WorkShop II – SimWaze

Rubén Darío Fúquene Castiblanco - 20192020004
Thomas Felipe Sarmiento Oñate - 20201020068

Overview

This document describes the core components of the SimWaze platform. Although the system includes a complete architectural design with data ingestion, processing, and analytics, the focus here is on functional areas that structure the application's logic and data interaction.

Problem Definition and System Design

The traffic system must manage a road infrastructure of 137,957 roads per city (taking Bogotá as an example) with dynamic traffic data generated every 2 minutes per road. This captures the number of vehicles and average speed, resulting in more than 198 million records daily per city (137,957 roads $\times$ 2 every 2 minutes). The system must serve 180 million users requiring response times of less than 140 milliseconds.

The system design is structured as a three-layer interconnected architecture that separates responsibilities according to the nature of the data and access patterns. The first layer focuses on massive real-time data collection, handling the continuous flow of traffic information arriving every 2 minutes from multiple sources across the city. The second layer is dedicated to the permanence and structuring of historical data, organizing information for efficient analytical queries and maintaining the temporal context necessary for traffic pattern analysis. The third layer constitutes the high-speed service subsystem, optimized to respond to end-user queries with minimal latency through intelligent caching strategies and precomputed data.

The proposed system as a distributed architecture considers that static data of the road network changes minimally, while traffic data requires constant updating but follows predictable patterns by area and time. The system implements geographic partitioning to distribute the load by city sectors, taking advantage of the fact that traffic queries tend to be geographically localized. The architecture also considers that most queries are concentrated in popular routes and peak hours, allowing differentiated caching strategies based on access frequency and the temporal criticality of the data.

1. Business Model

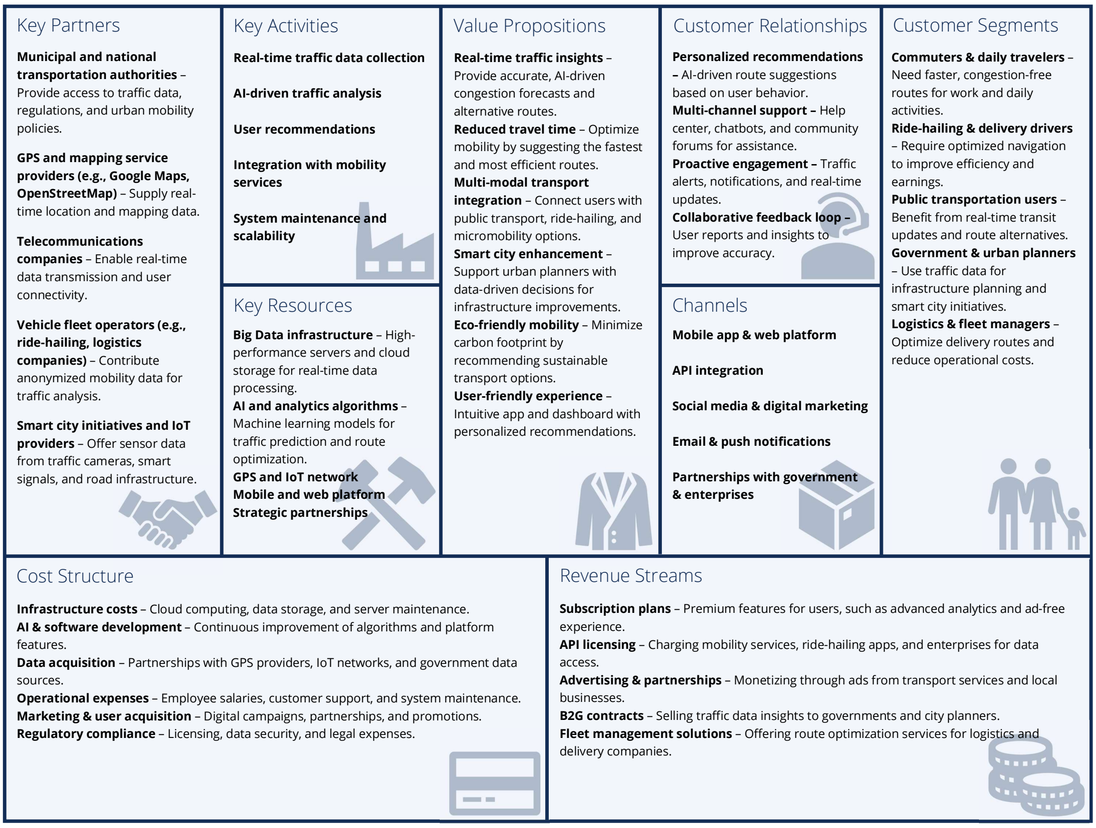

Figure 1: Business model

2. User Histories

Title 1	Priority 5	Estimate 5

User History: As a commuter, I want to receive real-time traffic updates, so I can choose the fastest route to work.

Acceptance Criteria:

- View two or more routes with similar travel times.
- Understand route suggestions (traffic, closures).
- Display in less than a second.
- Real-time traffic notifications.

Title 2	Priority 3	Estimate 4

UserHistory: As a commuter, I want optimized route suggestions, so I can complete more deliveries in less time.

AcceptanceCriteria:

- Select POIs and generate route with those points.
- Use different colors for routes.
- Change destination at any time.

Title 3	Priority 2	Estimate 3

UserHistory: As a commuter, I want real-time bus and train arrival times, so I can plan my trips efficiently.

AcceptanceCriteria:

- Search routes via bus/train.
- Search by time.

Title 4	Priority 5	Estimate 3

UserHistory: As a commuter, I want an easy-to-use interface, so I can quickly understand how to navigate the app.

AcceptanceCriteria:

- App guidance after sign-up.
- First search under 1 minute.

Title 5	Priority 2	Estimate 2

UserHistory: As a commuter, I want bike-friendly route recommendations, so I can travel safely and efficiently.

AcceptanceCriteria: –

Title 6	Priority 1	Estimate 4

UserHistory: As an administrator, I want access to traffic data insights, so I can improve urban mobility and reduce congestion.

Title 6	Priority 1	Estimate 4

AcceptanceCriteria:

- Display info in ¡ 1 min.
- Search by day/week/month/year.

Title 7	Priority 1	Estimate 5

UserHistory: As an administrator, I want to track and optimize fleet routes, so I can lower fuel costs and delivery times.

AcceptanceCriteria:

- Search best route by hour.
- Display in ¡ 1 minute.

Title 8	Priority 5	Estimate 4

UserHistory: As a premium subscriber, I want an ad-free experience and advanced analytics, so I can get the best insights for my travels.

AcceptanceCriteria:

- Search priority, ¡1s response.
- No ads.
- Offset ad revenue with payment.
- See daily travel info.

Information Requirements and Alignment with Business Goals

1. Types of Information Retrieved

The SimWaze platform retrieves and processes various types of information to fulfill its value propositions and user needs:

- **Real-Time Traffic Data:** Includes congestion levels, road closures, accidents, and travel time across routes. Essential for commuter and delivery user stories (e.g., US1, US2).
- **Route Alternatives and Recommendations:** System must compute and display multiple optimized route suggestions, using real-time and historical traffic data.

- **Transit Arrival Times:** Integration with public transportation data (buses and trains), providing real-time ETAs (US3).
- **Personalized User Preferences:** Includes interface accessibility settings, preferred transportation modes, and history-based suggestions.
- **Environmental Metrics:** Eco-friendly routing suggestions to help reduce users' carbon footprint (US8).
- **Administrative Insights:** Dashboards and analytics for urban planners, fleet managers, and government contracts (US6, US7).
- **Usage and Subscription Tracking:** Ad-free experiences, premium feature access, and personalized statistics for paying users (US9).

2. Link to Business Model and User Stories

The retrieval of these information types is directly aligned with both the user stories and the business model defined for SimWaze:

- **Real-time traffic insights** and **reduced travel time** directly support the commuter and delivery driver segments (US1, US2, US3, US5).
- **Smart city enhancement** and **eco-friendly mobility** are supported through data provided to administrators (US6, US7, US8) and external agencies, contributing to B2G revenue streams.
- **Personalized recommendations** and **user-friendly experience** enhance customer satisfaction and retention (US4, US9).
- **Subscription plans** and **ad-free premium experience** relate to the monetization strategy in the revenue model (US9).
- **Multi-modal transport integration** and **integration with mobility services** are supported by real-time route calculation and POI-based suggestions (US2, US3).

System Components

This section presents the main functional components that structure the SimWaze solution. Each component includes a group of entities related to specific responsibilities, such as user management, route navigation, integration with external services, among others. The goal is to organize the data model according to functional areas, identifying the key entities, their attributes, and how they contribute to the overall system logic.

2.1. User Management

This component handles everything related to users: from registration and login, to user settings, sessions, roles, and activity logs. It allows the system to differentiate between types of users (commuters, subscribers, admins) and ensures that each has access to the features and data that match their role. It also manages account preferences like language, notifications, and interface themes.

Users

Attribute	Type	Description
id	int (PK)	Unique identifier for each user.
name	varchar(100)	Full name of the user.
email	varchar(100)	Email used for login and communication.
password_hash	varchar(200)	Encrypted user password.
role_id	int (FK)	References the user's role.
language	varchar(100)	Preferred language for the interface.
created_at	datetime	Timestamp when the account was created.
last_login_at	datetime	Timestamp of the last login.
status	enum	User status: active, suspended, deleted, etc.

Roles

Attribute	Type	Description
id	int (PK)	Unique identifier for the role.
name	varchar(100)	Name of the role (e.g., Admin, User).
description	varchar(500)	Description of the role's purpose and permissions.

Sessions

Attribute	Type	Description
id	int (PK)	Session identifier.
user_id	int (FK)	User associated with the session.
token	varchar(200)	Authentication token.
ip_address	varchar(100)	IP address of the session.
device	varchar(100)	Device or browser used.
created_at	datetime	Session start time.
expires_at	datetime	Session expiration time.

UserSettings

Attribute	Type	Description
id	int (PK)	Unique settings identifier.
user_id	int (FK)	Related user.
language	varchar(100)	Interface language preference.
dark_mode	tinyint	Whether dark mode is enabled.
default_view	varchar(100)	Default screen after login.
notifications_enabled	tinyint	Whether user receives notifications.
created_at	datetime	Settings creation date.
updated_at	datetime	Last update timestamp.

UserActivityLogs

Attribute	Type	Description
id	int (PK)	Log entry identifier.
user_id	int (FK)	User who performed the action.
action	varchar(500)	Description of the action taken.
ip_address	varchar(100)	IP from which the action was executed.
device	varchar(100)	Device or browser used.
location	varchar(200)	Geographic location, if available.
created_at	datetime	When the action occurred.

PasswordResets

Attribute	Type	Description
id	int (PK)	Password reset request identifier.
user_id	int (FK)	Associated user.
token	varchar(200)	One-time reset token.
expires_at	datetime	Token expiration date and time.
used	tinyint	Whether the token has been used.
created_at	datetime	Date the request was created.

Entity Relationship Matrix – User Management Component

This matrix shows the relationships between entities in the User Management component. An "X:X" indicates a direct relationship (e.g., foreign key or dependency).

Entity	Users	Roles	Sessions	UserSettings	ActivityLogs	PasswordResets
Users		1:N	N:1	1:1	1:N	1:N
Roles	N:1					
Sessions	1:N					
UserSettings	1:1					
UserActivityLogs	N:1					
PasswordResets	N:1					

Legend:

- **1:N** – One-to-many
- **N:1** – Many-to-one
- **1:1** – One-to-one

2.2. Route Search & Navigation

This is where the magic happens for end users. Based on map data and traffic conditions, the system allows users to search for optimal routes between two points. It uses real-time traffic events, historical data, and road segments to offer dynamic suggestions. It also tracks past trips and allows users to view alternative routes in case of congestion.

RoadSegments

Attribute	**Type**	**Description**
id	int (PK)	Unique identifier for the road segment.
name	varchar(100)	Name of the street or segment.
geometry	geometry/JSON	LineString or GeoJSON path of the road.
length_meters	float	Length of the segment in meters.
road_type	enum	Type of road (e.g., highway, local, bike lane).
speed_limit	float	Legal or estimated speed limit.
area	varchar(100)	City zone, district, or neighborhood.
is_active	boolean	Whether the segment is currently in use.

TrafficEvents

Attribute	Type	Description
id	int (PK)	Unique event identifier.
road_segment_id	int (FK)	Road segment affected by the event.
data_source_id	int (FK)	Data source reporting the event.
event_type	enum	Type of event (e.g., accident, congestion, closure).
average_speed	float	Measured average speed on the segment.
congestion_level	int	Congestion scale (e.g., 0 to 5).
timestamp	datetime	Time when the event was recorded.
duration_estimate	int	Estimated delay in minutes.

TripRecords

Attribute	Type	Description
id	int (PK)	Unique trip identifier.
user_id	int (FK)	User who made the trip (optional).
origin	geometry/JSON	Starting point coordinates.
destination	geometry/JSON	Destination point coordinates.
start_time	datetime	Time when the trip started.
end_time	datetime	Time when the trip ended.
distance_km	float	Total distance of the trip in kilometers.
duration_min	float	Total trip duration in minutes.
avg_speed	float	Average speed across the trip.

AlternativeRouteSuggestions

Attribute	Type	Description
id	int (PK)	Unique identifier for the suggested route.
trip_id	int (FK)	Original trip related to the suggestion.
geometry	geometry/JSON	Path of the alternative route.
estimated_time_min	float	Estimated time to complete the route.
distance_km	float	Length of the suggested route.
congestion_score	float	Traffic-based score or weight.
suggested_at	datetime	Time when the suggestion was generated.

Entity Relationship Matrix – Route Search & Navigation Component

This matrix shows the relationships between entities in the Route Search & Navigation component. An "X:X" marks a direct relationship between entities.

Entity	RoadSegments	TrafficEvents	TripRecords	AltRouteSuggestions
RoadSegments		N:1	N:N	N:N
TrafficEvents	1:N			
TripRecords	N:N			1:N
AltRouteSuggestions	N:N		N:1	

Legend:

- **1:N** – One-to-many
- **N:1** – Many-to-one
- **N:N** – Many-to-many (with junction table)

2.3. Notifications & Alerts

This module is responsible for keeping users informed. Whether it's an accident on their usual route or a better option to reach their destination, the system sends timely notifications through email, app alerts, or dashboards. Internally, it stores which alerts were delivered, when, and to whom, helping improve the user experience.

UserNotifications

Attribute	**Type**	**Description**
id	int (PK)	Unique identifier for the notification.
user_id	int (FK)	User who receives the notification.
type	enum	Notification type (e.g., congestion, info, reminder).
title	varchar(200)	Short title for the notification.
message	text	Full notification message content.
was_read	boolean	Whether the user has read the notification.
delivered_at	datetime	Timestamp when it was delivered.

SystemAlerts

Attribute	Type	Description
id	int (PK)	Unique identifier for the system alert.
alert_type	enum	Alert type (e.g., system_event, maintenance, weather).
message	text	Alert message to be distributed.
target_audience	enum	Who should receive it (e.g., all, admins, subscribers).
start_time	datetime	When the alert becomes active.
end_time	datetime	When the alert is no longer valid.
created_by	int (FK → Users)	User/admin who issued the alert.

Entity Relationship Matrix – Notifications & Alerts Component

This matrix shows the relationships between entities in the Notifications & Alerts component. An "X:X" indicates a direct relationship.

Entity	UserNotifications	SystemAlerts
UserNotifications		1:N
SystemAlerts	N:1	

Legend:

- **1:N** – One-to-many
- **N:1** – Many-to-one

2.4. Integration with External Services

The system relies heavily on third-party data sources: Google Maps APIs, TransMilenio data, IoT traffic sensors, and even scraped public information. This component handles the connection, authentication, and data flow from those services to the internal data lake and warehouse. It also monitors sync times and connection health.

ExternalDataSources

Attribute	Type	Description
id	int (PK)	Unique identifier for the external data source.
name	varchar(100)	Name of the source (e.g., Google Maps, TransMilenio).
type	enum	Source type: api, sensor, scraper, file, etc.
status	enum	Current connection status (e.g., active, failed, disabled).
last_sync_at	datetime	Timestamp of the last successful data ingestion.
sync_interval_min	int	Expected sync frequency in minutes.
endpoint_url	varchar(255)	URL or path used for connection (if applicable).
auth_type	enum	Authentication method: api_key, oauth, none.

ExternalSyncLogs

Attribute	Type	Description
id	int (PK)	Unique identifier for the sync log entry.
source_id	int (FK → ExternalDataSources)	External source involved in the sync.
sync_start_at	datetime	When the sync process started.
sync_end_at	datetime	When it finished.
status	enum	Result: success, partial, failed.
records_fetched	int	Number of data records fetched.
error_message	text	Error message if the sync failed.

Entity Relationship Matrix – Integration with External Services Component

This matrix shows the relationships between entities in the Integration with External Services component.

Entity	ExternalDataSources	ExternalSyncLogs
ExternalDataSources		N:1
ExternalSyncLogs	1:N	

Legend:

- **1:N** – One-to-many
- **N:1** – Many-to-one

2.5. Localization & Accessibility

Designed for a diverse user base, the system includes support for multiple languages (initially Spanish and English) and customizable interface settings. This module also ensures accessibility in terms of device compatibility and user interface design, making the platform usable for as many people as possible.

Localization and accessibility are implemented at the data level through the `UserSettings` entity. Preferences such as language, text size, interface theme, and contrast mode are directly configured per user. These preferences are considered throughout the platform UI and API responses, making localization a native part of the User Management logic.

2.6. Transactions

Subscriptions, plans, and payments all fall under this component. It manages billing cycles, stores payment history, and tracks which features are available to paid users. It also supports multiple payment methods and handles auto-renewal logic where applicable.

Plans

Attribute	Type	Description
id	int (PK)	Unique identifier for the plan.
name	varchar(100)	Name of the plan (e.g., Free, Pro, Premium).
description	text	Description of included features.
price	decimal(10,2)	Cost of the plan.
duration_in_days	int	Validity period of the plan in days.
is_active	boolean	Whether the plan is currently available.
created_at	datetime	Creation timestamp.

Subscriptions

Attribute	Type	Description
id	int (PK)	Unique subscription ID.
user_id	int (FK)	User who owns the subscription.
plan_id	int (FK)	Subscribed plan.
start_date	datetime	Start of the subscription.
end_date	datetime	End of the subscription.
status	enum	Status: active, cancelled, expired.
payment_method	varchar(50)	Method used for payment (e.g., card, PayPal).
auto_renew	boolean	Whether auto-renew is enabled.
created_at	datetime	Subscription creation timestamp.

Payments

Attribute	Type	Description
id	int (PK)	Payment identifier.
subscription_id	int (FK)	Related subscription.
amount	decimal(10,2)	Amount paid.
currency	varchar(10)	Currency used (e.g., COP, USD).
payment_date	datetime	When the payment was made.
payment_status	enum	e.g., successful, failed, refunded.
payment_gateway	varchar(100)	Gateway used (e.g., Stripe, PayPal).
transaction_reference	varchar(100)	ID from the payment processor.
created_at	datetime	When the record was created.

Entity Relationship Matrix – Transactions Component

This matrix shows the relationships between entities in the Transactions component.

Entity	Plans	Subscriptions	Payments
Plans		N:1	
Subscriptions	1:N		N:1
Payments		1:N	

Legend:

- **1:N** – One-to-many
- **N:1** – Many-to-one

2.7. Security & Audit Logging

This component ensures that the platform is secure, and that user actions are traceable. All login attempts, data updates, and suspicious behaviors are logged. These logs are used not only for debugging or analysis but also to comply with data governance standards.

UserActivityLogs

Attribute	Type	Description
id	int (PK)	Unique identifier for the log entry.
user_id	int (FK)	User who performed the action.
action	varchar(255)	Short description of the action taken.
ip_address	varchar(100)	IP address from which the action originated.
device	varchar(100)	Device or browser used.
location	varchar(150)	Geolocation data, if available.
created_at	datetime	Timestamp of the action.

LoginAttempts

Attribute	Type	Description
id	int (PK)	Unique ID for each login attempt.
user_id	int (FK)	User attempting to log in.
email	varchar(100)	Email used in the login attempt.
ip_address	varchar(100)	Originating IP address.
device	varchar(100)	Device used in the login attempt.
success	boolean	Whether the login was successful.
attempt_time	datetime	Time of the login attempt.

2.7.1 Entity Relationship Matrix – Security & Audit Logging Component

This matrix shows the relationships between entities in the Security & Audit Logging component.

Entity	UserActivityLogs	LoginAttempts
UserActivityLogs		
LoginAttempts		

Note: Both entities have a `N:1` relationship with `Users` (User Management component).

Cross-Component Entity Relationship Types

This matrix shows relationships between entities that belong to different components. Each row includes the relationship type between the two entities.

Entity A	**Entity B**	**Type**	**Relation Description**
Users	Subscriptions	1:N	A user may subscribe to multiple plans.
Users	UserNotifications	1:N	Users receive multiple personalized notifications.
Users	LoginAttempts	1:N	Each login attempt is linked to a user.
Users	UserActivityLogs	1:N	User actions are tracked in logs.
Users	TripRecords	1:N	Trips can be associated with specific users.
TrafficEvents	UserNotifications	1:N	A traffic event may trigger many user notifications.
TrafficEvents	ExternalDataSources	N:1	Each traffic event originates from a specific external data source.

Legend:

- **1:N** – One-to-many
- **N:1** – Many-to-one

3. ER

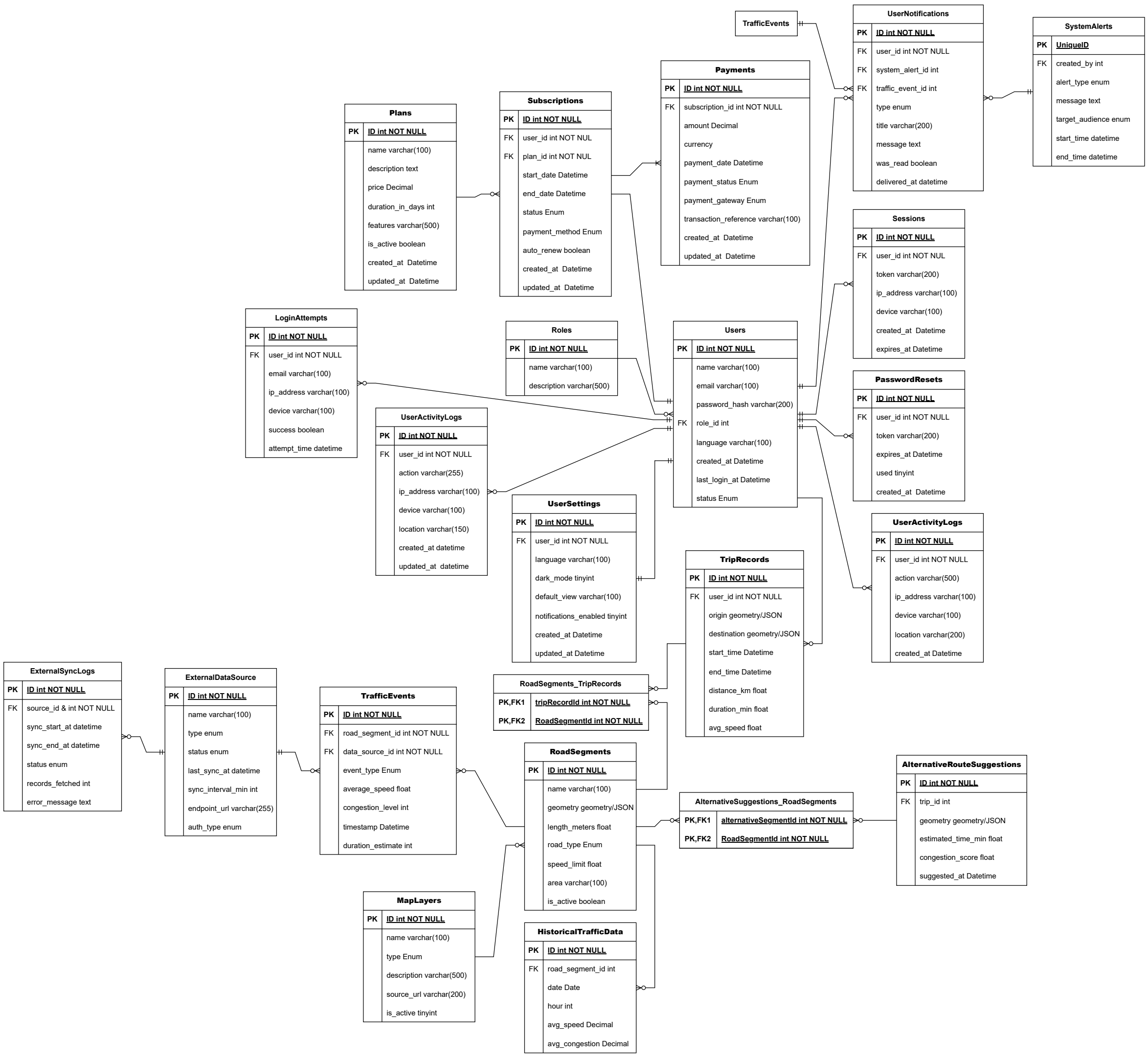

Figure 2: Entity Relation Model

4. System Architecture Overview

Figure 3 illustrates a high-level architecture model divided into three main subsystems, as described in the problem statement.

There are three primary data sources:

- External APIs queried for live or scheduled data.
- Open data platforms (OpenData Info) offering publicly available datasets.
- Our own application, *SimWaze*, which provides additional internal data.

All external data is collected and stored in a central MongoDB instance within a staging area. This NoSQL database also logs synchronization events, recording when data was accessed or updated.

An ETL (Extract, Transform, Load) process is responsible for converting this unstructured and semi-structured data into a structured relational format. The processed data is then stored in a PostgreSQL-based data warehouse, organized into three key entities:

- **Roads**: Contains data related to traffic infrastructure.
- **User Management**: Manages user profiles and access control.
- **Transactions**: Records transactional activities such as subscription purchases.

These entities are exposed to the backend for queries and data retrieval.

To improve performance, particularly for frequent queries to the **Roads** entity, a lightweight map service is deployed using Apache and supported by a Redis-based datamart. This helps offload read operations from the main database and reduces latency.

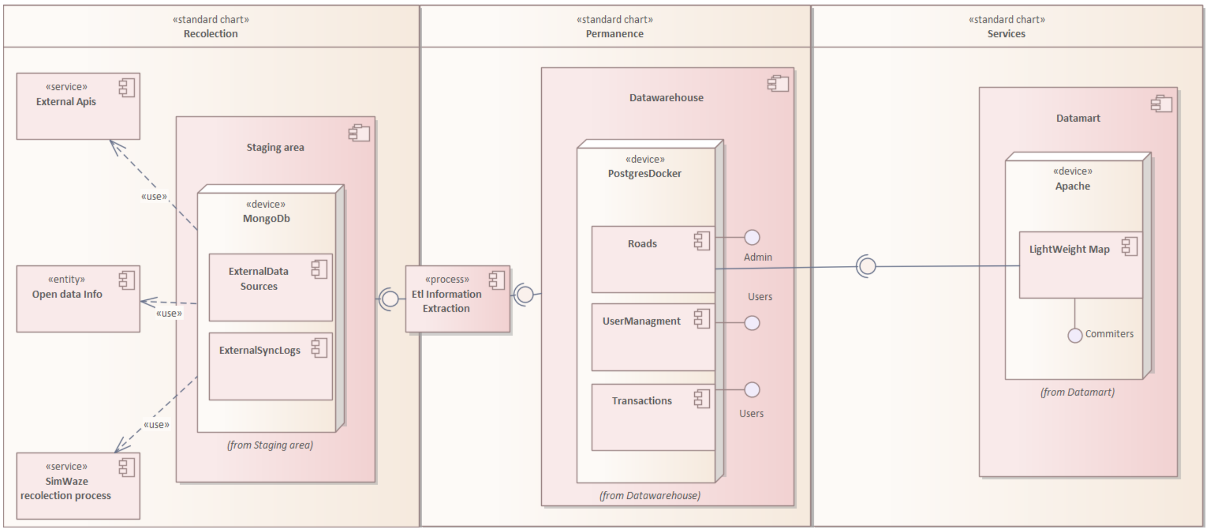

Figure 3: Entity-Relationship Architecture Model showing the data flow and subsystem components

Query Proposal for Information Requirements

Information Requirement: Real-Time Traffic Data

Purpose: To retrieve the most recent traffic conditions across road segments, providing commuters with up-to-date travel information.

SQL Query:

```
SELECT rs.name AS road_segment,
       te.level AS congestion_level,
       te.reported_at
FROM TrafficEvents te
JOIN RoadSegments rs ON te.road_segment_id = rs.id
WHERE te.reported_at > NOW() - INTERVAL '5 minutes'
ORDER BY te.reported_at DESC;
```

Explanation: This query returns the latest traffic congestion data by joining 'TrafficEvents' with 'RoadSegments', ordered by report time. It is used to feed the commuter UI with real-time conditions.

NoSQL Source (MongoDB):

Before ETL processing, traffic data may arrive in a raw collection like 'externalTrafficLogs'. A typical document might look like:

```
{
  "source": "Google Maps",
  "segment_id": "AV123",
  "congestion": "High",
  "timestamp": "2025-05-29T12:47:00Z"
}
```

MongoDB Query (pre-ETL stage):

```
db.externalTrafficLogs.find({
  timestamp: { $gte: ISODate("2025-05-29T12:42:00Z") }
}).sort({ timestamp: -1 })
```

Explanation: This NoSQL query is used before ETL during the ingestion phase. It helps validate the freshness and format of the incoming traffic data.

In our architecture, NoSQL is used in the *staging area* for flexible ingestion and rapid insertion. SQL is used post-ETL for reliable, structured access in production services.

Information Requirement: Route Alternatives and Recommendations

Purpose: To retrieve multiple optimized route suggestions for a given trip, using historical data and current traffic conditions.

SQL Query:

```
SELECT ars.id AS suggestion_id,
       STRING_AGG(rs.name, ' → ') AS route_path,
```

```
       ars.estimated_duration,
       ars.created_at
FROM AlternativeRouteSuggestions ars
JOIN AlternativeRouteRoadSegment arrs ON ars.id = arrs.route_id
JOIN RoadSegments rs ON arrs.segment_id = rs.id
WHERE ars.trip_id = 42
GROUP BY ars.id, ars.estimated_duration, ars.created_at
ORDER BY ars.estimated_duration ASC;
```

Explanation: This query retrieves all alternative routes for a given trip, builds the path as a string (ordered externally), and displays estimated durations. The system can use this data to rank options and suggest the best one.

Optional MongoDB Query (raw ingestion example):

```
db.routeSuggestions.find({
  trip_id: 42
}).sort({ estimated_duration: 1 })
```

Explanation: This NoSQL query might be used early in the ETL process to fetch raw route suggestions from external APIs (like Google or Waze), before transforming and persisting them into the relational schema.

The platform leverages NoSQL during data collection and preprocessing, but structured SQL queries in the warehouse are used to serve optimized route suggestions to end users.

Information Requirement: Transit Arrival Times

Purpose: To allow users to plan their trips efficiently based on the estimated arrival times of public transport (buses, trains) at a specific station or route.

SQL Query (Relational – PostgreSQL):

```
SELECT ts.name AS station_name,
       tl.name AS line_name,
       ta.estimated_arrival,
       ta.direction
FROM TransitArrivals ta
JOIN TransitStations ts ON ta.station_id = ts.id
JOIN TransitLines tl ON ta.line_id = tl.id
WHERE ts.name = 'Calle 72'
  AND ta.estimated_arrival >= NOW()
ORDER BY ta.estimated_arrival ASC
```

```
LIMIT 5;
```

Explanation: This query shows the next 5 expected arrivals for a specific station (‘Calle 72‘), including line name and direction. It supports the user need to check transport schedules in real time (US3).

MongoDB Query (Pre-ETL ingestion):

```
db.transitETAs.find({
  station_name: "Calle 72",
  timestamp: { $gte: ISODate("2025-05-29T14:00:00Z") }
}).sort({ estimated_arrival: 1 }).limit(5)
```

Explanation: Used during the ingestion phase to gather arrival estimates from external APIs like TransMilenio or OpenMobilityData. These documents are later structured in the SQL database via ETL.

Transit data flows from NoSQL ingestion into SQL-based analytics to ensure both flexibility and performance for time-sensitive queries.

Note: Entities such as `TransitArrival`, `TransitStation`, and `TransitLine` were not explicitly modeled in the current version of the data architecture. However, their inclusion is relevant for supporting multimodal transportation features (e.g., bus/train schedules). These entities will be considered in a future iteration of the data model, where integration with urban mobility datasets will be explored in greater detail.

Information Requirement: Personalized User Preferences

Purpose: To personalize the user experience based on accessibility needs, transport preferences, and historical behavior.

SQL Query:

```
SELECT u.name,
       us.language,
       us.dark_mode,
       us.transport_mode,
       COUNT(tr.id) AS total_trips,
       MAX(tr.created_at) AS last_trip
FROM Users u
JOIN UserSettings us ON u.id = us.user_id
LEFT JOIN TripRecords tr ON u.id = tr.user_id
WHERE u.id = 101
GROUP BY u.name, us.language, us.dark_mode, us.transport_mode;
```

Explanation: This query combines UI settings with preferred transportation mode and trip history. It is used to customize route suggestions and adjust the interface for accessibility and language preferences.

MongoDB Query (Pre-ETL, onboarding phase):

```
db.userProfileRaw.findOne({
  user_id: 101
})
```

Example document:

```
{
  "user_id": 101,
  "preferences": {
    "language": "es",
    "dark_mode": true,
    "transport_mode": "bike"
  },
  "source": "mobile-app"
}
```

Explanation: This document may be inserted by the mobile app during user onboarding. These values are later transformed and loaded into the relational ‘UserSettings‘ table.

User preferences are first captured via NoSQL during initial sessions or form submissions, then structured in SQL for use in personalization, analytics, and session continuity.

Information Requirement: Environmental Metrics

Purpose: To promote sustainable travel by suggesting eco-friendly routes and tracking the environmental performance of user mobility patterns.

SQL Query:

```
SELECT ars.id AS route_id,
       ars.estimated_duration,
       ars.eco_score,
       STRING_AGG(rs.name, ' → ') AS route_path
FROM AlternativeRouteSuggestions ars
JOIN AlternativeRouteRoadSegment arrs ON ars.id = arrs.route_id
JOIN RoadSegments rs ON arrs.segment_id = rs.id
WHERE ars.trip_id = 101
ORDER BY ars.eco_score DESC
```

```sql
LIMIT 3;
```

Explanation: This query returns the top 3 most eco-friendly alternative routes for a given trip, showing their path and estimated duration. The `eco_score` field allows ranking based on sustainability, supporting user story US8.

Note: The attribute `eco_score` used in this query is not yet explicitly mapped in the current data model. It represents a calculated metric used to rank route suggestions based on their environmental impact (e.g., congestion, route length, elevation, preferred transport mode). In a future iteration of the data architecture, this field will be formally incorporated into the `AlternativeRouteSuggestions` entity, along with a defined scale and methodology for its computation.

SQL Query (User Analytics – Premium Feature):

```sql
SELECT u.name,
       COUNT(tr.id) AS total_trips,
       AVG(ars.eco_score) AS avg_eco_score
FROM Users u
JOIN TripRecords tr ON u.id = tr.user_id
JOIN AlternativeRouteSuggestions ars ON tr.id = ars.trip_id
WHERE u.subscription_plan = 'premium'
GROUP BY u.name;
```

Explanation: Premium users get access to sustainability analytics. This query shows their average eco score across all trips, enabling personalized insights on their environmental impact.

Environmental metrics are computed as part of the route optimization process and stored in the warehouse. Premium users benefit from visibility into their travel behavior and its ecological footprint.

Information Requirement: Administrative Insights

Purpose: To provide urban mobility authorities and administrators with access to traffic patterns and congestion metrics, allowing informed decision-making and planning.

SQL Query

```sql
SELECT rs.name AS road_segment,
       DATE_TRUNC('hour', te.reported_at) AS hour,
       COUNT(*) AS event_count,
```

```
       AVG(te.level) AS avg_congestion
FROM TrafficEvents te
JOIN RoadSegments rs ON te.road_segment_id = rs.id
WHERE te.reported_at BETWEEN '2025-05-20' AND '2025-05-27'
GROUP BY rs.name, hour
ORDER BY rs.name, hour;
```

Explanation: This query returns the average congestion level and number of traffic events per road segment, grouped by hour over a week. It feeds administrative dashboards that monitor trends and identify bottlenecks.

MongoDB Query (Pre-ETL, raw ingestion check):

```
db.externalTrafficLogs.aggregate([
  { $match: { timestamp: { $gte: ISODate("2025-05-27T00:00:00Z") } } },
  { $group: {
      _id: "$segment_id",
      count: { $sum: 1 },
      avg_level: { $avg: "$congestion_level" }
  }},
  { $sort: { count: -1 } }
])
```

Explanation: Used before ETL to verify incoming traffic logs. Helps detect segments with high reporting frequency or abnormal congestion levels prior to data integration into the warehouse.

Real-time logs are processed via NoSQL for ingestion validation, while structured SQL reports provide deep analytics and historical views for administrators, supporting key business partners and contracts.

Information Requirement: Usage and Subscription Tracking

Purpose: To deliver personalized insights and ensure correct access to premium features, including ad-free experience and trip statistics for paying users.

SQL Query:

```
SELECT u.name,
       p.name AS plan_name,
       COUNT(tr.id) AS total_trips,
       MAX(tr.created_at) AS last_trip,
```

```
       s.auto_renew,
       s.start_date,
       s.end_date
FROM Users u
JOIN Subscriptions s ON u.id = s.user_id
JOIN Plans p ON s.plan_id = p.id
LEFT JOIN TripRecords tr ON u.id = tr.user_id
WHERE p.name = 'Premium'
GROUP BY u.name, p.name, s.auto_renew, s.start_date, s.end_date;
```

Explanation: This query provides a profile for each premium user, including plan name, trip history, and subscription status. It supports premium analytics and enables user-specific dashboards.

MongoDB Query (Pre-ETL, optional tracking):

```
db.userSessionLogs.find({
  plan: "Premium",
  ads_displayed: false
}).limit(5)
```

Explanation: Used during onboarding or session tracking to confirm that no ads are shown to premium users. This data may be discarded or summarized during ETL but is useful for debugging and validation.

Subscription tracking is fully managed in SQL for performance and historical analysis. NoSQL may assist in validating runtime behavior before structured aggregation.