configurations without being rebuilt.
- Secrets Management: Highly sensitive data (e.g., database passwords, JWT signing secrets, API keys for third-party services) are not stored in docker-compose.yml. Instead, they are stored in a .env file on the host machine. This file is explicitly referenced by Docker Compose using the env_file directive. The .env file is excluded from version control via .gitignore and its permissions are strictly controlled on the host.
- Data Persistence: Stateful services, primarily MongoDB, are configured with Docker volumes (e.g., mongodb_data:/data/db). These volumes are stored in a managed directory on the host (/var/lib/docker/volumes/) by default. This ensures that all database data persists across container restarts, recreations, and even updates of the MongoDB image itself. Volume management (backups, snapshots) is a critical operational responsibility.