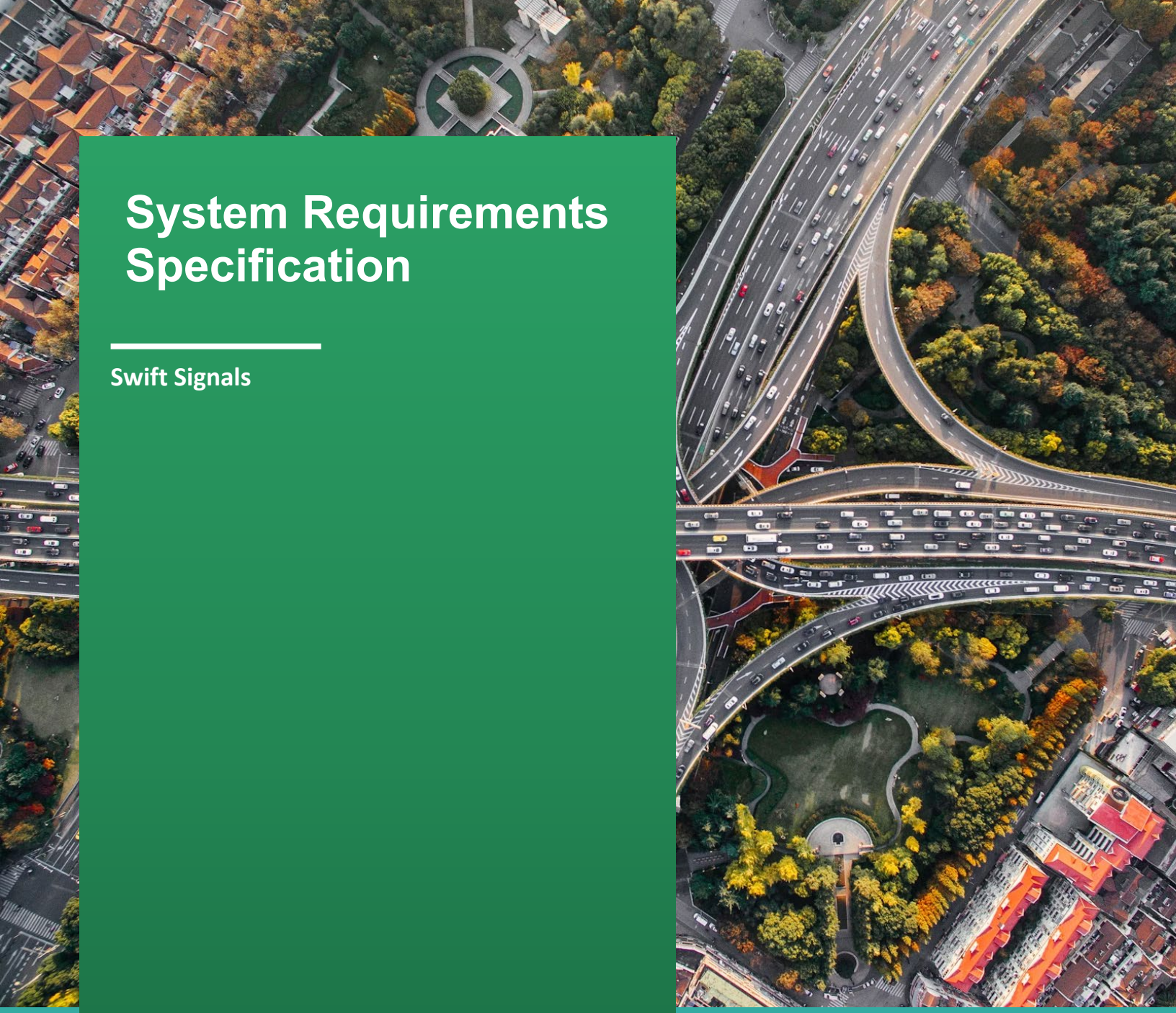


System Requirements Specification

Swift Signals

INSIDE INSIGHTS

For Southern Cross Solutions (Pty) Ltd.
insideinsights2025@gmail.com
Capstone Project
University of Pretoria

Swift Signals

Table of Contents

Overview

Swift Signals is a data-driven, simulation-powered traffic light optimization platform designed to address traffic congestion in urban environments. Developed in collaboration with Southern Cross Solutions, the project aims to equip municipal traffic departments with advanced tools to analyse intersection performance and improve traffic signal efficiency using machine learning.

Traffic congestion remains one of South Africa's most costly infrastructure challenges, with the South African Road Federation estimating annual productivity losses of approximately R1 billion. *Swift Signals* tackles this issue by providing a modular, web-based platform capable of simulating real-world intersection behaviour and dynamically optimizing traffic light phase configurations using historical traffic data. The system leverages modern software engineering principles—microservices, containerization, and continuous deployment pipelines—to deliver a scalable and maintainable solution.

Objectives

The *Swift Signals* platform is designed with the following objectives:

- **Simulate traffic flow at intersections** using historical and real-time data to model congestion patterns, vehicle throughput, and signal behaviour across different times of day.
- **Apply Swarm Optimization Algorithms** (e.g., Particle Swarm Optimization) to generate and evaluate multiple traffic light timing configurations, selecting the most efficient ones based on defined performance metrics such as wait time and throughput.
- **Leverage MongoDB** as the primary database to store and manage flexible, time-series traffic datasets, simulation outputs, and optimization results with scalability and efficiency.
- **Implement a microservices architecture** to modularize the system into distinct services such as simulation, optimization, API management, frontend interface, and data storage—each deployable and testable independently.

- **Develop a responsive web portal** using React.js and TailwindCSS, allowing users to:
    - Configure intersections and signal sequences.
    - Launch and monitor simulations.
    - View optimization reports and visual analytics.
    - Interact with traffic flow visualizations and system alerts.
- **Follow an Agile development process**, working in two-week sprints with continuous stakeholder feedback, bi-weekly deliverables, and regular sprint review sessions to ensure alignment with client expectations.
- **Design for future scalability**, including:
    - Support for optimizing multiple intersections simultaneously.
    - Integration of real-time traffic data feeds.
    - Expansion to more complex intersection models (e.g., turn-only lanes, variable lane counts).

User Characteristics

User Backgrounds

Swift Signals is built with two distinct user groups in mind, each defined by their background, expertise, and motivation for engaging with the platform:

Traffic Departments

These users represent official entities responsible for managing and improving road traffic infrastructure. They are typically employed by municipalities, city councils, or government transport agencies.

Characteristics:

- Operate in structured, regulatory environments.
- Possess domain knowledge in traffic engineering and urban planning.
- Prioritize reliability, accuracy, and measurable improvements in traffic flow.
- Often work within teams, requiring collaboration and role-based system access.
- Value data-driven decision-making and visual reporting to support operational planning.

Aspiring Traffic Engineers / Hobbyists

This group includes students, academic researchers, urban planning enthusiasts, and technology hobbyists with an interest in traffic optimization and simulation.

Characteristics:

- Motivated by curiosity, learning, or personal interest in intelligent transport systems.
- Come from varied technical backgrounds, ranging from engineering to data science.
- Tend to explore systems in a self-directed, experimental manner.
- Value intuitive interfaces, transparent models, and educational insights.
- Are not bound by institutional constraints and often use the system informally.

Use Cases

UC-001: User Authentication and Account Management

UC-001.1: User Registration

Actor:

- New user

Preconditions:

- User has valid email address

Main Success Scenario:

1. User navigates to sign-up page
2. System displays registration form with traffic light indicator
3. User enters username (3-32 characters)
4. User enters valid email address
5. User enters password
6. System validates all inputs in real-time
7. User submits form
8. System creates account and redirects to login page

Extensions:

- **E1:** Username already exists → System displays error
- **E2**: Invalid email format → System shows format requirements
- **E3:** Weak password → System enforces strength requirements

Postconditions:

- New user account created

UC-001.2: User Login

Actor:

- Registered user

Preconditions:

- User has valid account credentials

Main Success Scenario:

1. User navigates to login page
2. System displays login form with interactive traffic light
3. User enters email → Red and Yellow lights activate
4. User enters password → Green light activates
5. User submits form
6. System validates credentials
7. System generates JWT authentication token
8. System redirects user to dashboard

Extensions:

- **E1:** Invalid credentials → System displays error
- **E2:** Authentication fails → System shows retry option

Postconditions:

- User authenticated, session established

UC-001.3: Password Recovery

Actor:

- User with forgotten password

Preconditions:

- User has registered email address

Main Success Scenario:

1. User clicks "Forgot Password?" link
2. System displays password recovery modal
3. User enters registered email address
4. System sends reset email
5. User clicks link in email
6. System displays password reset form
7. User enters new password
8. System updates password

Postconditions:

- Password updated, user can login

UC-002: Intersection Management

UC-002.1: Create New Intersection

Actor:

- Authenticated user

Preconditions:

- User has valid authentication token

Main Success Scenario:

1. User navigates to Intersections page
2. User clicks "Add Intersection" button
3. System displays creation modal with three sections:

   General Information:

   - Intersection name input (required, unique)
   - Traffic density selection (Low/Medium/High radio buttons)

   Location Details:

   - Address/Cross streets input (required)
   - City selection (defaults to Pretoria)
   - Province selection (defaults to Gauteng)
   - *Simulation Parameters:*
   - Intersection type dropdown (Traffic Light)
   - Green light duration (seconds, 1-300 range)
   - Yellow light duration (seconds, 1-60 range)
   - Red light duration (seconds, 1-300 range)
   - Vehicle speed (km/h, 1-120 range)
   - Random seed (auto-generated, editable)
4. User configures all parameters
5. System validates input ranges and required fields
6. User clicks "Create Intersection"

7. System sends POST request to /intersections endpoint
8. System creates intersection in database
9. System refreshes intersections list

Extensions:

- **E1:** Validation failure → System highlights invalid fields
- **E2:** Name conflict → System shows error message

Postconditions:

- New intersection created

UC-002.2: Edit Existing Intersection

Actor:

- Authenticated user (owner or admin)

Preconditions:

- Intersection exists, and user has edit permissions

Main Success Scenario:

1. User clicks "Edit" button on intersection card
2. System fetches current intersection data
3. System populates edit modal with existing values
4. User modifies desired parameters
5. System validates changes
6. User clicks "Update Intersection"
7. System sends PATCH request to update intersection
8. System refreshes intersections list

Postconditions:

- Intersection updated

UC-002.3: Delete Intersection

Actor:

- Authenticated user (owner or admin)

Preconditions:

- Intersection exists and user has delete permissions

Main Success Scenario:

1. User clicks "Delete" button on intersection card
2. System displays confirmation modal
3. System shows warning about permanent deletion
4. User confirms deletion
5. System sends DELETE request
6. System removes intersection from database
7. System refreshes intersections list

Postconditions:

- Intersection permanently removed

UC-002.4: Search and Filter Intersections

Actor:

- Authenticated user

Preconditions:

- User has access to intersections list

Main Success Scenario:

1. User accesses search bar on Intersections page
2. User types search query (name)
3. System implements real-time search with 500ms debounce
4. System filters intersections based on query:
    - **Name search:** Case-insensitive partial match
    - **ID search:** Exact numeric match
5. System displays filtered results in real-time

Extensions:

- **E1:** No results found → System displays "No intersections found" message

Postconditions:

- User views filtered intersection list

UC-003: Traffic Simulation Execution

UC-003.1: Run Basic Simulation

Actor:

- Authenticated user with configured intersection

Preconditions:

- Intersection exists with valid parameters

Main Success Scenario:

1. User clicks "Simulate" button on intersection card
2. System navigates to Simulation Results page
3. System passes intersection data via React Router state
4. System displays simulation interface with:
    - Intersection details header
    - Simulation visualization charts
    - Run and Optimize buttons
5. User clicks "Run" button
6. System displays loading animation
7. System executes simulation via backend API
8. System displays results with performance metrics

Postconditions:

- Simulation results displayed

UC-003.2: Execute Optimization

Actor:

- Authenticated user with simulation results

Preconditions:

- Basic simulation has been executed

Main Success Scenario:

1. User clicks "Optimize" button
2. System displays optimization loading state

3. System runs optimization algorithm via **/intersections/{id}/optimise** endpoint
4. System displays optimized parameters alongside original
5. System updates intersection status to "optimised"
6. User can compare original vs. optimized configurations
7. System displays side-by-side performance comparison

Extensions:

- **E1:** No improvement found → System indicates current parameters are optimal

Postconditions:

- Optimized parameters generated

UC-003.3: Real-time Parameter Adjustment

Actor:

- Authenticated user during simulation

Preconditions:

- Simulation interface is active

Main Success Scenario:

1. User modifies speed or density parameters
2. System updates input fields in real-time
3. System validates parameter ranges (0-100)
4. System provides immediate visual feedback
5. User can adjust parameters multiple times
6. System enables "Run" button for new simulation

Postconditions:

- Parameters updated, ready for new simulation

UC-004: Results Analysis and Visualization

UC-004.1: Performance Metrics Display

Actor:

- Authenticated user with simulation results

Preconditions:

- Simulation has been executed successfully

Main Success Scenario:

1. System displays performance dashboard with:

   Primary Metrics:

   - Average speed (km/h)
   - Average travel time (seconds)
   - Average waiting time (seconds)
   - Total vehicles processed
   - Total travel time

   Secondary Metrics:

   - Maximum and minimum speeds
   - Vehicle count over time
   - Distance travelled per vehicle

2. System presents metrics in multiple formats
3. User can drill down into specific metrics

Postconditions:

- Performance analysis available

UC-004.2: Interactive Chart Visualization

Actor:

- Authenticated user viewing simulation results

Preconditions:

- Simulation data is available

Main Success Scenario:

1. System renders multiple chart types using Chart.js:

   Bar Charts:

   - Speed and density parameters
   - Original vs. optimized comparisons

   Line Charts:

   - Vehicle count over time
   - Average speed trends

Histograms:

- Speed distribution
- Wait time distribution

2. Charts feature interactive elements:

- Hover tooltips
- Click to expand to full-screen modal
- Responsive design

Postconditions:

- Interactive visualizations available

UC-005: Dashboard and Analytics

UC-005.1: Overview Dashboard Display

Actor:

- Authenticated user accessing main dashboard

Preconditions:

- User has valid authentication

Main Success Scenario:

1. System displays dashboard with:

Summary Cards:

- Total Intersections count
- Active Simulations count
- Optimization Runs count

Quick Action Buttons:

- New Intersection
- Run Simulation
- View Map

Recent Simulations Table:

- #, Intersection, Status, Actions columns
- Color-coded status badges
- View Details buttons

Traffic Density Chart:

- Traffic-Density-based line chart (Low to High)
- Gradient fill with smooth lines

Recent Intersections Section:

- Ranked by last use
- View All Button

2. System loads data from API endpoints
3. System provides loading states

Postconditions:

- Dashboard displayed with data

UC-005.2: Interactive Map Visualization

Actor:

- Authenticated user viewing map dashboard

Preconditions:

- User has intersections with geographic data

Main Success Scenario:

1. User clicks "View Map" button
2. System opens map modal with Leaflet map
3. System loads intersection data from API
4. System displays intersections on map with markers
5. User can interact with map (pan, zoom)
6. System provides map controls

Postconditions:

- Interactive map displayed

UC-006: User Management

UC-006.1: User Role Management

Actor:

- System administrator

Preconditions:

- User has admin privileges

Main Success Scenario:

1. Admin accesses Users page
2. System displays user management interface:

   User Table:

   - ID, Name, Email, Role, Last Login columns
   - Pagination (9 rows per page, responsive to 7 on smaller screens)
   - Edit and Delete action buttons

   Role Definitions:

   - **Admin:** Full system access
   - **User:** Simulation capabilities
3. Admin can modify user roles via edit interface
4. System enforces role-based access control

Postconditions:

- User roles managed

UC-006.2: User Account Administration

Actor:

- System administrator or user (self-management)

Preconditions:

- User has appropriate permissions

Main Success Scenario:

1. User accesses account management
2. System displays profile information and security settings
3. User can modify account details via edit modal
4. System validates all changes
5. System logs all account modifications

Postconditions:

- Account updated

UC-007: Geographic Integration

UC-007.1: Map-Based Intersection Selection

Actor:

- Authenticated user creating simulations

Preconditions:

- User has internet connection for map services

Main Success Scenario:

1. User accesses Map tab in simulation creation
2. System loads OpenStreetMap tiles via Leaflet
3. System displays interactive map with controls
4. User clicks on map at desired location
5. System processes click coordinates
6. System calls Overpass API to find nearest intersection
7. System snaps to nearest valid intersection
8. System displays intersection details
9. System adds intersection to selected list

Extensions:

- **E1:** No intersection found → System uses exact coordinates
- **E2:** API timeout → System shows retry option

Postconditions:

- Geographic locations captured

UC-007.2: Street Search and Intersection Finding

Actor:

- Authenticated user searching for locations

Preconditions:

- User has internet connection

Main Success Scenario:

1. User accesses Search tab in simulation creation
2. User enters location search query
3. System calls Nominatim API for geocoding
4. System displays search results with addresses
5. System processes results to find intersections
6. System calls Overpass API for detailed intersection data
7. System displays found intersections
8. User can select intersections from results

Postconditions:

- Intersections found and available for selection

UC-008: Help and Support Features

UC-008.1: Manage Help Menu

Actor:

- Authenticated user

Preconditions:

- User is viewing any page within the Swift Signals application

Main Success Scenario:

1. User clicks the "HELP" button on the side of the screen
2. System slides the Help Menu into view
3. User clicks the "X" button inside the Help Menu
4. System hides the Help Menu by sliding it out of view

Postconditions:

- **On success:** The visibility of the Help Menu is toggled (opened or closed)

UC-008.2: Use Swift Chat

Actor:

- Authenticated user

Preconditions:

- User must be logged in and Help Menu is open

Main Success Scenario:

1. User clicks the "Swift Chat" tab
2. System sends a "WELCOME" event to the backend and displays a welcome message from the chatbot
3. User types a question in the text input field
4. User presses Enter or clicks "Send"
5. System displays the user's message in the chat window and sends the query to the Dialogflow backend
6. System displays a "typing" indicator
7. System receives a response from the backend
8. System displays the bot's response in the chat window

Extensions:

- **E1:** User Interacts with Quick Reply
    1. Bot response includes quick reply buttons
    2. User clicks a quick reply
    3. System sends payload to backend as a new query
    4. Use case resumes from step 6
- **E2:** Chatbot Initiates Tutorial
    1. Backend detects intent start.tutorial
    2. Response includes tutorial action
    3. System triggers UC-008.4: Take Interactive Tutorial
- **E3:** Backend Connection Fails
    1. System request to backend fails (e.g., network/server issue)
    2. System displays error message ("Sorry, I'm having trouble connecting.")
    3. Use case ends

Postconditions:

- **On success**: User receives chatbot response and/or quick reply options
- **On failure:** System displays error message, no bot response

UC-008.3: Access General Help

Actor:

- Authenticated user

Preconditions:

- Help Menu is open

Main Success Scenario (Browse FAQs):

1. User clicks "General Help" tab
2. System displays "Tutorials" and "Frequently Asked Questions" sections
3. User clicks "Frequently Asked Questions" header
4. System expands to show FAQ list
5. User clicks a question
6. System expands to show the answer

Extensions:

- **E1:** User Launches a Tutorial
    1. User clicks "Tutorials" header
    2. System expands to show available tutorials
    3. User clicks a specific tutorial (e.g., "Dashboard Tutorial")
    4. System triggers UC-008.4: Take Interactive Tutorial

Postconditions:

- On success: User finds relevant help information or launches a tutorial

UC-008.4: Take Interactive Tutorial

Actor:

- Authenticated user

Preconditions:

- Tutorial has been requested by chatbot or General Help menu

Main Success Scenario:

1. System closes Help Menu
2. System displays an overlay and highlights the first UI element with a tooltip
3. User reads tooltip and clicks "Next"
4. System highlights next UI element in the sequence
5. Steps 3–4 repeat until final step
6. User clicks "Close" on final tooltip
7. System removes overlay and tooltips

Extensions:

- **E1:** Navigation Required
    1. System detects user not on correct page
    2. System displays confirmation popup ("Navigate to correct page?")
    3. User clicks "Yes"
    4. System navigates to required page
    5. Use case resumes from step 1
- **E2:** User Declines Navigation
    1. In step 3 of E1, user clicks "No"
    2. System closes popup and aborts tutorial
- **E3:** User Aborts Tutorial
    1. At any step, user clicks "Close" on tooltip
    2. System immediately removes overlay and tooltips

Postconditions:

- On success: User is guided through all tutorial steps
- On failure: Tutorial does not start or is aborted

User Stories

Login Screen

Login Form Functionality

- Given I am on the login page, I should see a form with fields for username and password.
- When I enter text in the username field, the input should update the username state.
- When I enter text in the password field, the input should update the password state.
- When I submit the form with both fields filled, the system should log the input values and redirect me to the dashboard page.
- When I submit the form with either field empty, the system should display a console message indicating that all fields must be filled.

Traffic Light Indicator

- Given I am on the login page, I should see a traffic light component with red, yellow, and green lights.
- When the username field is empty, the red and yellow lights should appear inactive (dimmed with a neutral border).
- When I enter text in the username field, the red and yellow lights should activate (bright with a glowing effect and distinct border).
- When I enter text in the password field, the green light should activate (bright with a glowing effect and distinct border).
- The traffic light should have a modern, visually appealing design with smooth transitions and shadow effects.

Forgot Password Modal

- Given I am on the login page, I should see a "Forgot Password?" link.
- When I click the "Forgot Password?" link, a modal should appear with an email input field and "Cancel" and "Send Reset Link" buttons.
- When I enter an email and click "Send Reset Link," the system should log the email and close the modal.

- When I click "Cancel," the modal should close without logging any data.
- The modal should be styled consistently with the login page and appear centered with a semi-transparent background.

Navigation

- Given I am on the login page, I should see a "Register Here" link.
- When I click the "Register Here" link, I should be redirected to the signup page.
- When I successfully submit the login form, I should be redirected to the dashboard page.

Visual and Accessibility Requirements

- The login page should have a responsive design, adapting to mobile and desktop screens.
- The page should include a logo and a welcome message ("Welcome to Swift Signals") at the top.
- Form inputs should have accessible labels (e.g., using sr-only for screen readers).
- The traffic light and form elements should use Tailwind CSS for consistent styling.
- The page should use a light colour scheme with a gradient background and shadow effects for a modern look.

Sign-Up Screen

Sign-Up Form Functionality

- Given I am on the sign-up page, I should see a form with fields for username, email, and password.
- When I enter text in the username field, the input should update the username state.
- When I enter text in the email field, the input should update the email state.
- When I enter text in the password field, the input should update the password state.
- When I submit the form with all fields filled, the system should log the input values and redirect me to the login page.

- When I submit the form with any field empty, the system should display a console message indicating that all fields must be filled.

Traffic Light Indicator

- Given I am on the sign-up page, I should see a traffic light component with red, yellow, and green lights.
- When the username field is empty, the red light should appear inactive (dimmed with a neutral border).
- When I enter text in the username field, the red light should activate (bright with a glowing effect and distinct border).
- When I enter text in the email field, the yellow light should activate (bright with a glowing effect and distinct border).
- When I enter text in the password field, the green light should activate (bright with a glowing effect and distinct border).
- The traffic light should have a modern, visually appealing design with smooth transitions and shadow effects.

Navigation

- Given I am on the sign-up page, I should see a "Login here" link.
- When I click the "Login here" link, I should be redirected to the login page.
- When I successfully submit the sign-up form, I should be redirected to the login page.

Visual and Accessibility Requirements

- The sign-up page should have a responsive design, adapting to mobile and desktop screens.
- The page should include a logo and a welcome message ("Welcome to Swift Signals") at the top.
- Form inputs should have accessible labels (e.g., using sr-only for screen readers).
- The traffic light and form elements should use Tailwind CSS for consistent styling.
- The page should use a light colour scheme with a gradient background and shadow effects for a modern look.

Dashboard Screen

Summary Cards

- Given I am on the dashboard page, I should see three summary cards displaying:
  - Total Intersections (e.g., "24") with a road icon.
  - Active Simulations (e.g., "8") with a play icon.
  - Optimization Runs (e.g., "156") with a chart line icon.
- Each card should have a consistent design with Tailwind CSS styling, including icons in distinct colours (blue, green, purple) and a clean layout.

Quick Actions

- Given I am on the dashboard page, I should see three quick action buttons:
  - "New Intersection" with a plus icon.
  - "Run Simulation" with a play icon.
  - "View Map" with a map icon.
- Each button should have a distinct background colour (indigo, green, purple) and hover effects.
- Clicking the buttons should be clickable but currently log no action (placeholder functionality).

Recent Simulations Table

- Given I am on the dashboard page, I should see a table listing recent simulations with columns for:
  - ID (e.g., "#1234").
  - Intersection (e.g., "Main St & 5th Ave").
  - Status (e.g., "Complete", "Running", "Failed").
  - Actions (e.g., "View Details" button).
- The status should display with color-coded badges (green for Complete, yellow for Running, red for Failed).
- The "View Details" button should be styled as a clickable link but currently log no action.

Traffic Volume Chart

- Given I am on the dashboard page, I should see a line chart displaying traffic volume over time (e.g., 6 AM to 10 AM).
- The chart should use Chart.js with a gradient fill (red-to-transparent), smooth line tension, and customized tooltips styled with Tailwind CSS colours.
- The chart should be responsive, with no grid lines on the x-axis, light grid lines on the y-axis, and formatted tick labels.
- The chart should clean up properly when the component unmounts to prevent memory leaks.

Top Intersections Section

- Given I am on the dashboard page, I should see a section listing the top intersections by vehicle volume (e.g., "Main St & 5th Ave: 15,000 vehicles").
- The section should include an "Avg Daily Volume" summary (e.g., "12,000 vehicles").
- The data should be displayed in a clean, bordered list with Tailwind CSS styling.

Visual and Accessibility Requirements

- The dashboard should include a Navbar at the top and a Footer at the bottom, consistent with the application's design.
- The page should have a responsive layout, adapting to mobile and desktop screens using a grid system for larger screens.
- The page should use a light background (gray-100) with dark mode support (gray-900).
- All text, including headings, should be styled with Tailwind CSS for consistency and readability.
- The chart and table should be accessible, with clear labels and colour contrasts suitable for screen readers and visual clarity.

Intersections Page

Search Functionality

- Given I am on the intersections page, I should see a search bar with a placeholder text "Search by Name or ID...".

- When I type in the search bar, the list of intersections should dynamically filter to show only those matching the search query by name (case-insensitive) or ID.
- The search bar should include a search icon (from Lucide React) positioned on the right side.
- The search bar should be styled with Tailwind CSS, including a border, rounded edges, and a focus ring in red.

Intersection List

- Given I am on the intersections page, I should see a list of intersection cards, each displaying:
  - ID (e.g., "1").
  - Name (e.g., "Main St & 1st Ave").
  - Location (e.g., "Pretoria CBD").
  - Lanes (e.g., "4-way, 2 lanes each").
- Each card should be rendered using the IntersectionCard component and support three actions: Simulate, Edit, and Delete.
- Clicking the Simulate, Edit, or Delete buttons should log a corresponding message to the console (e.g., "Simulate 1", "Edit 1", "Delete 1").
- If no intersections match the search query, the list should display no cards.

Add Intersection Button

- Given I am on the intersections page, I should see an "Add Intersection" button styled in red (bg-red-700, hover:bg-red-800).
- Clicking the button should log "Add Intersection" to the console (placeholder functionality).
- The button should be positioned to the right of the search bar in the top bar.

Visual and Accessibility Requirements

- The intersections page should include a Navbar at the top, consistent with the application's design.
- The page should have a responsive layout, adapting to mobile and desktop screens, with a maximum width of 6xl (Tailwind CSS) and padding.
- The page should use a light background (gray-100) with a scrollable container for the intersection list.

- The search bar should have an accessible placeholder and be usable with screen readers.
- Intersection cards should be spaced vertically and styled with Tailwind CSS for consistency and readability.
- The search icon should be clearly visible and properly aligned within the search bar.

Simulations Page

Simulation and Optimization Tables

- Given I am on the simulations page, I should see two tables: one for "Recent Simulations" and one for "Recent Optimizations".
- Each table should display columns for:
    - Simulation ID (e.g., "SIM001").
    - Intersection (e.g., "Main St & 1st Ave").
    - Avg Wait Time (e.g., "45.2").
    - Throughput (e.g., "1200").
    - Graph (bar chart comparing wait time and throughput).
    - Status (e.g., "Complete", "Running", "Failed").
    - Actions (View Results and Delete buttons).
- The status should display with color-coded badges (green for Complete, yellow for Running, red for Failed).
- The bar chart should use Chart.js with gradient fills (green for wait time, blue for throughput), no axis labels, and modern styling (rounded bars, tooltips).
- Clicking "View Results" should trigger an alert with the simulation ID (e.g., "Viewing results for simulation SIM001").
- Clicking "Delete" should trigger an alert with the simulation ID (e.g., "Deleting simulation SIM001").
- Each table should support pagination with 4 rows per page, displaying "Prev", page numbers, and "Next" buttons styled with Tailwind CSS gradients.

Filtering by Intersection

- Given I am on the simulations page, each table should have a dropdown to filter by intersection or "All Intersections".

- Selecting an intersection should filter the table to show only simulations/optimizations for that intersection and reset the page to 0.
- The dropdown should list all unique intersections from the respective table's data, plus "All Intersections".

New Simulation/Optimization Modal

- Given I am on the simulations page, I should see a "New Simulation" button for the simulations table.
- Clicking "New Simulation" should open a modal titled "New Simulation" (or "New Optimization" for future functionality).
- The modal should include:
    - A text input for "Simulation Name".
    - A text area for "Simulation Description".
    - An intersection selection section with three tabs: List, Search, and Map.
- List Tab: Displays a dropdown of available intersections (from both tables' data). Selecting an intersection adds it to the selected intersections list.
- Search Tab: Allows typing a location and clicking "Add" to include it in the selected intersections list.
- Map Tab: Displays an interactive Leaflet map centered at [-26.2041, 28.0473] (South Africa) with zoom level 6. Clicking the map places a marker, logs coordinates, and adds them as an intersection to the selected list.
- Selected intersections should appear as removable pills (with a cross button) styled with Tailwind CSS.
- Clicking "Create" should validate that a name and at least one intersection are provided, log the data, clear the form, and navigate to /simulation-results with the data.
- Clicking "Cancel" or the close button (cross) should close the modal without saving.
- The modal should have a semi-transparent background, dark mode support, and Tailwind CSS styling.

Visual and Accessibility Requirements

- The page should include a Navbar at the top and a Footer at the bottom, consistent with the application's design.

- The page should have a responsive layout, using a two-column grid for medium and larger screens, and a single-column layout for mobile.
- The page should use a light background (gray-100) with dark mode support (gray-900).
- Tables, inputs, and buttons should be styled with Tailwind CSS for consistency and readability.
- The map should use OpenStreetMap tiles with proper attribution.
- All inputs and interactive elements (dropdowns, buttons, map) should be accessible, with clear labels and sufficient contrast for screen readers and visual clarity.
- Font Awesome icons (eye for View, trash for Delete) should be used in the table action buttons.

Simulation Results Page

Simulation Data Display

- Given I am on the simulation results page, I should see the simulation's name, description, and intersections passed from the previous page via react-router-dom state.
- The simulation name should be displayed as a large, bold heading with a teal-to-emerald gradient.
- The description should be a paragraph in gray text.
- Intersections should appear as clickable pills with a teal border and hover effects, styled using Tailwind CSS.
- If no simulation data is available (e.g., no state passed), a "Loading..." message should display in the center.

Simulation Visualization Section

- Given I am on the simulation results page, I should see a "Simulation Visualization" section with:
    - A bar chart displaying speed and density parameters (default: speed=50, density=30) using Chart.js.
    - The chart should have green bars (#34D399, #10B981), no legend, and customized tooltips with a dark background.

- Axes should display "Parameter" (x) and "Value" (y, 0-100 range) with light grid lines and bold labels.
- The section should include input fields for adjusting speed and density (number inputs, 0-100 range).
- Input fields should be styled with Tailwind CSS, using a gray background and teal focus ring.
- A "Run" button should toggle a "Running..." state for 2 seconds when clicked, disabling itself during this period.
- An "Optimize" button should adjust optimized parameters (increase speed by 20 up to 100, decrease density by 10 down to 0).
- Clicking the chart should open a full-screen modal displaying the same chart with a title ("Simulation Visualization").

Optimized Visualization Section

- Given I am on the simulation results page, I should see an "Optimized Visualization" section with:
  - A bar chart displaying optimized speed and density parameters (default: speed=70, density=20) using Chart.js.
  - The chart should have blue bars (#3B82F6, #2563EB), no legend, and customized tooltips with a dark background.
  - Axes should match the simulation chart's configuration.
  - The section should include input fields for adjusting optimized speed and density (number inputs, 0-100 range).
  - Input fields should match the simulation section’s styling.
  - A "Run" button should toggle a "Running..." state for 2 seconds when clicked, disabling itself during this period.
  - Clicking the chart should open a full-screen modal displaying the same chart with a title ("Optimized Visualization").

Full-Screen Chart Modal

- Given I click a chart in either visualization section, a full-screen modal should appear with:
  - The selected chart (simulation or optimized) rendered at full size using Chart.js.

- A title matching the chart type ("Simulation Visualization" or "Optimized Visualization").
- A close button (cross) in the top-right corner to exit the modal.
- The modal should have a semi-transparent black background and be centered with a maximum width of 7xl.
- Charts should clean up properly when the modal closes to prevent memory leaks.

Visual and Accessibility Requirements

- The page should include a Navbar at the top and a dynamic Footer at the bottom, consistent with the application's design.
- The page should have a responsive layout, using a two-column grid for medium and larger screens and a single-column layout for mobile.
- The page should use a dark gradient background (gray-900 to black) with light text for readability.
- Charts, inputs, and buttons should be styled with Tailwind CSS, including gradients, shadows, and hover effects.
- Input fields should have accessible labels and be constrained to valid ranges (0-100).
- All interactive elements (buttons, charts, inputs) should have sufficient contrast and be usable with screen readers.

Users Page

Users Table Display

- Given I am on the users page, I should see a table displaying user data with columns for:
  - ID (e.g., "1").
  - Name (e.g., "John Doe").
  - Email (e.g., "email@email.com").
  - Role (e.g., "Admin", "Engineer", "Viewer").
  - Last Login (e.g., "2025-05-13 09:00").
- The table should be rendered using the UsersTable component, passing the current page's user data and handlers for Edit and Delete actions.

- Clicking the Edit button for a user should log "Edit user [ID]" to the console.
- Clicking the Delete button for a user should log "Delete user [ID]" to the console.

Pagination

- Given I am on the users page, I should see pagination controls below the table, including:
  - A "Previous" button with a left arrow icon, disabled on the first page.
  - Numbered page buttons, showing up to 5 pages (first, last, and up to 3 around the current page).
  - An ellipsis ("...") for skipped page numbers when there are more than 5 pages.
  - A "Next" button with a right arrow icon, disabled on the last page.
  - Clicking a page number should display the corresponding user data slice (9 or 7 rows per page, depending on screen size).
  - Clicking "Previous" or "Next" should navigate to the adjacent page, updating the table data.
  - Pagination buttons should be styled with Tailwind CSS, with the active page highlighted in blue and hover effects on others.

Responsive Row Adjustments

- Given I am on the users page, the table should display 9 rows per page by default.
- When the screen size is 1400px or smaller and 800px or shorter, the table should display 7 rows per page, resetting to the first page.
- The row adjustment should occur dynamically when the window size changes, using a media query listener.
- The total number of pages should update automatically based on the number of users and rows per page.

Visual and Accessibility Requirements

- The page should include a Navbar at the top, consistent with the application’s design.
- The page should have a responsive layout, with a maximum width of 6xl (Tailwind CSS) and padding for content.

- The page should use a light background (gray-100) for a clean appearance.
- The table and pagination controls should be styled with Tailwind CSS for consistency and readability.
- Pagination buttons should have accessible labels (e.g., "Previous page", "Next page") and use SVG icons for arrows.
- The table should be accessible, with proper semantic structure and support for screen readers (handled by the UsersTable component).

Authentication

- As a **user**,
  I want to **register, log in, and manage my account**,
  so that I can **use the system securely and privately**.
- As a **user**,
  I want to **reset my password** if I forget it,
  so that I can **regain access** to my account.

Simulation Creation

- As a **user**,
  I want to **create a new simulation** with basic intersection details,
  so that I can **model traffic behaviour**.
- As a **user**,
  I want to **manually configure** traffic flow probabilities and light timings,
  so that I can **experiment with different traffic patterns**.
- As a **traffic department user**,
  I want to **upload real-world traffic data**,
  so that the **system can create simulations based on actual conditions**.

Simulation Insights

- As a **user**,
  I want to **view performance metrics** for each simulation,
  so that I can **understand and evaluate the traffic behaviour**.

Optimization

- As a **user**,
  I want to **optimize my simulation's timing**,
  so that I can **observe what timing's best suit my simulation**
- **As a user,**
  I want to compare the original and optimized configurations,
  so that I can see how the optimization has improved traffic flow.

Functional Requirements

Requirements

R1: Welcome page
R1.1: Must provide users with information about the application and give them an idea of what the application is all about.
R1.2: Must allow the user to access either the login or signup page.

R2: Login/Signup
R2.1: Must allow users to login their existing account details
R2.2: If a user does not have an account, allow user to create account
R2.3: Once registered, allow users to login
R2.4: Allow users to switch between light/dark mode.

R3: Dashboard page
R3.1: Include navbar in all pages that allows users to access various pages such as the simulations page.
R3.2: Show statistics such as total intersections, active simulations, traffic volume and top intersections.
R3.3: Show recent simulations as well as their status and allow users to view details about them.
R3.4: Allow users to insert a new intersection, run simulation and view the map.
R3.5: Allow users to log-out by clicking the logout button.
R3.6: Allow users to access the help menu in all accessible pages, which also has an AI chatbot. It is used for direct access to documentation, support and other helpful resources.

R4: Intersections page
R4.1: Allow users to view all listed intersections, which include information about the intersection
R4.2: Allow users to search for a specific intersection
R4.3: Allow users to add a new intersection
R4.4: Allow users to be able to simulate, edit and delete a specific intersection

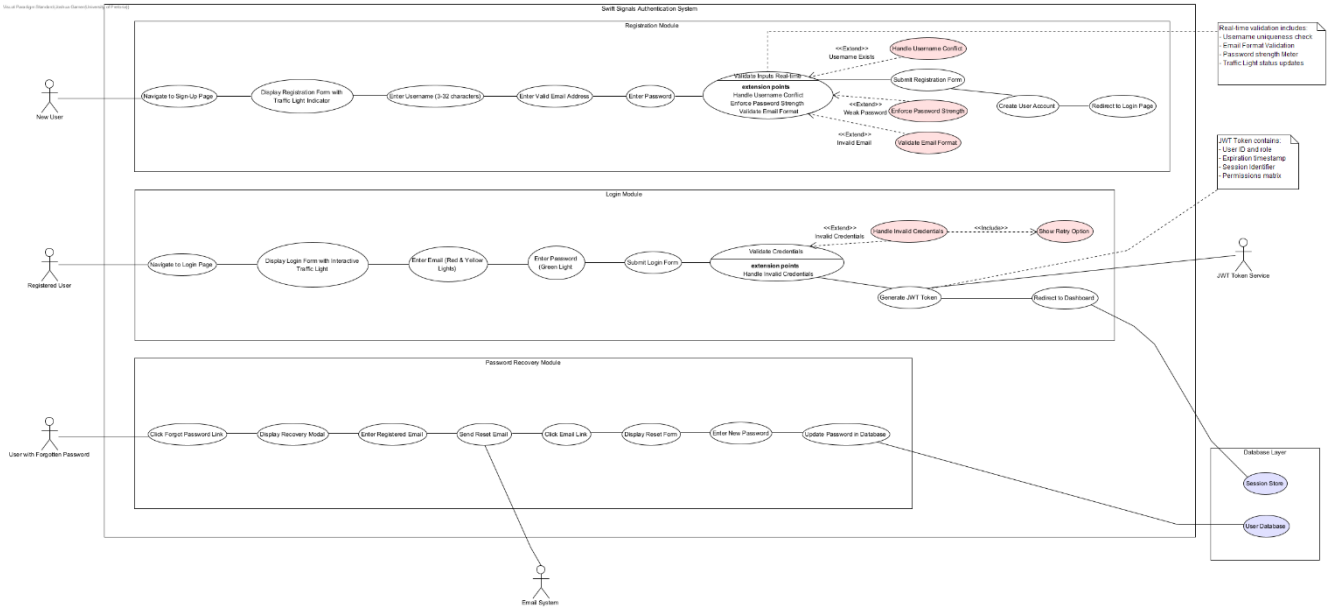
R5: Simulation page
R5.1: Users should be able to view recent simulations and optimizations with information about them such as average wait time and throughput.
R5.2: Allow users to create a new simulation. Users must be able to name it and select a specific intersection to create a new simulation.
R5.3: Allow users to select a specific intersection through typing or the use of the map for filtering purposes.
R5.4: Allow users to navigate through simulation and optimization tables if there are more entries that can fit on one page by clicking on 'Next', 'Prev' or a page number buttons.
R5.5: After simulation has been created, users should be provided with a detailed, interactive environment to analyse and compare the initial simulation with an optimized version.
R5.6: Users should be able to view simulation details
R5.7: Must include a **View Rendering** and **Optimize** buttons
R5.8: Users should be able to view a side-by-side rendering of the Simulation and Optimised Simulation, where on the left is the original and the right is the optimized rendering.

R6: Users Page
R6.1: Users must be able to view all listed users, with their information included.
R6.2: Allow users to be able to navigate through table pages.
R6.3: Users must be able to edit and delete information.

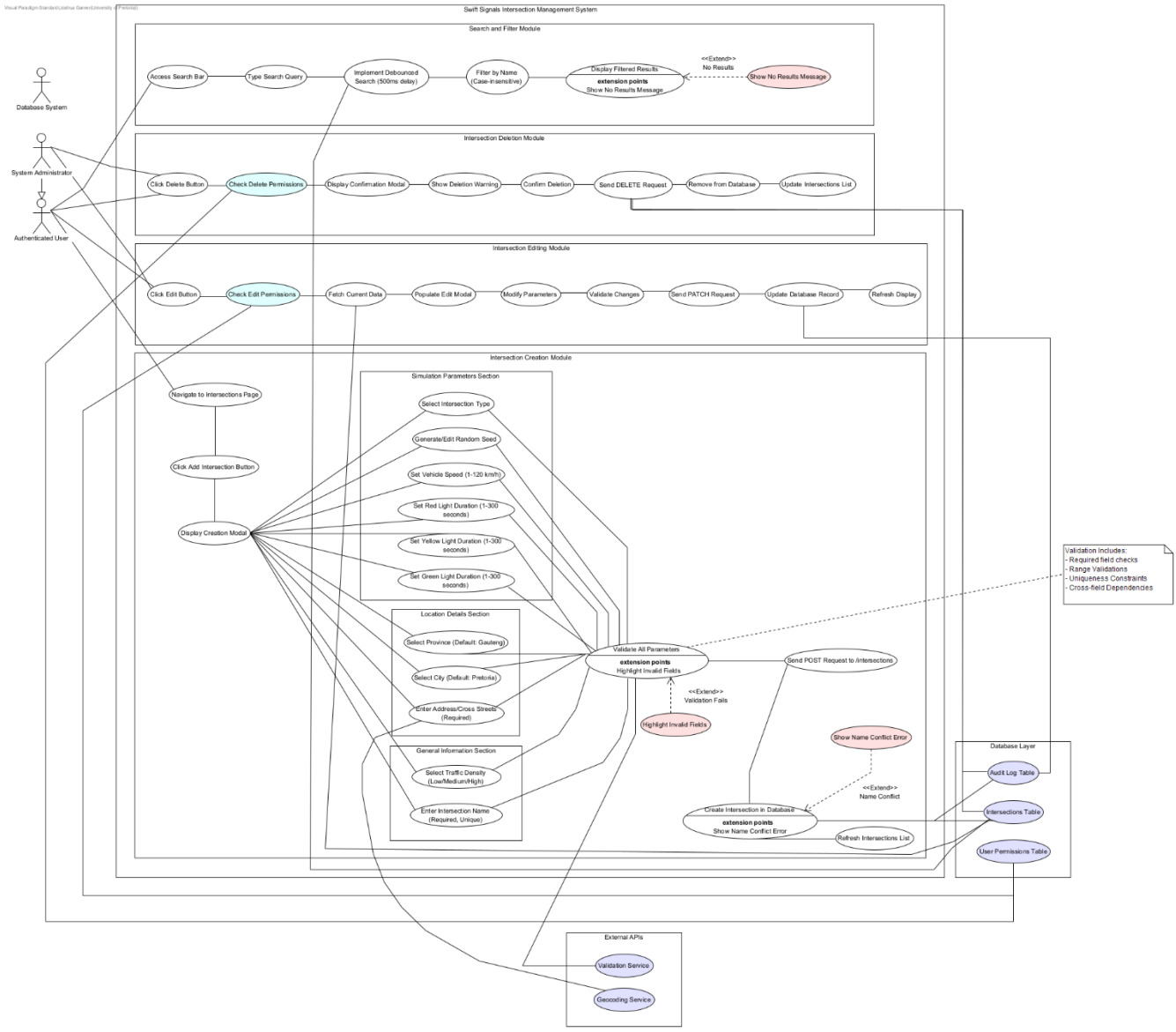
R7: Help Menu
R7.1: Allow users to access the help menu in all accessible pages
R7.2: Allow users to access either General Help or Swift Chat (AI Chatbot)
R7.3: Allow users to select default help options
R7.4: Allow users to input a command in the AI Chatbot
R7.5: Must include interactive tutorials

Use Case Diagrams

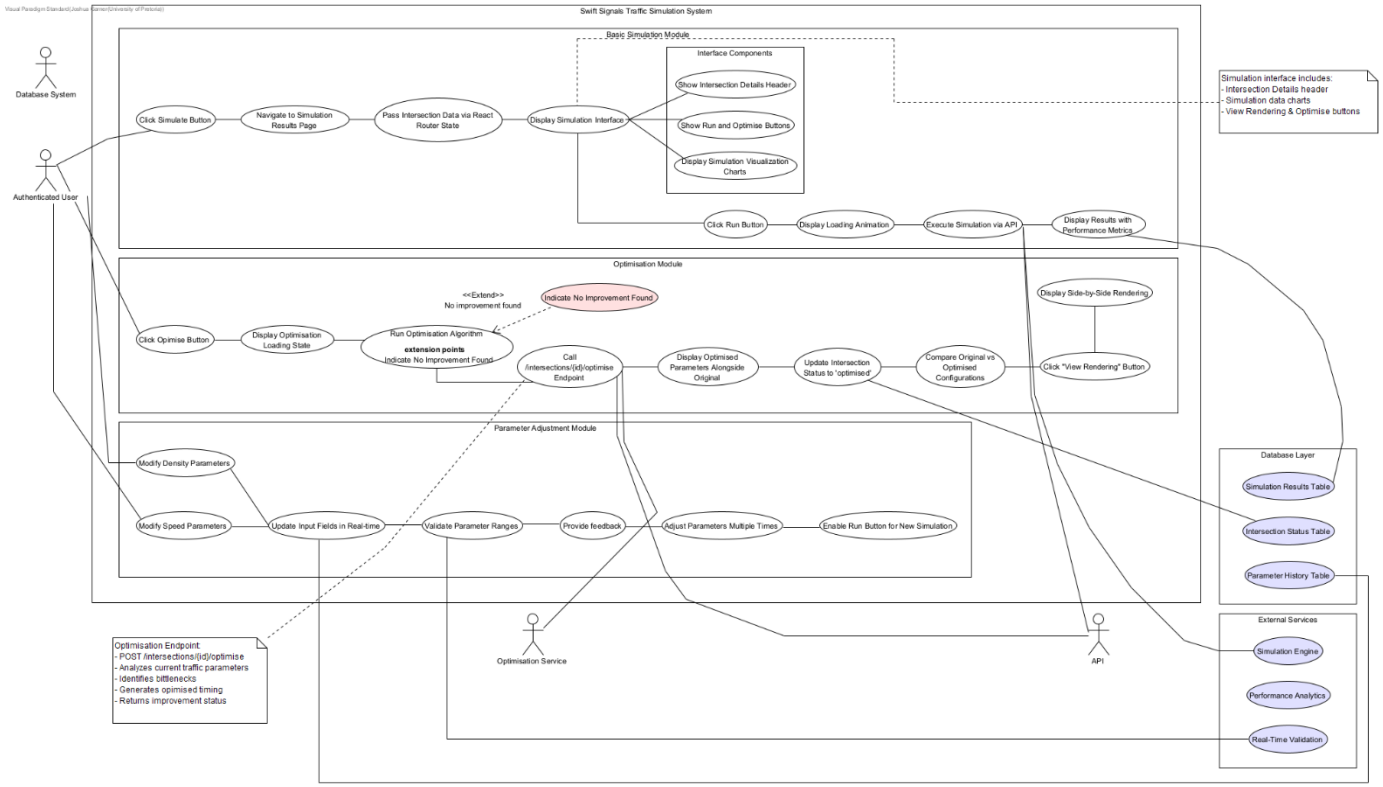
User Authentication



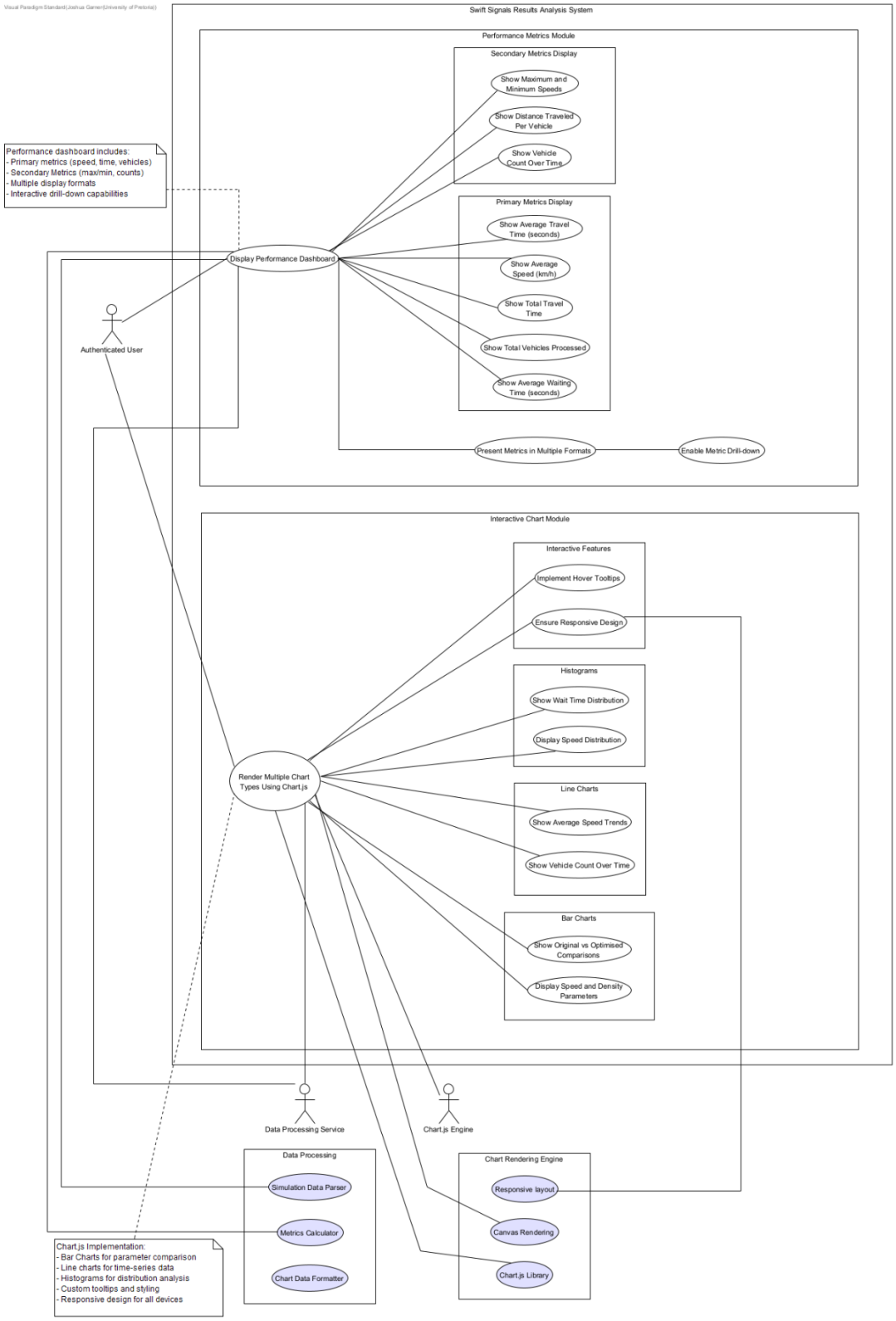
Intersection Management



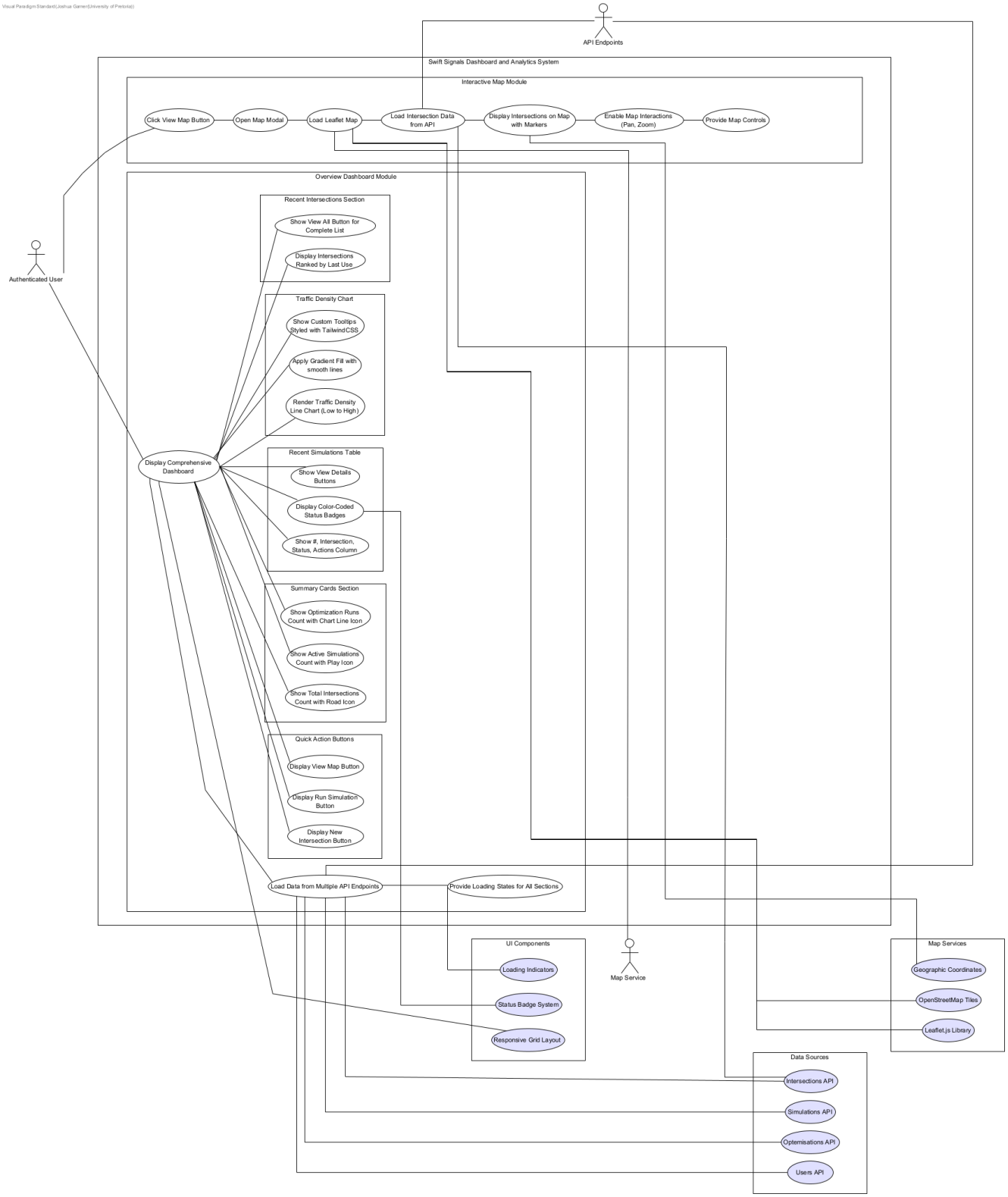
Simulation & Optimisation Execution



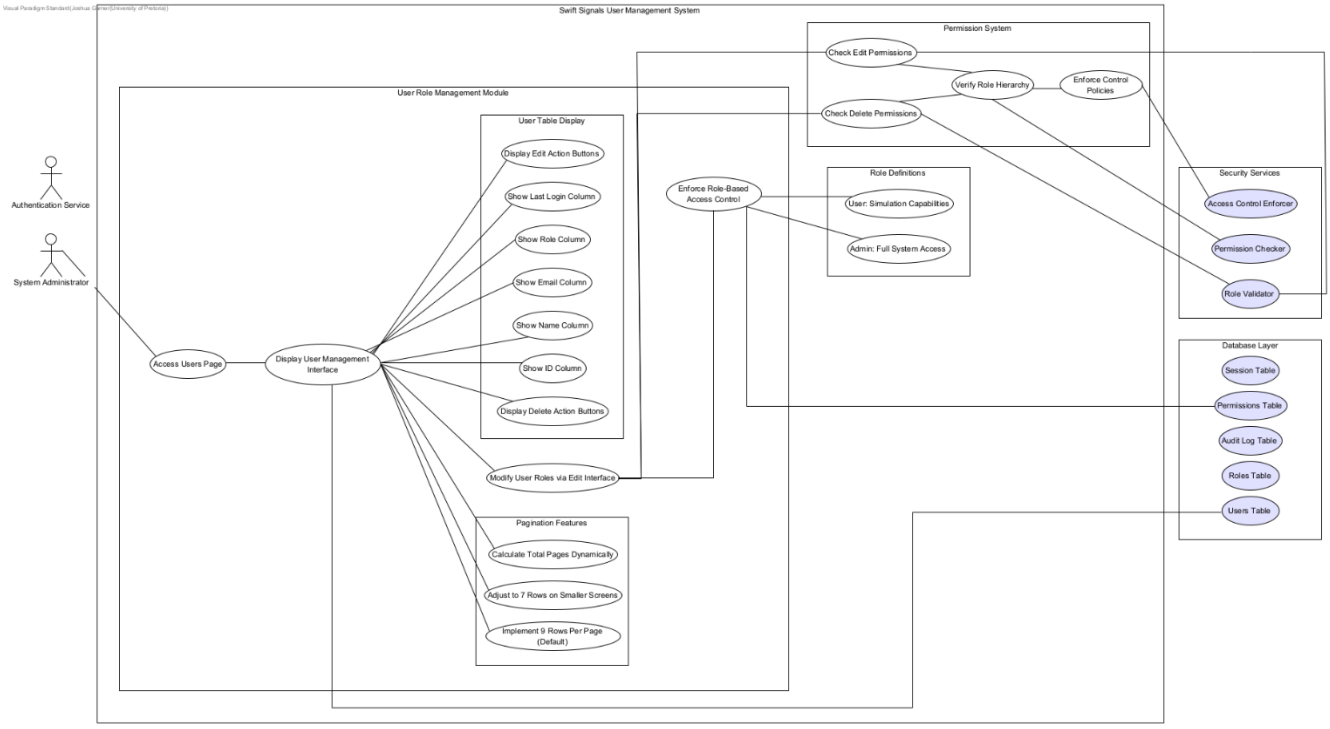
Results Analysis & Visualization



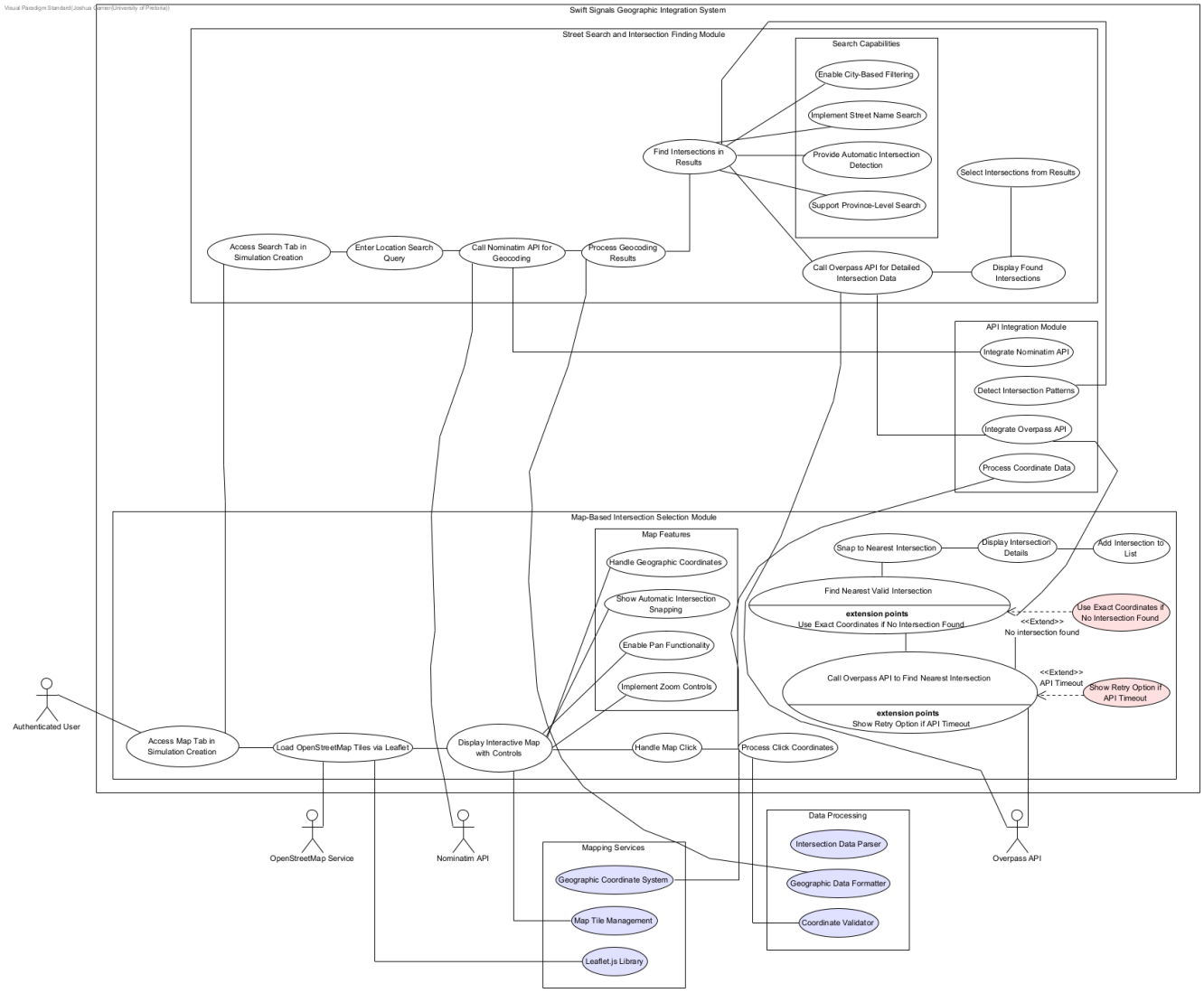
Dashboard & Analytics



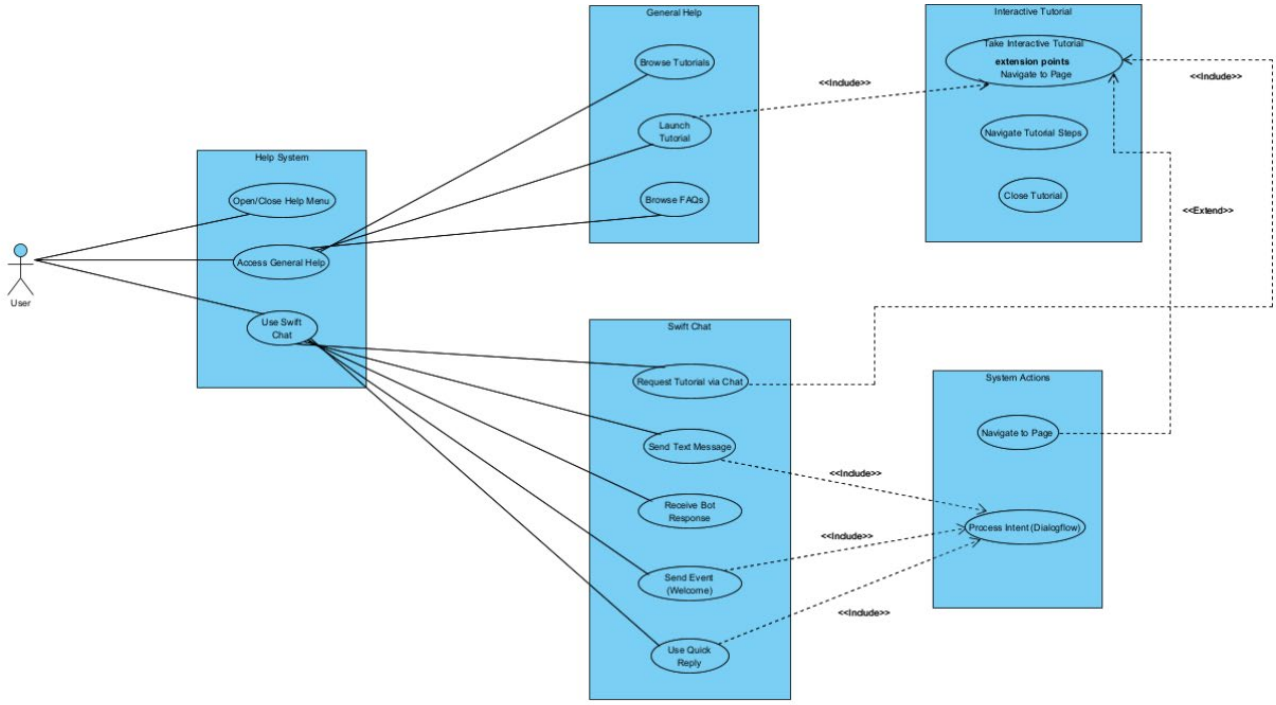
User Management



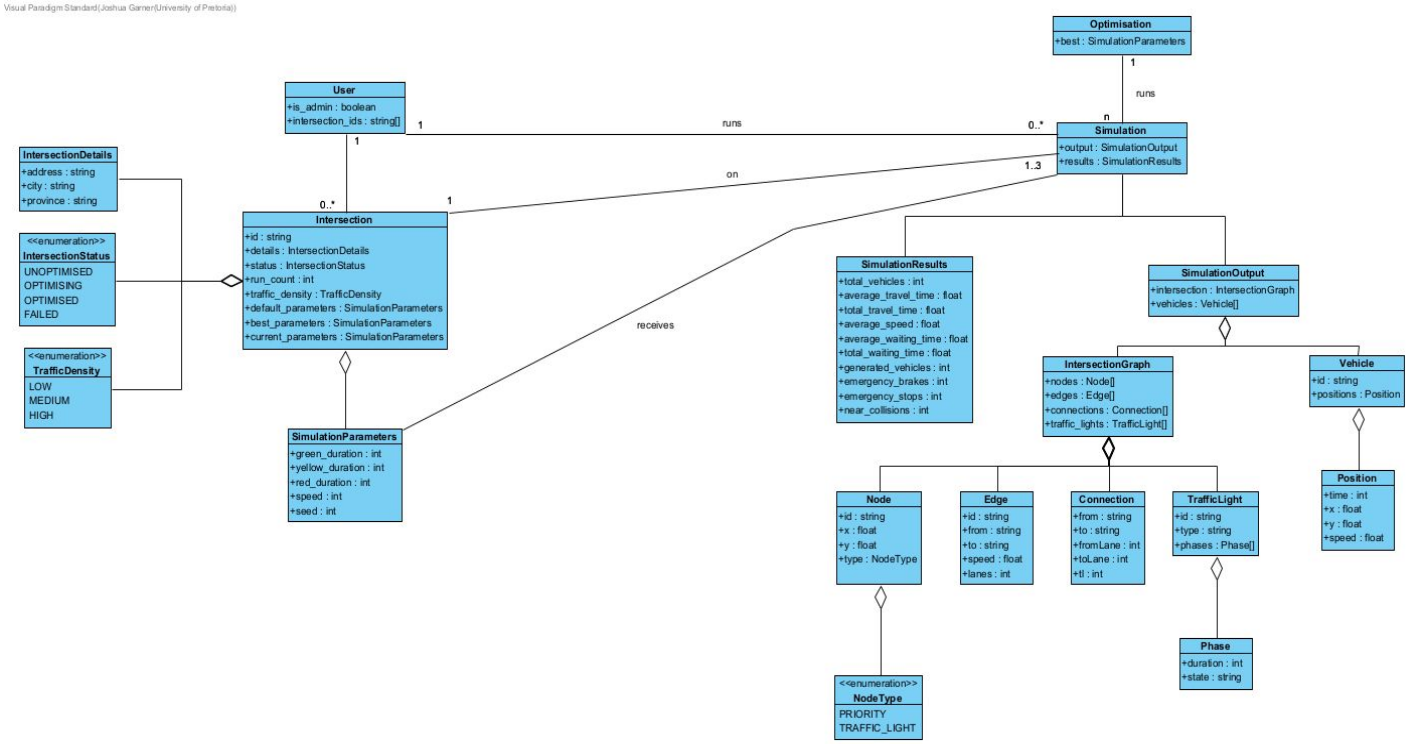
Geographic Integration



Help Menu & Chatbot



Domain Model



This domain model represents our vision for a comprehensive and optimized traffic simulation system, centring around the Simulation entity which integrates traffic parameters, optimization settings, and results data.

Architectural Requirements & Design

Non-Functional Requirements

Quality Requirements

Swift Signals is designed to meet stringent quality standards, ensuring the platform remains extensible, maintainable, reliable, secure, and easy to use in real-world traffic management scenarios. The system is a comprehensive traffic optimization system designed to analyze traffic patterns, simulate intersection performance, and optimize traffic light timing using artificial intelligence algorithms.

Core System Requirements

Usability

Why it matters?

- **The system needs to accommodate for both technical experts and non-technical users.**

Key Expectations:

- **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.**
- **Help documentation and user guides should support users of varying technical expertise levels.**

Extensibility

Why it matters?

- **The system must be capable of seamless integration with existing traffic department infrastructure and real-world data sources.**
- **Municipal traffic systems vary widely in their technologies and data formats, requiring flexible integration capabilities.**
- **The system should adapt to different regional traffic management requirements and standards.**

Key Expectations:

- **The system must provide standardized APIs for integration with external traffic management systems.**
- **Support for multiple data formats and protocols commonly used by traffic departments (XML, JSON, REST, SOAP, etc.).**
- **Plugin architecture to support custom integrations without modifying core system functionality.**

Maintainability

Why it matters?

- **The system is expected to be used indefinitely by traffic departments.**

Key Expectations:

- **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.**

Reliability

Why it matters?

- **The system requires high availability to support continuous traffic analysis and optimization.**

Key Expectations:

- **Target uptime of 99.5% ensures continuous system availability.**
- **Health monitoring and alerting systems must provide proactive notification of system issues.**

Basic System Requirements

Scalability

Why it matters?

- **There are close to 5,500 traffic officers in South Africa and the system needs to be able to handle the necessary volume of users.**

- **The system is expected to accommodate future growth of traffic departments.**
- **Database systems should handle growing volumes of time-series traffic data and simulation results.**
- **The optimization component of the system must accommodate training and inference workloads that increase with system adoption.**

Key Expectations:

- **The service should be able to handle 10,000 traffic officer requests per second which is close to double the number of traffic officers**

Performance

Why it matters?

- **The system must deliver high-performance capabilities to handle: -> realistic traffic data processing -> effective optimisations of simulations -> effective rendering of simulations.**
- **The system needs to be able to perform as efficiently as possible to reduce user waiting time.**
- **The system must support concurrent users and simultaneous simulation runs without degradation in performance.**

Key Expectations:

- **Web interface responses must complete within 2 seconds under normal load.**
- **Traffic simulations must process hundreds of vehicles per intersection within acceptable timeframes.**
- **Artificial Inteligence optimisations for traffic lights must complete within 30 seconds per intersection.**

Security

Why it matters?

- **Due to municipal integrations the system must enforce strict security standards.**

Key Expectations:

- **The system must make use of role based access control**
- **The system must make use of rate limiting and input validation to prevent abuse.**

Architectural Strategies

Usability

- **Visualization components should clearly present complex traffic data and simulation results.**
- **Configuration workflows should guide users through intersection setup and optimization processes.**
- **Help documentation and user guides should support users of varying technical expertise levels.**
- **Responsiveness**

Extensibility

- **Standardized API interfaces with comprehensive documentation**
- **Plugin architecture with well-defined extension points**
- **Protocol abstraction layers to support multiple communication standards**
- **Environment-specific configuration management**

Maintainability

- **Each service should follow established coding standards and include unit and integration tests.**
- **Configuration management to enable 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.**

Reliability

- **Continuous monitoring with Elastic Search**

- **Redundancy and failover mechanisms to protect critical services.**
- **Circuit breaker patterns in communications to prevent cascading failures.**
- **Real-time health monitoring and alerts.**
- **Backup and recovery strategies safeguard historical traffic and simulation data.**

Scalability

- **Independent scaling of system services based on demand**
- **Efficient data storage and indexing**
- **Resource reuse**

Performance

- **Modularisation of system**
- **Efficient containerization**
- **Low-latency communication**

Security

- **TLS encryption**
- **Tokens**
- **Certification**
- **Rate limiting**
- **Input validation**
- **Audit logging**

Architectural Patterns

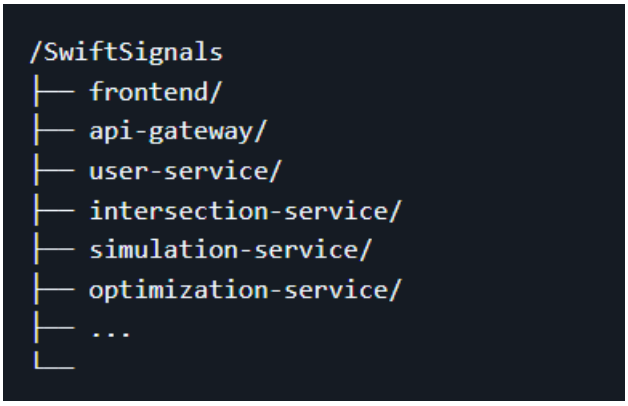
The system makes use of the following patterns:

- **Microservices**
- **Layered Architecture**
- **API-Gateway Pattern**
- **Repository Pattern**

Microservices

- **Enables easy integration with real traffic departments as all that would be needed is to add another microservice for external system connectivity**
- **Supports modularization of the system by breaking down complex traffic management functionality into smaller, focused services**
- **Enables independent scaling of system services based on demand, allowing traffic optimization components to scale separately from data processing services**
- **Provides redundancy and failover mechanisms through service isolation, preventing single points of failure**
- **Facilitates resource reuse across different traffic management functions while maintaining service boundaries**
- **Enables each service to follow established coding standards and include comprehensive testing independently**

How has it been integrated into Swift Signals?



Layered Architecture

- **Supports extensibility by providing clear integration points at each layer for external traffic management systems**
- **Enforces clear separation of concerns between presentation, business logic, and data layers for improved maintainability**
- **Provides structured security boundaries where each layer can implement appropriate access controls and validation**

- **Supports configuration management by isolating environment-specific settings within designated layers**
- **Enables well-defined service interfaces through standardized layer communication protocols**
- **Facilitates comprehensive API documentation by organizing functionality into logical layer groupings**

How has it been integrated into Swift Signals?

High-Level Layers

```
/SwiftSignals
├── ...
├── frontend/                  // Presentation Layer
│                              ------------------------
├── api-gateway/               // Access Layer
│                              ------------------------
├── user-service/              //
│                              //
├── intersection-service/      //
│                              // Service Layer
├── simulation-service/        //
│                              //
├── optimisation-service/      //
└──                            --------------------------
```

Low-Level Layers

```
/SwiftSignals
├── ...
├── user-service
│   ├── cmd/
│   ├── internal
│   │   ├── handler/    // Handler Layer
│   │   ├── service/    // Service Layer
│   │   ├── db/         // DB Layer
│   │   └── ...
│   └── ...
│
├── intersection-service
│   ├── cmd/
│   ├── internal
│   │   ├── handler/    // Handler Layer
│   │   ├── service/    // Service Layer
│   │   ├── db/         // DB Layer
│   │   └── ...
│   └── ...
├── ...
└──
```

API-Gateway Pattern

- **Enables extensibility by providing a centralized integration point for external traffic department systems and protocols**
- **Implements centralized rate limiting and input validation to prevent system abuse and protect backend services**
- **Provides single entry point for authentication and authorization, supporting role-based access control requirements**
- **Enables circuit breaker patterns in communications to prevent cascading failures across the system**
- **Supports API versioning strategies to ensure backward compatibility during system updates**
- **Centralizes audit logging for security monitoring and compliance tracking**

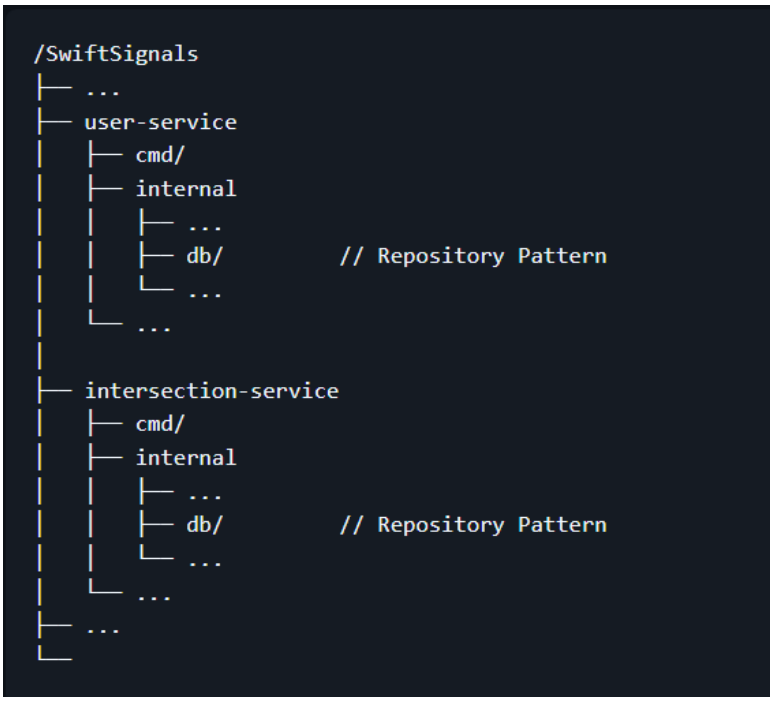
How has it been integrated into Swift Signals?

```
/SwiftSignals
├── ...
├── api-gateway
│   ├── cmd/
│   ├── internal
│   │   ├── ...
│   │   ├── middleware/     // Common endpoint functionality
│   │   ├── util/           // Uniform tools
│   │   └── model/          // Convert to consistent types
│   └── swagger/            // Consistent API documentation
├── ...
└──
```

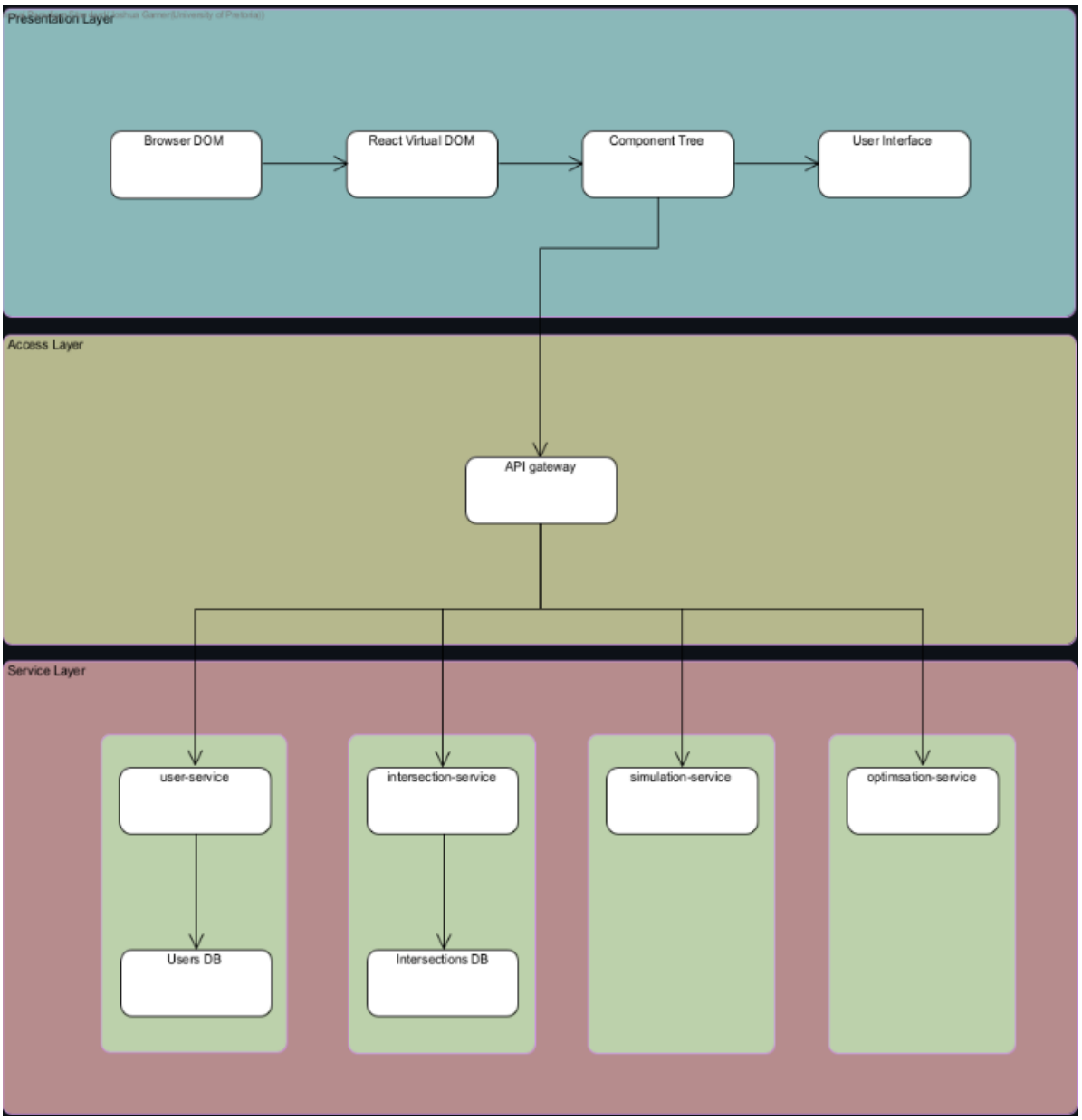
Repository Pattern

- **Promotes extensibility by abstracting data access, making it easy to integrate with different traffic department databases and data sources**
- **Provides efficient data storage and indexing strategies for user and traffic data**
- **Enables backup and recovery strategies through standardized data access interfaces**
- **Supports scalable database operations by abstracting data persistence complexity**
- **Facilitates comprehensive testing through mockable data access layers**
- **Ensures data consistency and integrity across different traffic management operations**

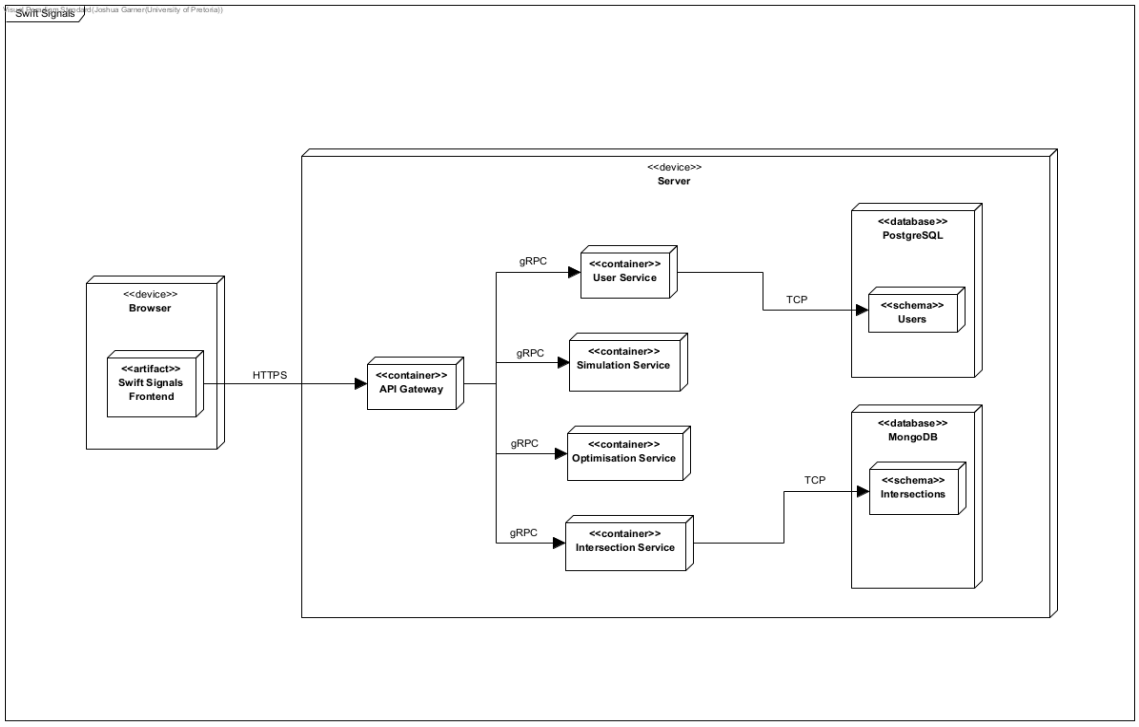
How has it been integrated into Swift Signals?



Architectural Diagram



Deployment Model



Service Contracts

This document defines the service contracts for all microservices in the Swift Signals system. It includes gRPC specifications for internal services and REST specifications for the API Gateway.

Services Overview

Service Name	Description	Protocol	Public?
user-service	Handles user registration, login, and admin privileges	gRPC	✗
intersection-service	Handles intersection creation, fetching and storing	gRPC	✗
simulation-service	Runs intersection simulations	gRPC	✗
optimization-service	Optimises simulations using AI models	gRPC	✗

Service Name	Description	Protocol	Public?
api-gateway	Single entry point for frontend requests (REST → gRPC)	REST/gRPC	☑

user-service

Overview

Handles user account creation, login, and access control. Only this service interacts with the user database and JWT creation.

Proto File: user.proto

```
service UserService {
  rpc RegisterUser(RegisterUserRequest) returns (UserResponse);
  rpc LoginUser(LoginUserRequest) returns (LoginUserResponse);
  rpc LogoutUser(UserIDRequest) returns (google.protobuf.Empty);

  rpc GetUserByID(UserIDRequest) returns (UserResponse);
  rpc GetUserByEmail(GetUserByEmailRequest) returns (UserResponse);
  rpc GetAllUsers(GetAllUsersRequest) returns (stream UserResponse);
  rpc UpdateUser(UpdateUserRequest) returns (UserResponse);
  rpc DeleteUser(UserIDRequest) returns (google.protobuf.Empty);

  rpc GetUserIntersectionIDs(UserIDRequest)
      returns (stream IntersectionIDResponse);
  rpc AddIntersectionID(AddIntersectionIDRequest)
      returns (google.protobuf.Empty);
  rpc RemoveIntersectionIDs(RemoveIntersectionIDRequest)
      returns (google.protobuf.Empty);

  rpc ChangePassword(ChangePasswordRequest) returns (google.protobuf.Empty);
  rpc ResetPassword(ResetPasswordRequest) returns (google.protobuf.Empty);

  rpc MakeAdmin(AdminRequest) returns (google.protobuf.Empty);
  rpc RemoveAdmin(AdminRequest) returns (google.protobuf.Empty);
}
```

Testing

```
/user-service
├── /cmd
├── /internal
│   ├── /handler
│   │   ├── /test        //unit tests for handler layer
│   │   └── handler.go
│   ├── /service
│   │   ├── /test        //unit tests for service layer
│   │   ├── contract.go
│   │   └── service.go
│   ├── /db
│   │   ├── /test        //unit tests for db layer
│   │   ├── contract.go
│   │   └── repository.go
│   └── /test            //integration tests for user-service
│
```

intersection-service

Overview

Handles creation and management of intersection instances.

Proto File: intersection.proto

```
service IntersectionService {
  rpc CreateIntersection(CreateIntersectionRequest)
      returns (IntersectionResponse);
  rpc GetIntersection(IntersectionIDRequest) returns (IntersectionResponse);
  rpc GetAllIntersections(GetAllIntersectionsRequest)
      returns (stream IntersectionResponse);
  rpc UpdateIntersection(UpdateIntersectionRequest)
      returns (IntersectionResponse);
  rpc DeleteIntersection(IntersectionIDRequest) returns (google.protobuf.Empty);
  rpc PutOptimisation(PutOptimisationRequest) returns (PutOptimisationResponse);
}
```

Testing

```
/intersection-service
├── /cmd
├── /internal
│   ├── /handler
│   │   ├── /test        //unit tests for handler layer
│   │   └── handler.go
│   ├── /service
│   │   ├── /test        //unit tests for service layer
│   │   ├── contract.go
│   │   └── service.go
│   ├── /db
│   │   ├── /test        //unit tests for db layer
│   │   ├── contract.go
│   │   └── repository.go
│   └── /test            //integration tests for intersection-service
│
```

simulation-service

Overview

Handles running simulations of intersections.

Proto File: simulation.proto

```
service SimulationService {
  rpc GetSimulationResults(SimulationRequest)
      returns (SimulationResultsResponse);
  rpc GetSimulationOutput(SimulationRequest) returns (SimulationOutputResponse);
}
```

optimisation-service

Overview

Receives simulation parameters and performs optimization using AI.

Proto File: optimisation.proto

```
service OptimisationService {
  rpc RunOptimisation(OptimisationParameters) returns (OptimisationParameters);
}
```

api-gateway

Overview

The only public-facing service. Handles REST requests from the frontend, translates them to gRPC requests for backend services, and returns responses.

Authentication

Required for all endpoints (except signup/login): JWT Bearer token

REST Endpoints

Endpoint	Method	Description	Input (Request Body)	Output (Response)
/admin/users	GET	Retrieves a paginated list of all users.	N/A	model.User array
/admin/users/{id}	GET	Retrieves a user by their ID.	N/A	model.User
/admin/users/{id}	PATCH	Updates a user's details by their ID.	model.UpdateUserRequest	model.User
/admin/users/{id}	DELETE	Deletes a user by their ID.	N/A	No Content (204)

Endpoint	Method	Description	Input (Request Body)	Output (Response)
/intersections	GET	Retrieves all intersections associated with the user.	N/A	model.Intersections
/intersections	POST	Creates a new intersection.	model.CreateIntersectionRequest	model.CreateIntersectionResponse
/intersections/{id}	GET	Retrieves a single intersection by its ID.	N/A	model.Intersection
/intersections/{id}	PATCH	Partially updates an existing intersection.	model.UpdateIntersectionRequest	model.Intersection
/intersections/{id}	DELETE	Deletes an intersec	N/A	No Content (204)

Endpoint	Method	Description	Input (Request Body)	Output (Response)
		tion by its ID.		
/intersections/{id}/optimise	GET	Generates and returns optimized simulation data.	N/A	model.SimulationResponse
/intersections/{id}/optimise	POST	Runs an optimization for an intersection.	N/A	model.OptimisationResponse
/intersections/{id}/simulate	GET	Generates and returns simulation data.	N/A	model.SimulationResponse
/login	POST	Authenticates a user and returns	model.LoginRequest	model.LoginResponse

Endpoint	Method	Description	Input (Request Body)	Output (Response)
		a token.		
/logout	POST	Invalidates the user's session or token.	N/A	model.LogoutResponse
/me	GET	Retrieves the profile of the authenticated user.	N/A	model.User
/me	PATCH	Updates the profile of the authenticated user.	model.UpdateUserRequest	model.User
/me	DELETE	Deletes the profile of the authent	N/A	No Content (204)

Endpoint	Method	Description	Input (Request Body)	Output (Response)
		icated user.		
/register	POST	Registers a new user.	model.RegisterRequest	model.RegisterResponse
/reset-password	POST	Resets a user's password.	model.ResetPasswordRequest	model.ResetPasswordResponse

Testing

```
/api-gateway
├── /cmd
├── /internal
│   ├── /handler
│   │   ├── /test        //unit tests for handler layer
│   │   └── ...
│   ├── /service
│   │   ├── /test        //unit tests for service layer
│   │   └── ...
│   ├── /client
│   │   ├── /test        //unit tests for client layer
│   │   └── ...
│   └── /test            //integration tests for api-gateway
│
```

Uniform Error Handling

```go
type ServiceError struct {
    Code    ErrorCode      `json:"code"`
    Message string         `json:"message"`
    Cause   error          `json:"-"`
    Context map[string]any `json:"context,omitempty"`
}

const (
    ErrValidation    ErrorCode = "VALIDATION_ERROR"
    ErrNotFound      ErrorCode = "NOT_FOUND"
    ErrAlreadyExists ErrorCode = "ALREADY_EXISTS"
    ErrUnauthorized  ErrorCode = "UNAUTHORIZED"
    ErrForbidden     ErrorCode = "FORBIDDEN"
    ErrConflict      ErrorCode = "CONFLICT_ERROR"
    ErrUnavailable   ErrorCode = "UNAVAILABLE_ERROR"
    ErrDatabase      ErrorCode = "DB_ERROR"
    ErrInternal      ErrorCode = "INTERNAL_ERROR"
    ErrExternal      ErrorCode = "EXTERNAL_ERROR"
)
```

Notes

- All internal services communicate using gRPC
- External clients use a REST API via the API Gateway
- Fields and message formats follow .proto definitions
- All endpoints are secured with JWT auth
- Admin-only actions (e.g., banning users) are restricted by role

Technology Choices

Here follows a detailed overview of the technologies, tools, and environments used to develop and deploy the Swift Signals traffic optimization platform, alongside clear justifications based on system quality requirements.

System Overview

Swift Signals is a web-based, microservices-driven traffic simulation and optimization platform for traffic planners and municipalities. It enables intersection configuration, simulation, and AI-driven signal optimization, designed to meet strict performance, scalability, and maintainability goals.

Technology Stack & Architecture

Frontend

- **Framework: React**
- **Build Tool: Vite**
- **Language: TypeScript**
- **Styling: Tailwind CSS**
- **Justification: Provides fast, responsive, maintainable UI; ensures consistent styling and seamless user experience.**

API Gateway

- **Language: Go**
- **Functions: Routes requests, handles load balancing, validates authentication, aggregates services**
- **Communication: HTTP/gRPC**
- **Justification: Go's performance and concurrency handle high traffic loads; gRPC enables efficient, secure inter-service communication.**

User Authentication Service

- **Language: Go**
- **Authentication: JWT-based login and signup**
- **Database: MongoDB**

- **Justification: JWT ensures secure, scalable user sessions; MongoDB provides flexible storage for user data.**

Simulation Service

- **Languages: Go & Python (integrating SUMO)**
- **Function: Executes realistic, configurable traffic simulations**
- **Justification: SUMO offers detailed, accurate modelling of urban traffic; Go ensures performance and stability.**

Optimization Service

- **Language: Go**
- **Techniques: Swarm Optimization Algorithms**
- **Function: Optimizes traffic signal timings based on simulation data**
- **Justification: Swarm algorithms enable efficient search for optimal signal patterns under dynamic traffic conditions.**

Metrics Service

- **Language: Go**
- **Function: Collects system and simulation metrics**
- **Database: Prometheus with local time-series storage**
- **Justification: Prometheus enables scalable, real-time metrics collection crucial for observability.**

Controller Service

- **Language: Go**
- **Function: Orchestrates communication between simulation and optimization components**
- **Justification: Ensures reliable coordination of system components, maintaining modularity and fault isolation.**

Communication Protocols

- **Inter-service Communication: gRPC**
- **External API Interface: REST with JSON**
- **API Specifications: Protobuf definitions maintained in Service Contracts**

Development & Deployment

- **Local Development: Minikube, Docker, Kubernetes, Tilt/Skaffold for hot-reload**
- **Production Environments:**
  - **On-premises servers**
  - **Cloud platforms (GCP, AWS, DigitalOcean)**
- **Deployment Features: Kubernetes orchestration, Docker Registry, load balancing, ingress controllers**

Observability & Monitoring

Logging

- **In-Service: Zap (Go), Logrus (Go), Python Logging**
- **Aggregation: Grafana Loki**
- **Collection: Promtail or Fluent Bit**

Monitoring

- **Metrics: Prometheus**
- **Dashboards: Grafana**
- **Exporters: prometheus/client_golang for Go services**

Design Principles

- **Open Source: Built on open-source tools (SUMO, Prometheus, Grafana)**
- **Portability: Compatible with any Kubernetes-compliant environment**
- **Extensibility: Supports new intersection models, AI techniques, metrics**
- **Security: JWT authentication, RBAC, secure gRPC with optional TLS**

Quality Requirements & Technology Justification

Performance

Requirement: **Real-time data processing, complex simulations, UI response < 2s, optimization < 30s**

Technology Justification: **Go's efficiency; SUMO's realism; React + Vite's fast frontend**

Scalability

Requirement: **Scale from single intersections to city-wide; handle large datasets**

Technology Justification: **Microservices, Kubernetes, MongoDB, Prometheus enable elastic scaling**

Maintainability

Requirement: **Independent service updates, iterative improvements**

Technology Justification: **Isolated microservices; strong typing with Go/TypeScript; Tailwind for UI consistency**

Security

Requirement: **Robust authentication, secure service communication**

Technology Justification: **JWT for secure sessions; gRPC for efficient, encrypted inter-service calls**

Responsiveness

Requirement: **Fast, seamless user interaction**

Technology Justification: **React, Vite, and Tailwind ensure lightweight, high-speed UI**

Technology Choices Justification Aligned to System Requirements

Swift Signals technology stack was chosen to directly address the system's architectural goals and critical quality requirements:

Microservices with Containerization (Go, Docker, Kubernetes)

- **Supports independent development, deployment, and horizontal scalability**
- **Aligns with maintainability, scalability, and operational constraints**
- **Enables resource isolation and efficient fault recovery**

API Gateway with gRPC & REST (Go)

- **Simplifies client interaction with a unified REST interface**
- **Provides efficient, secure inter-service communication via gRPC**
- **Implements cross-cutting concerns like rate limiting, logging, and authentication**
- **Enhances system reliability through circuit breaker and observer patterns**

SUMO for Simulation (Python/Go Integration)

- Delivers realistic, industry-standard traffic modelling
- Supports future extensibility with new intersection types and traffic behaviours

Swarm Optimization Algorithms (Go)

- Optimizes signal timing dynamically under varying traffic conditions
- Facilitates scalable, AI-driven optimization aligned with performance goals

MongoDB & PostgreSQL with Database-Per-Service Pattern

- MongoDB provides flexible user and configuration data storage
- PostgreSQL supports scalable, performant time-series traffic metrics
- Eliminates tight coupling between services, enhancing maintainability

React + Vite + Tailwind CSS Frontend

- Provides an intuitive, responsive user experience for all technical levels
- Facilitates rapid development and consistent UI styling

Prometheus & Grafana for Monitoring

- Real-time system visibility and performance metrics
- Supports proactive health monitoring aligned with availability requirements

CQRS and Event-Driven Architecture

- Enhances system resilience, decouples services, and optimizes read/write workloads
- Enables scalability and reliability for high-volume data flows

Security Practices (JWT, encrypted communication, RBAC)

- Enforces robust authentication and authorization mechanisms
- Ensures data protection in transit and at rest

Development & Operational Tools (CI/CD, YAML config, automated testing)

- Streamlines environment-specific deployments

- **Supports maintainability, testability, and system evolution**

These integrated technology choices ensure Swift Signals achieves its system requirements for performance, scalability, reliability, security, maintainability, and extensibility.