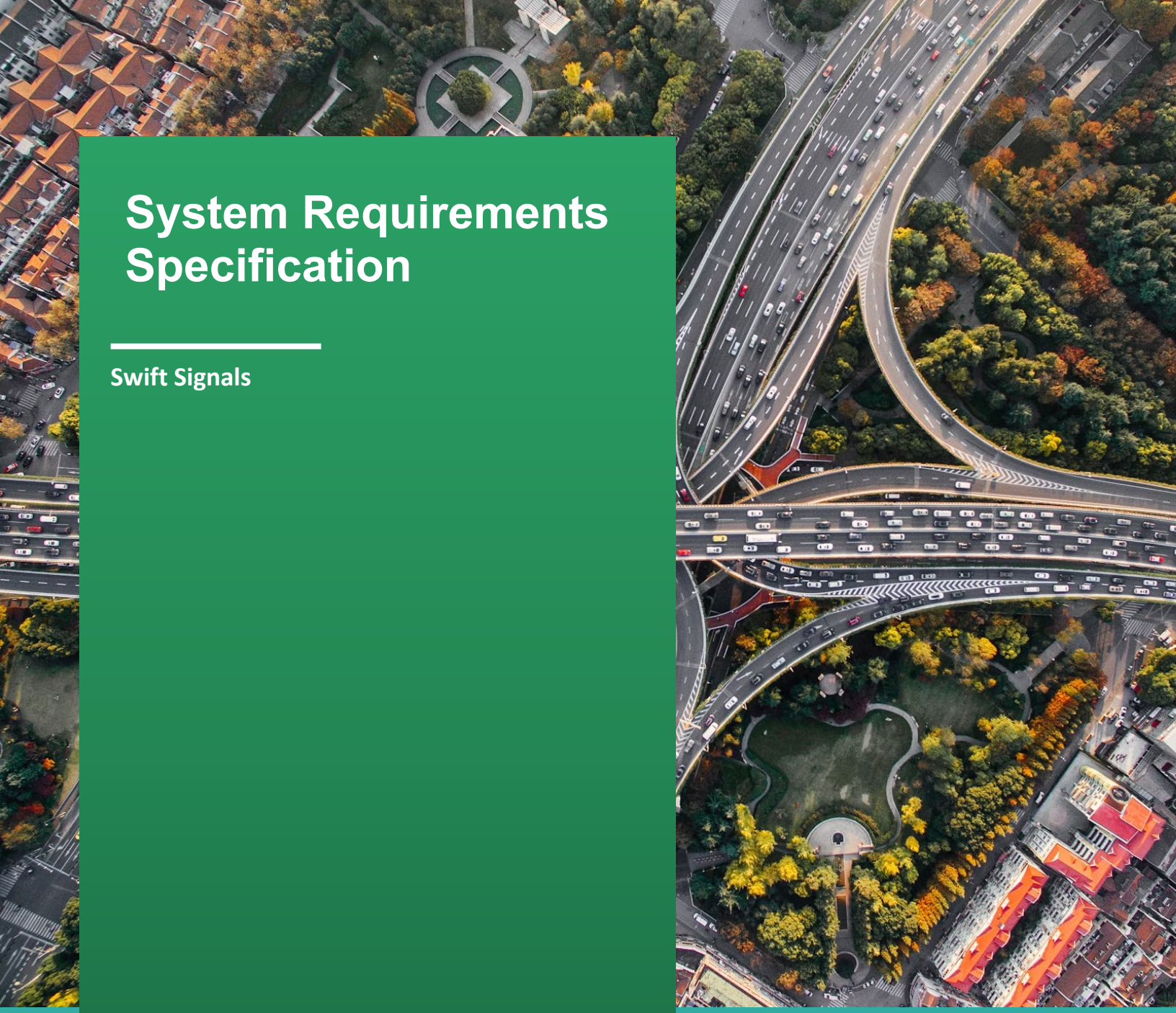


System Requirements Specification

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

Creating a New Simulation

Description:

- Allows a user to create a new traffic simulation for a single intersection. The user provides basic attributes and then manually configures the simulation's parameters.

Trigger:

- The user clicks the "New Simulation" button on the Simulations Page.

Preconditions:

- The user must be authenticated and logged into the system.

Postconditions:

- ***On Success***: A new simulation entity is created, configured, and saved in the system, associated with the user's account. The user is redirected to the "Run Simulation" page for the newly created simulation.
- ***On Failure***: The system displays an error message, and no simulation is created. The user remains on the creation screen.

Main Success Scenario (Manual Configuration):

1. The User clicks the "New Simulation" button.
2. The System presents a form to define the simulation's attributes.
3. The User enters a unique Name for the simulation.
4. The User specifies the Location (e.g., by selecting on a map or typing an address) and provides the Intersection Names for the intersection.
5. The System displays input fields for variable parameters.
6. The User enters values for Traffic probabilities for each road and Traffic light timing durations.
7. The User confirms the creation.
8. The System validates all provided information.
9. The System saves the new simulation.
10. The System redirects the user to the "Simulation Results Page" page for this new simulation.

Extensions (Alternative Flows):

8a. Validation Fails:

1. If the System determines that any input is invalid (e.g., name is not unique, probabilities are not valid), it displays a specific error message next to the field(s) in question.
2. The use case pauses, and the user is allowed to correct the invalid data and re-submit.

Help Menu

Manage Help Menu

Description:

- Allows a user to open and close the slide-out help menu.

Trigger:

- The user clicks the "HELP" button on the side of the screen or the 'X' button inside the menu.

Preconditions:

- The user is viewing any page within the Swift Signals application.

Postconditions:

- ***On Success***: The visibility of the Help Menu is toggled (opened or closed), giving the user access to its features.

Main Success Scenario:

1. The User clicks the "HELP" button.
2. The System presents the Help Menu, sliding it into view.
3. The User clicks the 'X' button.
4. The System hides the Help Menu, sliding it out of view.

Use Swift Chat

Description:

- Allows a user to interact with the "Swift Chat" chatbot to ask questions, receive answers, and trigger actions like starting tutorials.

Trigger:

- The user selects the "Swift Chat" tab within the open Help Menu.

Preconditions:

- The user must be authenticated and logged in.

Postconditions:

- ***On Success***: The user receives a relevant text response and/or interactive options (quick replies) from the chatbot.
- ***On Failure***: The system displays an error message in the chat interface, and no response is received from the bot.

Main Success Scenario:

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

Extensions (Alternative Flows):

8a. User Interacts with Quick Reply:

1. The bot's response includes a list of quick reply buttons.
2. The User clicks one of the quick reply buttons.
3. The System sends the payload of the selected reply to the backend as a new query.
4. The use case continues from step 6.

8b. Chatbot Initiates a Tutorial:

1. The Dialogflow backend identifies the user's intent as start.tutorial.
2. The backend response includes the action to start a specific tutorial.
3. The System triggers the "Take Interactive Tutorial" use case.

8c. Backend Connection Fails:

1. The System's request to the chatbot backend fails (e.g., due to a network error or server issue).
2. The System displays a generic error message in the chat interface (e.g., "Sorry, I'm having trouble connecting."). The use case ends.

Access General Help

Description:

- Allows a user to find help manually by browsing a list of available tutorials and a Frequently Asked Questions (FAQ) section.

Trigger:

- The user selects the "General Help" tab within the open Help Menu.

Preconditions:

- The user has opened the Help Menu.

Postconditions:

- ***On Success***: The user finds and consumes the desired help information or launches a tutorial.

Main Success Scenario (Browse FAQs):

1. The User clicks the "General Help" tab.
2. The System presents "Tutorials" and "Frequently Asked Questions" sections.
3. The User clicks the "Frequently Asked Questions" header.
4. The System expands the section to show a list of questions.
5. The User clicks on a question.
6. The System expands the selected item to display the answer.

Extensions (Alternative Flows):

2a. User Launches a Tutorial:

1. The User clicks the "Tutorials" header.
2. The System expands the section to show a list of available tutorials.
3. The User clicks on a specific tutorial button (e.g., "Dashboard Tutorial").

4. The System triggers the "Take Interactive Tutorial" use case.

Take Interactive Tutorial

Description:

- Guides the user through the application's user interface with a sequence of focused highlights and descriptive tooltips.

Trigger:

- The user requests a tutorial via the Chatbot or launches one from the General Help menu.

Preconditions:

- A tutorial has been requested by the user.

Postconditions:

- ***On Success***: The user has been guided through all steps of the tutorial. The tutorial overlay is removed from the screen.
- ***On Failure***: The tutorial does not start if the user declines a necessary navigation step.

Main Success Scenario:

1. The System closes the Help Menu.
2. The System displays an overlay on the page and highlights the first UI element of the tutorial sequence with a tooltip.
3. The User reads the tooltip and clicks the "Next" button.
4. The System advances the tutorial, highlighting the next UI element in the sequence.
5. The User repeats step 3 and 4 until the final step is reached.
6. The User clicks the "Close" button on the final tooltip.
7. The System removes the tutorial overlay and tooltips.

Extensions (Alternative Flows):

1a. Navigation Is Required:

1. The System detects that the user is not on the correct page for the requested tutorial.

2. The System displays a confirmation popup asking the user if they wish to navigate to the correct page.
3. The User clicks "Yes".
4. The System navigates the user to the required page.
5. The use case continues from the Main Success Scenario, step 1.

3a. User Declines Navigation:

1. In step 3 of the "Navigation is Required" flow, the user clicks "No".
2. The System closes the confirmation popup, and the tutorial is aborted. The use case ends.

2b. User Aborts Tutorial:

1. At any point during the tutorial, the user clicks the "Close" button on the tooltip.
2. The System immediately removes the tutorial overlay and tooltips. The use case ends.

View and Manage Simulations

Description:

- Allows the user to view a list of all their saved simulations, filter the list to find specific ones, and delete simulations that are no longer needed.

Trigger:

- The user navigates to the "Simulations" Page.

Preconditions:

- The user must be authenticated and logged in.

Postconditions:

- The user has viewed their list of simulations, or a simulation has been permanently removed from the system.

Main Success Scenario (Manual Configuration):

1. The User navigates to the simulation’s dashboard.
2. The System retrieves and displays a list of all simulations associated with the user's account, showing key details like Name, Location, and Status.
3. The User clicks on a the "view button" of a simulation in the list.

4. The System redirects the user to the "Simulation Results" page (Run a Simulation Use Case) for the selected simulation.

Extensions (Alternative Flows):

2a. Filter Simulation List:

1. At step 2, the user interacts with filter controls (a drop down for intersections).
2. The user selects a filter criterion.
3. The System refreshes the list to show only the simulations that match the criterion. The use case continues.

3a. Delete a Simulation:

At step 3, the user clicks the "Delete" icon next to a simulation in the list.

1. The System presents a confirmation dialog (e.g., "Are you sure you want to delete this simulation?").
2. The User confirms the deletion.
3. The System permanently removes the simulation and its associated data from the database.
4. The System refreshes the simulation list to reflect the removal.

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: Login/Signup
R1.1: Must allow users to login their existing account details
R1.2: If user does not have an account, allow user to create account
R1.3: Once registered, allow users to login
R1.4: Allow users to switch between light/dark mode.

R2: Dashboard page
R2.1: Include navbar in all pages that allows users to access various pages such as the simulations page.
R2.2: Show statistics such as total intersections, active simulations, traffic volume and top intersections.
R2.3: Show recent simulations as well as their status and allow users to view details about them.
R2.4: Allow users to insert a new intersection, run simulation and view the map.
R2.5: Allow users to logout by clicking the logout button.

R3: Intersections page
R3.1: Allow users to view all listed intersections
R3.2: Allow users to search a specific intersection
R3.3: Allow users to add a new intersection
R3.4: Allow users to be able to simulate, edit and delete a specific intersection

R4: Simulation page
R4.1: Users should be able to view recent simulations and optimizations with information about them such as average wait time and throughput.
R4.2: Allow users to insert a new simulation.
R4.3: Allow users to select a specific intersection for filtering purposes.
R4.4: Allow users to navigate through simulation pages by clicking on 'next', 'prev' or a page number buttons.

R5: Users Page

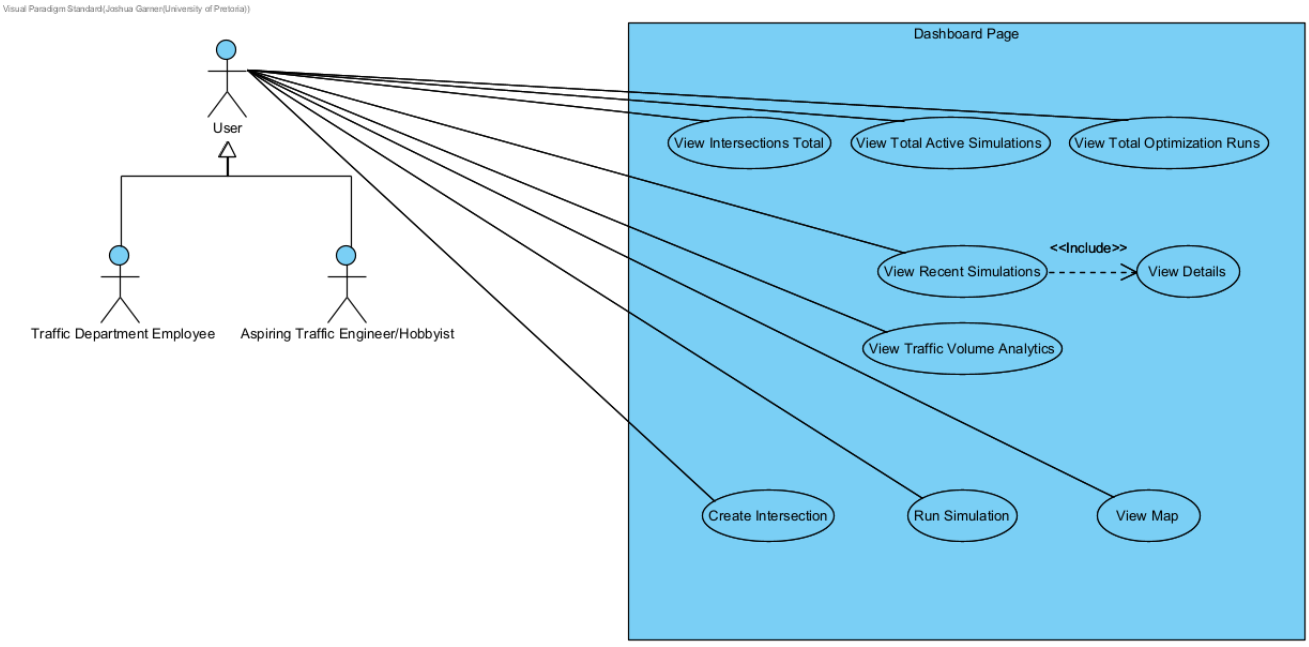
R5.1: Users must be able to view all listed users, with their information included.

R5.2: Allow users to be able to navigate through table pages.

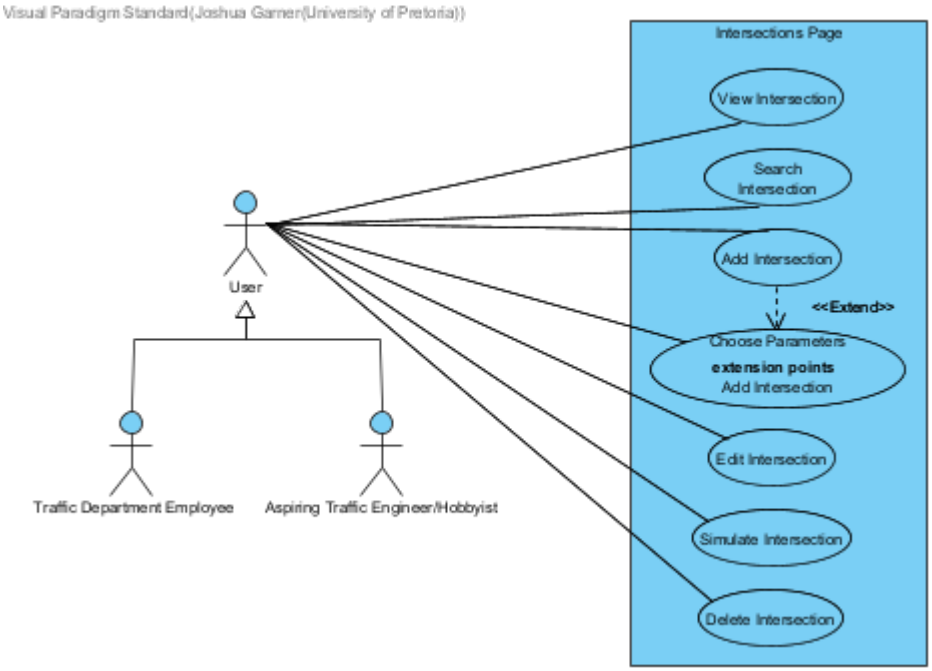
R5.3: Users must be able to edit and delete information.

Use Case Diagrams

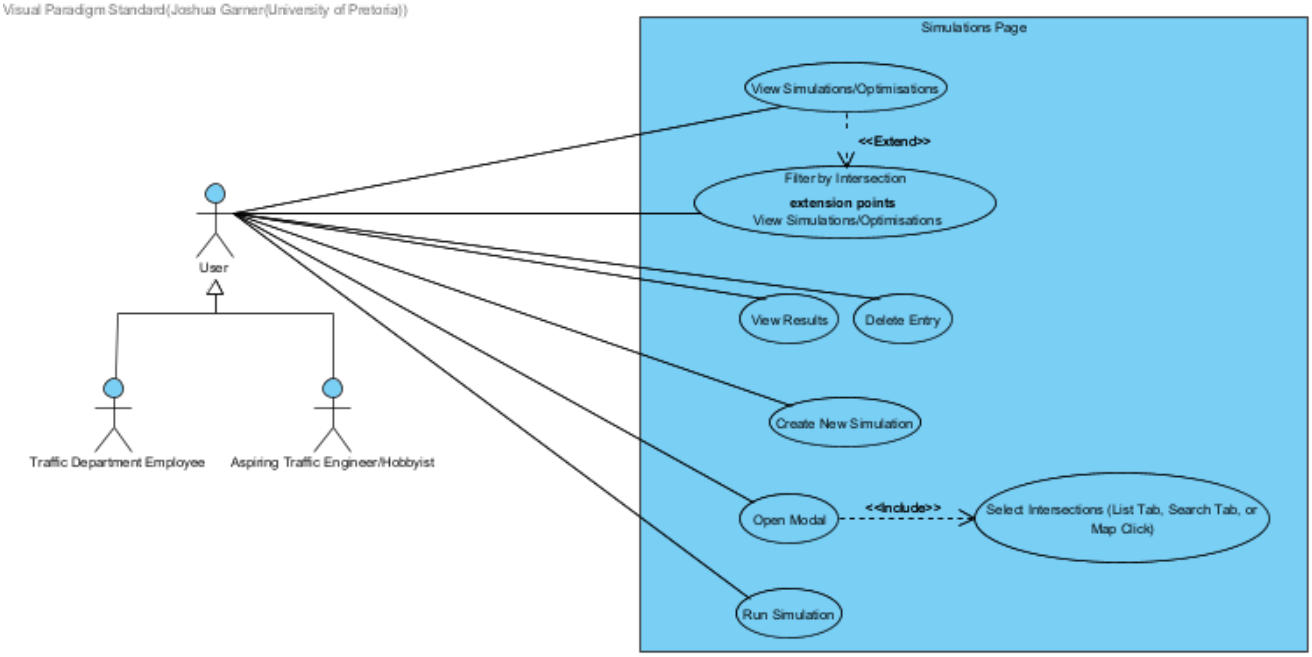
Dashboard



Intersections

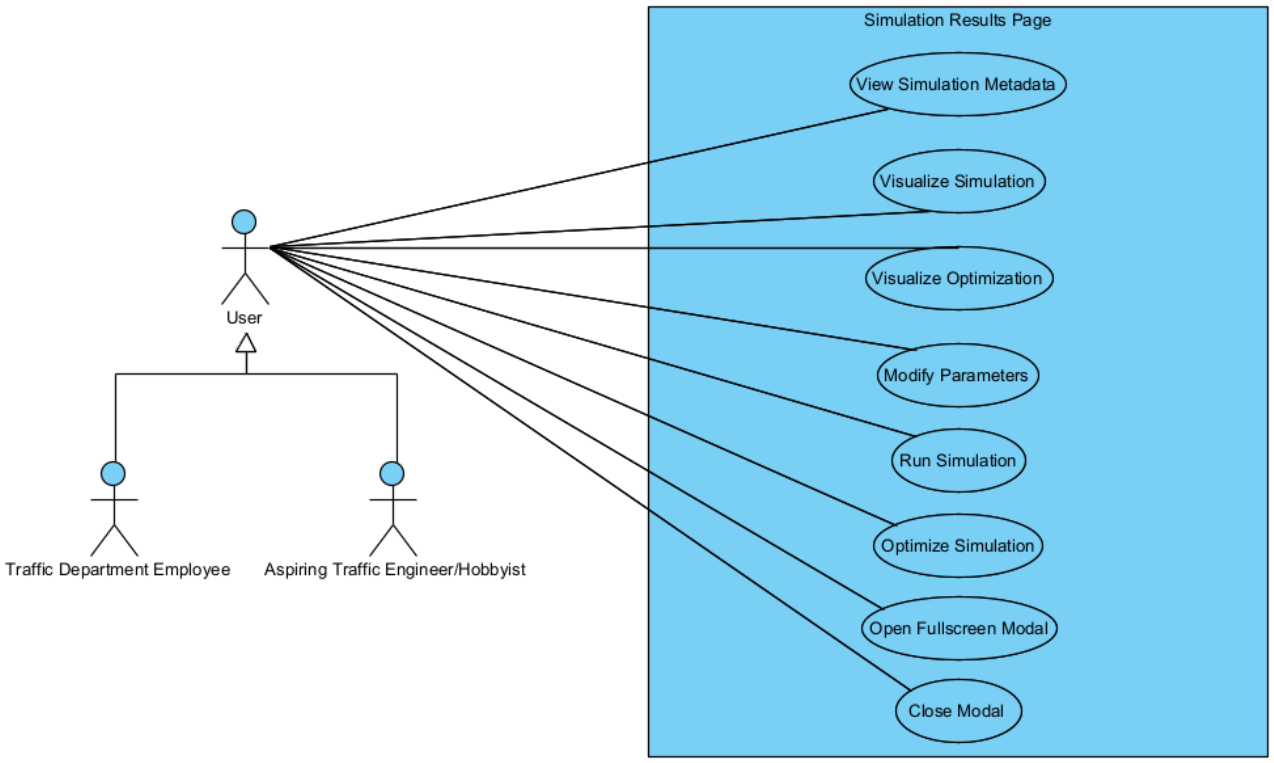


Simulations



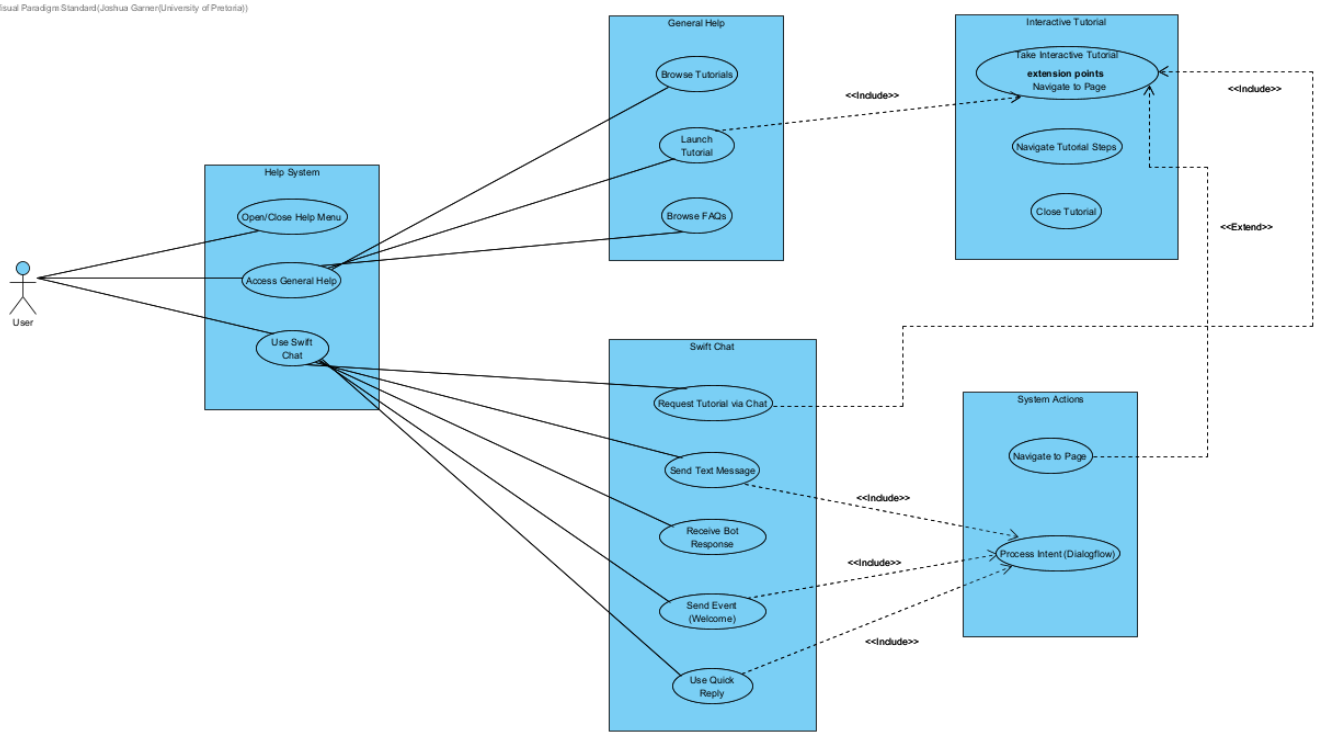
Simulation Results

Visual Paradigm Standard(Joshua Garner(University of Pretoria))

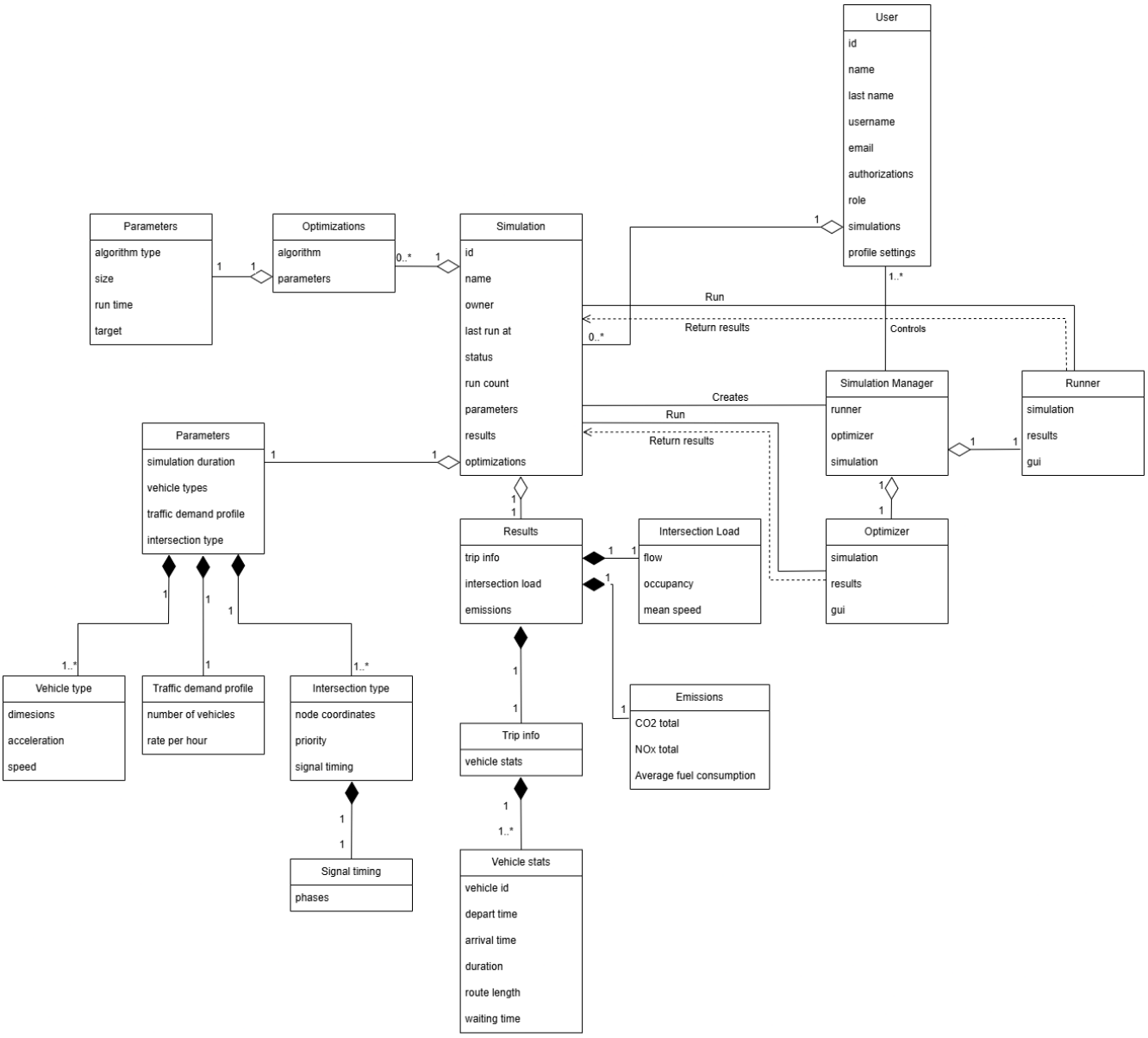


Help Menu

Visual Paradigm Standard(Joshua Garner(University of Pretoria))



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

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.

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 Patterns

Microservices Architecture

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.

Services:

- **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.

Design Patterns

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.

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

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.