Swift Signals

Architectural Requirements

Contents

System Overview 3

Architectural Design Strategy 3

Justification: 3

Architectural Strategies 3

Primary Styles: 3

Justification: 5

Architectural Quality Requirements 5

Prioritized Quality Requirements: 5

Full Quality Requirements: 5

Architectural Design and Pattern 8

System Overview Diagram: 8

Design Patterns in Use: 8

Design justification: 9

Architectural Constraints 9

Technology Choices 11

Conclusion 12

System Overview

Swift Signals is a comprehensive traffic optimization system designed to analyse traffic patterns, simulate intersection performance, and optimize traffic light timing using machine learning algorithms. The system employs a microservices architecture to ensure scalability, maintainability, and independent deployment of core components including user management, traffic simulation, AI optimization, real-time control, metrics collection, and a web-based frontend interface.

Architectural Design Strategy

The architectural design strategy for Swift Signals combines **decomposition** based on **functionality** and **quality requirement-driven design**.

- **Decomposition**: The system is broken into discrete, independent services (microservices), aligning with functional boundaries such as simulation, optimization, user management, control, and metrics collection.
- **Quality Requirement Alignment**: The architecture is driven by critical quality attributes including performance, scalability, maintainability, and security.

Justification:

This hybrid approach ensures that the system can be scaled, maintained, and evolved independently while guaranteeing system-wide performance and reliability, essential for a critical, traffic-facing optimization platform.

Architectural Strategies

Swift Signals adopts a combination of proven architectural styles:

Primary Styles:

Swift Signals employs a microservices architecture to achieve modularity, scalability, and independent deployment capabilities. Each core function is implemented as a separate service with its own data store and deployment lifecycle.

- ***Microservices architecture:***
    - API Gateway: Central entry point providing request routing, authentication, and rate limiting
    - User Service: Handles authentication and user management, with JWT-based login/signup functionality

- Simulation Service: Traffic simulation engine with SUMO integration for intersection modelling
- Optimization Service: Applies machine learning or heuristic algorithms to generate optimized traffic signal strategies
- Control Service: Acts as a coordinator between the Simulation Service and the AI Service, ensuring smooth data flow and trigger sequencing
- Metrics Service: Collects and serves statistics about simulation runs and AI performance
- Frontend Service: React-based and Vite web interface for system interaction and visualization

- ***API Gateway Pattern***
  The API Gateway serves as the single entry point for all client requests, providing essential cross-cutting concerns including request routing, authentication, rate limiting, and HTTP-to-gRPC translation. This pattern simplifies client interactions by presenting a unified REST interface while allowing internal services to communicate efficiently via gRPC. The gateway implements load balancing for downstream services and provides centralized logging and monitoring capabilities.
- ***Database Per Service Pattern***
  Each microservice maintains its own database to ensure loose coupling and independent scaling. The User Service utilizes MongoDB for flexible user profile storage, while the Metrics Service employs PostgreSQL for time-series data. This pattern prevents database-level coupling between services and allows for technology choices optimized for specific data access patterns.
- ***Event-Driven Architecture***
  Services communicate through asynchronous events using gRPC streaming capabilities to reduce coupling and improve system resilience. The simulation service publishes events when simulations complete, triggering downstream processing in the AI and metrics services via efficient binary Protocol Buffer messages. This pattern enables loose coupling between services while maintaining high-performance communication channels and supports eventual consistency across the system.
- ***CQRS (Command Query Responsibility Segregation)***
  The metrics service implements CQRS to separate read and write operations for optimal performance. Historical traffic data writes are optimized for ingestion speed, while query operations are optimized for complex analytics and reporting. This separation allows for independent scaling of read and write workloads.

- ***Layered Architecture***
  Each microservice follows a layered architecture pattern, organizing code into distinct layers for improved maintainability and separation of concerns. The handler layer manages routing and input validation, serving as the interface between incoming requests and internal logic. The service layer contains the core business logic, orchestrating operations across domains. The database layer handles data persistence and encapsulates interactions with the underlying storage systems. This structure promotes clean abstractions, simplifies testing, and supports long-term scalability and developer on-boarding.

Justification:

These combined strategies align with the system's performance, scalability, and resilience requirements while supporting modular growth and independent deployment.

Architectural Quality Requirements

Prioritized Quality Requirements:

Priority	Quality Requirement	Specification (Testable Metric)
1	Performance	UI response < 2 seconds; simulations handle hundreds of vehicles; optimization inference < 30 seconds
2	Scalability	Horizontal scaling to support city-wide traffic networks; scalable databases for time-series traffic data
3	Reliability and Availability	99.5% system uptime; redundant critical services; automated failover mechanisms
4	Security	Role-based access control; encrypted data in transit and at rest; audit logging for sensitive actions
5	Maintainability	Independent service deployments; API versioning; minimum 80% test coverage; clear documentation

Full Quality Requirements:

Performance
The system must deliver high-performance capabilities to handle realistic traffic data processing and simulation workloads. Response times for web interface interactions

should not exceed 2 seconds under normal load conditions. The simulation service must be capable of processing complex intersection scenarios with hundreds of vehicles within acceptable timeframes. Traffic optimization should complete within 30 seconds for single intersection analysis. The system must support concurrent users and simultaneous simulation runs without degradation in performance.

Scalability

Swift Signals is designed to scale horizontally across multiple dimensions. The microservices architecture enables individual services to scale independently based on demand. The system must support scaling from single intersection analysis to city-wide traffic network optimization. Database systems should handle growing volumes of time-series traffic data and simulation results. The optimization service must accommodate training and inference workloads that increase with system adoption. Container orchestration through Kubernetes ensures efficient resource utilization and automatic scaling capabilities.

Reliability and Availability

The system requires high availability with a target uptime of 99.5% to support continuous traffic analysis and optimization. Critical services like the gateway service must implement redundancy and fail-over mechanisms. Data persistence layers must include backup and recovery procedures to prevent loss of historical traffic data and trained models. The API gateway should implement circuit breaker patterns to handle service failures gracefully. Health monitoring and alerting systems must provide proactive notification of system issues.

Security

Security is paramount given the system's potential integration with municipal traffic infrastructure. Authentication and authorization mechanisms must protect against unauthorized access to traffic functionality. API endpoints require rate limiting and input validation to prevent abuse. Sensitive configuration data and model parameters must be encrypted at rest and in transit. The system should implement audit logging for all configuration changes and control actions. Network security policies must isolate internal services from external access points.

Maintainability

The code base must support long-term maintenance and evolution through clear separation of concerns and comprehensive documentation. Each microservice should follow established coding standards and include unit and integration tests. Configuration management through YAML files enables environment-specific deployments without code changes. Service interfaces should be well-defined with comprehensive API documentation to facilitate integration and troubleshooting. API versioning strategies ensure backward compatibility during system updates.

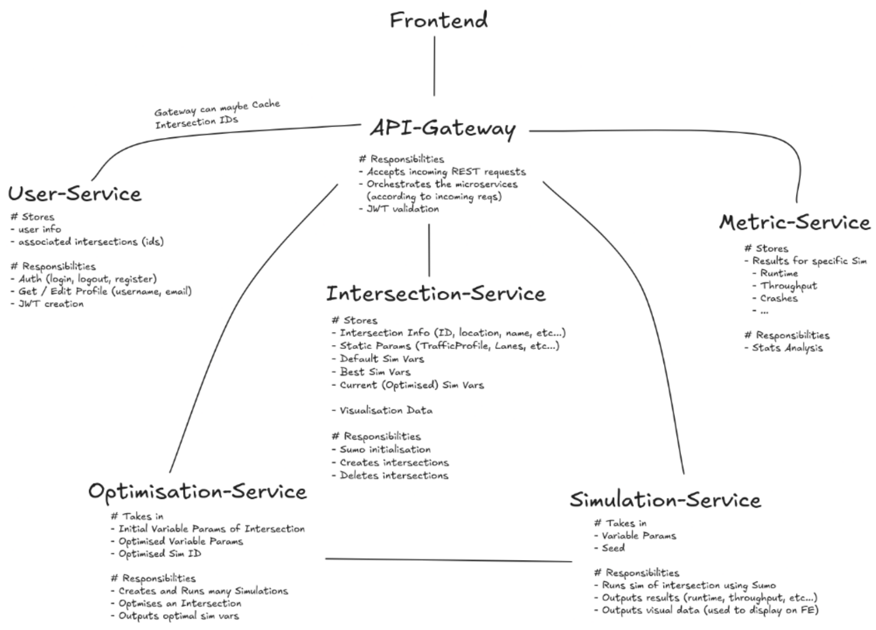
Extensibility
The architecture must accommodate future enhancements and integration with external traffic management systems. New intersection types and traffic patterns should be addable without requiring changes to core simulation logic. The optimization service should support multiple AI algorithms and model types. Plugin architectures enable third-party integrations for traffic data sources and optimization algorithms. The metrics service should accommodate new performance indicators and reporting requirements.

Usability
The web interface must provide an intuitive experience for traffic engineers and system administrators. Visualization components should clearly present complex traffic data and simulation results. Configuration workflows should guide users through intersection setup and optimization processes. Real-time simulation dashboards must display system status and traffic conditions effectively. Help documentation and user guides should support users of varying technical expertise levels.

Interoperability
The system must integrate with existing traffic management infrastructure and data sources. Standard protocols and data formats ensure compatibility with third-party traffic monitoring systems. REST APIs enable integration with municipal traffic control centers. Export capabilities allow sharing of optimization results with external planning tools. The system should support common traffic data formats and simulation standards.

Architectural Design and Pattern

System Overview Diagram:


Frontend
Gateway can maybe Cache Intersection IDs
API-Gateway
# Responsibilities
- Accepts incoming REST requests
- Orchestrates the microservices (according to incoming reqs)
- JWT validation
User-Service
# Stores
- user info
- associated intersections (ids)
# Responsibilities
- Auth (login, logout, register)
- Get / Edit Profile (username, email)
- JWT creation
Metric-Service
# Stores
- Results for specific Sim
- Runtime
- Throughput
- Crashes
- ...
# Responsibilities
- Stats Analysis
Intersection-Service
# Stores
- Intersection Info (ID, location, name, etc...)
- Static Params (TrafficProfile, Lanes, etc...)
- Default Sim Vars
- Best Sim Vars
- Current (Optimised) Sim Vars
- Visualisation Data
# Responsibilities
- Sumo initialisation
- Creates intersections
- Deletes intersections
Optimisation-Service
# Takes in
- Initial Variable Params of Intersection
- Optimised Variable Params
- Optimised Sim ID
# Responsibilities
- Creates and Runs many Simulations
- Optmises an Intersection
- Outputs optimal sim vars
Simulation-Service
# Takes in
- Variable Params
- Seed
# Responsibilities
- Runs sim of intersection using Sumo
- Outputs results (runtime, throughput, etc...)
- Outputs visual data (used to display on FE)

Design Patterns in Use:

- ***Factory Pattern***
  The simulation service employs the Factory pattern to create appropriate traffic models based on intersection types and optimization requirements. This pattern enables runtime selection of algorithms and supports extensibility for new optimization approaches.
- ***Observer Pattern***
  The gateway service implements the Observer pattern to notify interested parties when traffic light configurations change. This enables real-time updates to the frontend interface and logging of configuration changes for audit purposes.
- ***Singleton Pattern***
  Shared resources like database connection pools and configuration managers are implemented as singletons to ensure efficient resource utilization and consistent configuration across service instances.
- ***Decorator Pattern***
  The API gateway uses the Decorator pattern to add cross-cutting concerns like authentication, logging, and rate limiting to request processing pipelines. This pattern enables flexible composition of middleware components.
- ***Circuit Breaker Pattern***
  Inter-service gRPC communication implements the Circuit Breaker pattern to

handle failures gracefully and prevent cascade failures. This pattern improves system resilience by failing fast when downstream services are unavailable and provides automatic retry mechanisms with exponential backoff for transient failures.

Design justification:

These patterns enhance modularity, reusability, and resilience. The system's architecture ensures:

- Minimal coupling between services
- Scalable, fault-tolerant communication
- Extensibility for future intersection types, AI models, or integrations
- Compliance with security and operational constraints

Architectural Constraints

Technology Constraints

- Open Source Software Only: All system components must use open-source technologies and libraries to comply with project constraints
- Containerization: All services must be containerized using Docker for consistent deployment across environments
- Kubernetes Orchestration: Production deployments must use Kubernetes for container orchestration and service management
- gRPC Communication: Inter-service communication must use gRPC protocols with Protocol Buffer serialization for high-performance, type-safe communication between microservices

Infrastructure Constraints

- Cloud-Native Design: The system must be designed for cloud deployment with Southern Cross Solutions providing infrastructure resources
- Container Resource Limits: Each service must operate within defined CPU and memory constraints to ensure efficient resource utilization
- Network Security: Internal service communication must be secured through service mesh or VPN technologies
- Storage Limitations: Database storage must be planned for cost-effectiveness while meeting performance requirements

Development Constraints

- Independent Deployment: Each microservice must be independently deployable without affecting other services
- Automated Testing: All services must include comprehensive unit and integration tests with minimum 80% code coverage
- CI/CD Pipeline: Automated build, test, and deployment pipelines must be implemented using GitHub Actions
- Configuration Management: Environment-specific configurations must be externalized and managed through YAML files

Operational Constraints

- Monitoring and Logging: All services must implement structured logging and expose health check endpoints
- Data Retention: Historical traffic data must be retained according to specified policies while managing storage costs
- Backup and Recovery: Critical data must be backed up with defined recovery time objectives
- Security Compliance: System must implement security best practices including secret management and access controls

Performance Constraints

- Response Time: Web interface interactions must complete within 2 seconds under normal load
- Simulation Performance: Traffic simulations must complete within acceptable timeframes based on intersection complexity
- Concurrent Users: System must support multiple concurrent users without performance degradation

Integration Constraints

- Third-Party APIs: Integration with external traffic data sources must handle rate limits and API versioning
- SUMO Compatibility: Simulation service must maintain compatibility with SUMO traffic simulation software
- Standard Protocols: External integrations must use industry-standard protocols and data formats
- Backward Compatibility: API changes must maintain backward compatibility for existing integrations

Technology Choices

API Gateway

- **Options Considered:** Go, Node.js, Python
- **Choice:** Go
- **Justification:** High concurrency performance, strong gRPC support, minimal runtime footprint.

Simulation Service

- **Options Considered:** SUMO (Python/Go integration), custom-built simulation engine, VISSIM
- **Choice:** SUMO with Go/Python integration
- **Justification:** Industry-standard, open-source, realistic urban traffic simulation; extensibility with Go.

Optimization Service

- **Options Considered:** Custom Go-based AI, Python-based ML frameworks, Swarm Algorithms
- **Choice:** Swarm Algorithms in Go
- **Justification:** Efficient, adaptable optimization technique integrated within the Go-based microservices stack.

Databases

- **Options Considered:** MongoDB, PostgreSQL, Redis
- **Choices:** MongoDB (flexible user/config data); PostgreSQL (time-series metrics)
- **Justification:** Aligns with data access patterns; supports Database Per Service pattern.

Frontend

- **Options Considered:** React, Angular, Vue.js
- **Choice:** React with Vite, TailwindCSS
- **Justification:** Rapid development, responsive UI, strong ecosystem, aligned with maintainability goals.

Conclusion

The Swift Signals architectural requirements specification defines a robust, scalable, and maintainable system for traffic optimization. The microservices architecture provides the flexibility needed for independent development and deployment while addressing critical quality attributes including performance, security, and extensibility. The combination of established architectural and design patterns ensures a solid foundation for the system's evolution and long-term success in optimizing traffic flow and reducing congestion costs.