Concordia University

SOEN 6461 Software Design Methodologies

Assignment 2

Group members: Duy Thanh Phan – 40269832
Omnia Alam – 40261762

Table of Contents

1. Requirements Analysis:

1.1. Functional Requirements:

<u>**Traffic Monitoring and Control Subsystem:**</u>

- **Real-time Traffic Data Collection:** The system should be able to collect real-time data from IoT devices such as street cameras, road sensors, and traffic signals.
- **Traffic Analytics:** Analyze the collected traffic data to identify traffic congestion, accidents, and other anomalies.
- **Traffic Signal Control:** Adjust traffic signals based on predefined schedules and in real-time to optimize traffic flow.
- **Traffic Data Visualization:** Provide user-friendly dashboards for traffic operators and city officials to monitor traffic in real-time and view historical data.
- **Traffic Data Storage:** Store historical traffic data for analysis, periodical reports, and future planning.

<u>**Public Transport Management Subsystem:**</u>

- **Real-time Transit Tracking:** Provide real-time or near real-time tracking of public transport vehicles.
- **Itinerary Planning:** Use algorithms via various criteria such as traffic, stops and demands to provide optimized routes and schedules to reduce travel time and congestion.
- **Fare Solution:** Calculate electronic fare based on the itinerary and provide payment solution.
- **Schedule Information:** Monitor and update the system with information on public transit services, such as bus/train locations, schedules, service interruptions, and delays. Inform the users of critical changes in real-time.
- **Integration with Traffic Control:** Coordinate public transport routes with traffic signal adjustments for smoother traffic flow.

<u>**Rider Experience Subsystem:**</u>

- **Journey Planner:** Offer travel options for residents to plan their journeys using a combination of public transport and private vehicles based on real-time traffic and preferences.
- **Mobile Ticketing:** Allow passengers to purchase tickets, passes, or access public transport services using mobile devices.
- **Real-time Updates:** Provide real-time updates on public transport arrivals, service interruptions and delays, and traffic conditions.
- **User Feedback:** Collect and analyze user feedback to continuously improve the rider experience.

<u>**Interrelation between the Subsystems:**</u>

The Traffic Monitoring and Control subsystem should feed real-time traffic data to both the Public Transport Management and Rider Experience subsystems to make informed decisions about routes and traffic conditions.

Public Transport Management should coordinate with Traffic Monitoring to optimize routes based on real-time traffic data.

Rider Experience should provide real-time updates to users based on data received from both Traffic Monitoring and Public Transport Management to ensure passengers receive accurate information for planning their journeys.

1.2. Non-Functional Requirements:

For Traffic Operators / Management Side:

- **Scalability:** A vast number of IoT devices are going to send consecutive small network packets (for sensors), or visual feed (for cameras) to the system in real time. This prompts a requirement to ensure efficient data processing and storage scalability to accommodate increasing data volumes. The system should be able to handle a growing number of IoT devices and users as the city expands.
  - For IoT sensors, the system should be able to handle a minimum of 30,000 devices, with each device sending 30 network packets in a minute.
  - For cameras, the system should handle at least 10,000 cameras with 720p video resolution at 30 frames per second.
  - For human operators, the system should be capable of handling a peak load of 30,000 operators to work at the same time.
- **Availability and Fault Tolerance:** To meet the expectation of the customers, high availability should be implemented. The system should be accessible 24/7 to support real-time traffic management. Implement redundancy and failover mechanisms to minimize downtime. To do this requirement, the system should have an uptime of 99.5% and aim for greater reliability. It is possible to increase the uptime to 99.9%, but achieving it can be quite costly, so a compromise was provided.
- **Performance:** In terms of performance, the aim is to guarantee a page load time of 700 milliseconds for operators at the 99th percentile. Considering that the data coming from different sensors are coming from different formats, the latency over the network of incoming data from sensors and cameras should not exceed 2 seconds at the 90th percentile. These data should be visible within the system within a few minutes at most.
- **Data Consistency:** Since the data is being streamed from sensors to the system in real-time, it is acceptable to lose a couple of seconds of data as long as the user can access the system. Therefore, we favor availability over consistency. However, the lag in the stream data would make it less effective in monitoring, so we should strive to reduce the latency as low as possible to guarantee an acceptable level of data consistency.
- **Security:** Implement strong data encryption and access control measures to protect sensitive traffic data and user information. Regular security audits and updates to guard against cyber threats.

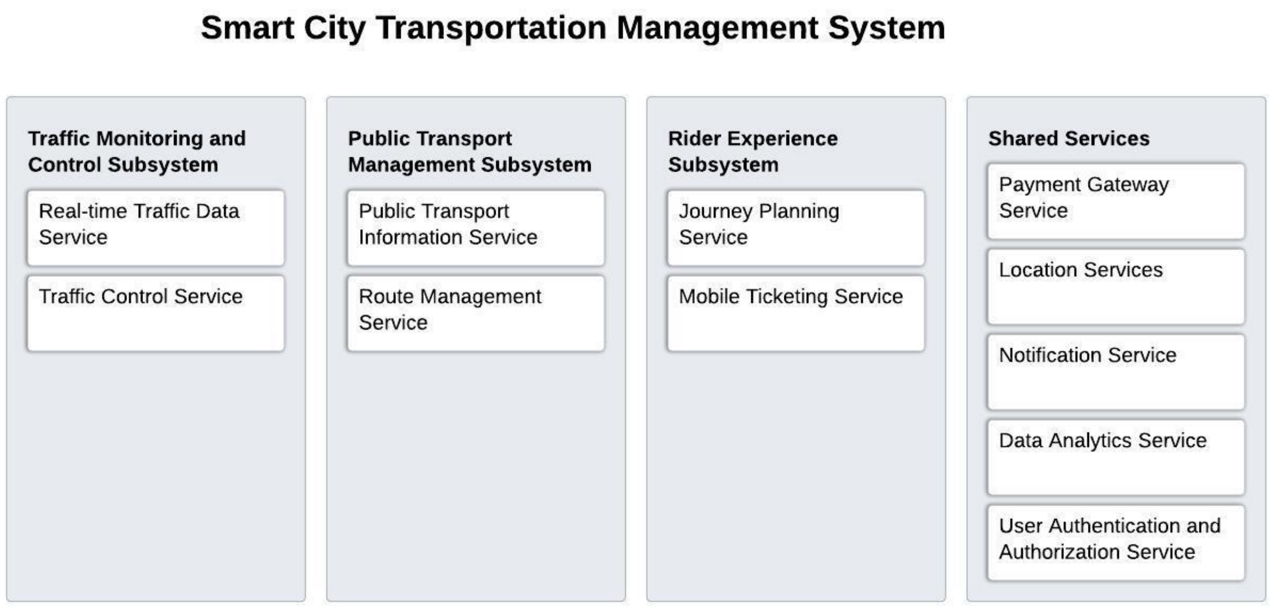
For Passengers Side:

- **Scalability:** Many users will query the system to get the latest traffic information, get travel guidance, and pay for the service. For the scale of 5 million citizens in Technopolis, the system should be able to handle a peak load of hundreds of thousands of passengers accessing the service at the same time.
- **Availability and Fault Tolerance:** To meet the expectation of the passengers, the system for passengers should have high availability. To achieve this requirement, the system should have an uptime of 99.5% and aim for greater reliability. For the payment and ticketing system, a higher rate of 99.9% is required because it directly affects the travel experience of the end-users.

- **Performance:** Due to the nature of travelling, users on the go will mostly use the mobile applications. For standard queries such as querying bus schedules, the system should return a response from 500 milliseconds. Complex requests such as travel planning require generating the best itinerary based on the current traffic conditions may take longer but not longer than 5 seconds.
- **Data Consistency:** Regarding data consistency for passengers, the system adopts a strategy that prefers availability. It is important that the user can have information, albeit not up-to-date, rather than having nothing at all.
- **Security:** Access control measures should be implemented to protect sensitive traffic data and user information.

2. System Design:

2.1. Service-Oriented Design:


Smart City Transportation Management System
Traffic Monitoring and Control Subsystem
Real-time Traffic Data Service
Traffic Control Service
Public Transport Management Subsystem
Public Transport Information Service
Route Management Service
Rider Experience Subsystem
Journey Planning Service
Mobile Ticketing Service
Shared Services
Payment Gateway Service
Location Services
Notification Service
Data Analytics Service
User Authentication and Authorization Service

<u>Traffic Monitoring and Control Subsystem:</u>

Real-time Traffic Data Service:

- Responsible for collecting, storing, and serving real-time traffic data.
- Exposes RESTful APIs for retrieving and updating traffic data.

Traffic Control Service:

- Manages traffic signals and devices based on real-time data.
- Offers RESTful APIs for controlling traffic signals and devices.

<u>Public Transport Management Subsystem:</u>

Public Transport Information Service:

- Manages public transport information and schedules.
- Provides RESTful APIs for retrieving public transport data.

Route Management Service:

- Handles the creation, updating, and deletion of public transport routes.
- Exposes RESTful APIs for route management.

Rider Experience Subsystem:

Journey Planning Service:

- Responsible for planning journeys based on user preferences and real-time data.
- Offers RESTful APIs for journey planning.

Mobile Ticketing Service:

- Manages ticket purchasing and retrieval.
- Provides RESTful APIs for purchasing and viewing tickets.

Shared Services:

User Authentication and Authorization Service:

- Handles user authentication and authorization for all subsystems.
- Ensures that only authorized users can access the services.

Data Analytics Service:

- Performs data analysis on traffic and public transport data to provide insights and optimizations.
- Offers APIs for data analytics requests.

Notification Service:

- Sends notifications and alerts to users regarding traffic conditions, route changes, or ticket updates.
- Exposes APIs for sending notifications.

Location Services:

- Provides geospatial data and services for location-based features in journey planning and traffic monitoring.

Payment Gateway Service:

- Facilitates payment processing for mobile ticketing.

2.2. Microservices Design for Subsystem Scalability:

Traffic Monitoring and Control Microservices:

- Real-time Data Collection Microservice
- Traffic Analytics Microservice
- Traffic Signal Control Microservice

Public Transport Management Microservices:

- Route Optimization Microservice
- Real-time Tracking Microservice
- Fare Collection Microservice

Rider Experience Microservices:

- Journey Planning Microservice
- Mobile Ticketing Microservice
- User Feedback Microservice

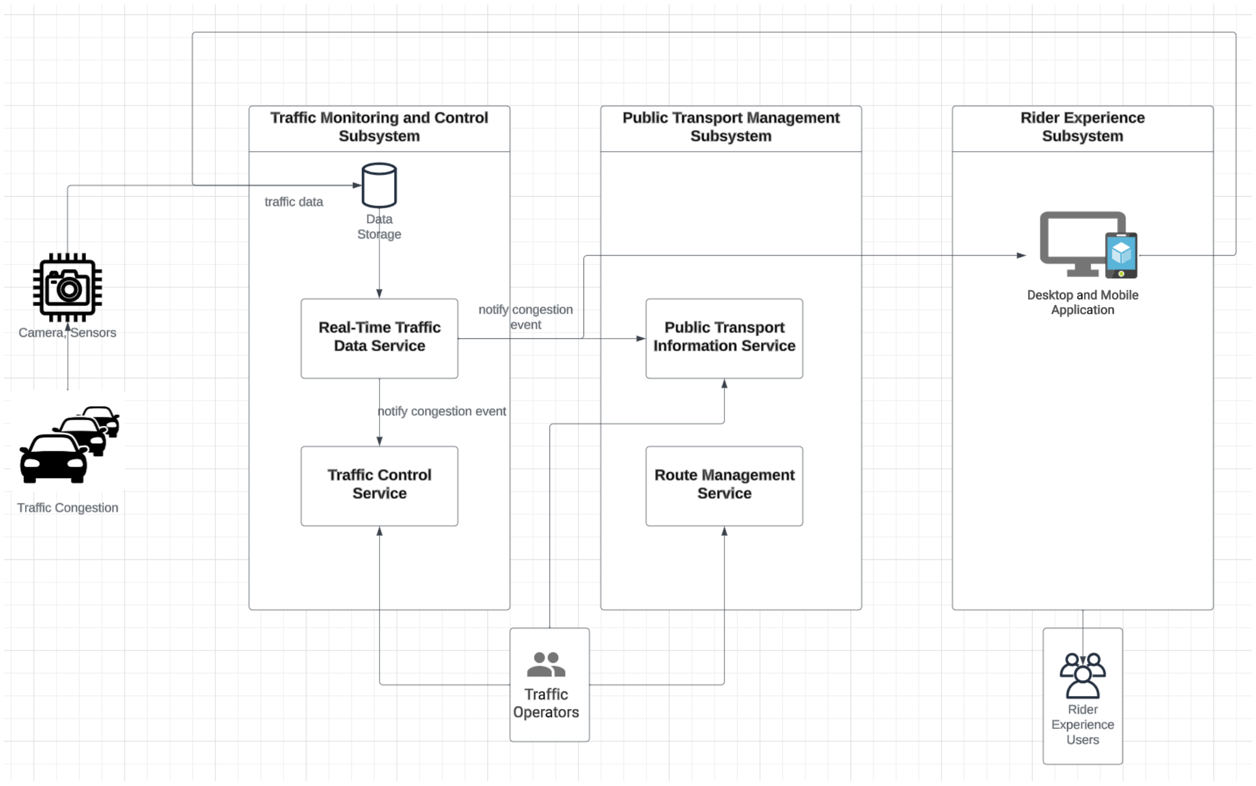
Interrelation Between Microservices:

- Each microservice should have a well-defined API for communication with other microservices.
- Microservices should be containerized and orchestrated using tools like Docker and Kubernetes for easy scaling and management.
- Implement API gateways and service discovery for routing requests to the appropriate microservice.

2.3. Prototype's Essential Functionalities:

Since the TMS includes various use cases in each subsystem, this document would only describe a core scaled-down scenario that is Traffic Congestion Mitigation

1. A traffic jam occurs at a specific road. The recording traffic data is sent from IoT devices and from passengers' devices that install the Rider Experience app on that road to the Traffic Monitoring and Control subsystem for analysis.
2. Based on various image recognition algorithm criteria and historical data analysis, the Traffic Monitoring and Control subsystem detects the congestion and record in the system.
3. Via an event-driven implementation, the event of the congestion is sent to the Public Transport Management subsystem in real-time. There are two possibilities here. The transport management system can make decisions to change the traffic lights automatically, or traffic operators could, based on the event and related analysis, make manual decisions such as deploying traffic control officers to the congestion site or redirecting the public transport vehicles using a new optimized route to avoid the congestion.
4. Using the same event-driven implementation, the congestion event is sent to subscribed passengers using the Rider Experience application in the nearby areas. As the event of congestion is recorded in the system, it can be used to account for optimized routes for passengers planning to travel to or out of the affected area.

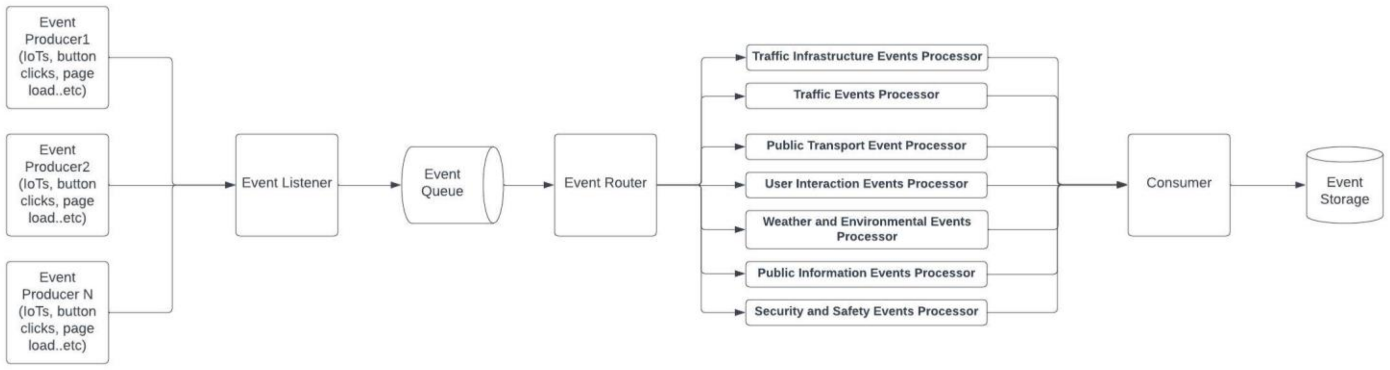

Traffic Monitoring and Control Subsystem
Public Transport Management Subsystem
Rider Experience Subsystem
traffic data
Data Storage
Camera, Sensors
Traffic Congestion
Real-Time Traffic Data Service
notify congestion event
notify congestion event
Traffic Control Service
Public Transport Information Service
Route Management Service
Desktop and Mobile Application
Traffic Operators
Rider Experience Users

2.4. An Event-Driven Design:

In order implement the event-Driven Design, first we need to identify who are the event sources and whose will be the consumers. Then we must design

- **Event Sources:** These sources may include IoT devices (e.g., sensors, button click, cameras), user interactions, and external data providers (e.g., weather services).
- **Event Listener:** if there are any event changes the event listener will be called and pass them to the event queue.
- **Event Broker/Event Queue**: we need to implement a central event broker or message queue, such as Apache Kafka, RabbitMQ, or a similar technology. This component will serve as the central hub for collecting and distributing events.
- **Define Events**: We need to define event types. Which can be an Enum class.
- **Event Producers:** IoT devices can produce traffic-related events.
- **Event Consumers:** we may have consumers responsible for traffic signal adjustments, public transport route optimization, user notification, and more.
- **Event Processing:** Event consumers will process incoming events, which may involve data analysis, decision-making, and taking appropriate actions. For example, if a "Traffic Congestion Detected" event is received, the consumer may trigger traffic signal adjustments to alleviate congestion. In this project, we must create controllers for each of the events to handle the event process.
- **Event Storage:** to store historical events for auditing, analysis, and troubleshooting purposes. Event storage can be useful for tracking system behavior and diagnosing issues.

A typical workflow diagram for the Event-Driven Diagram:

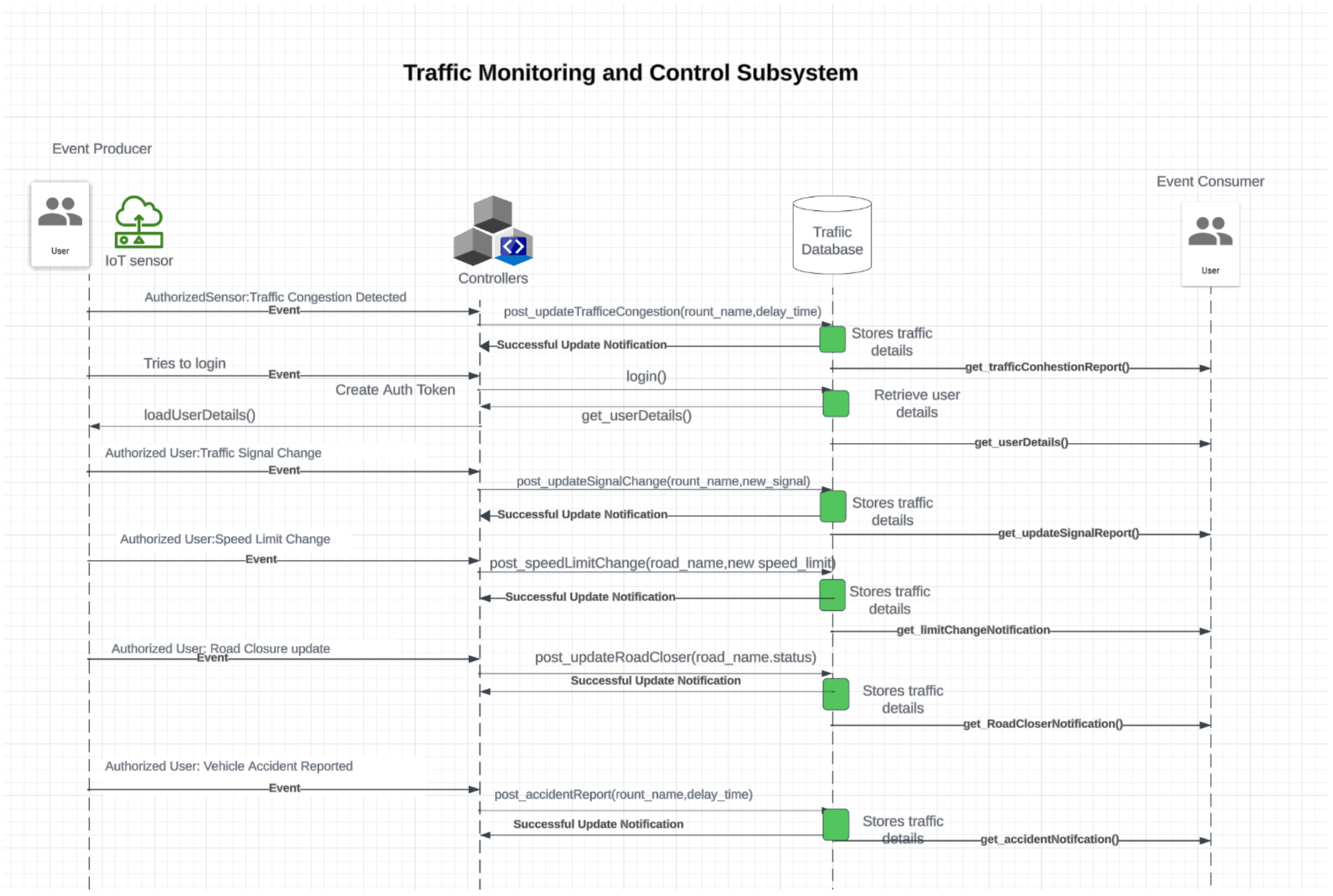

Event Producer1 (IoTs, button clicks, page load..etc)
Event Producer2 (IoTs, button clicks, page load..etc)
Event Producer N (IoTs, button clicks, page load..etc)
Event Listener
Event Queue
Event Router
Traffic Infrastructure Events Processor
Traffic Events Processor
Public Transport Event Processor
User Interaction Events Processor
Weather and Environmental Events Processor
Public Information Events Processor
Security and Safety Events Processor
Consumer
Event Storage

List of possible Event Types and Processors:

1. Traffic Events Processor:

- Traffic Congestion Detected
- Vehicle Accident Reported
- Road Closure Alert
- Traffic Signal Change
- Speed Limit Change
- Road Closure Notification

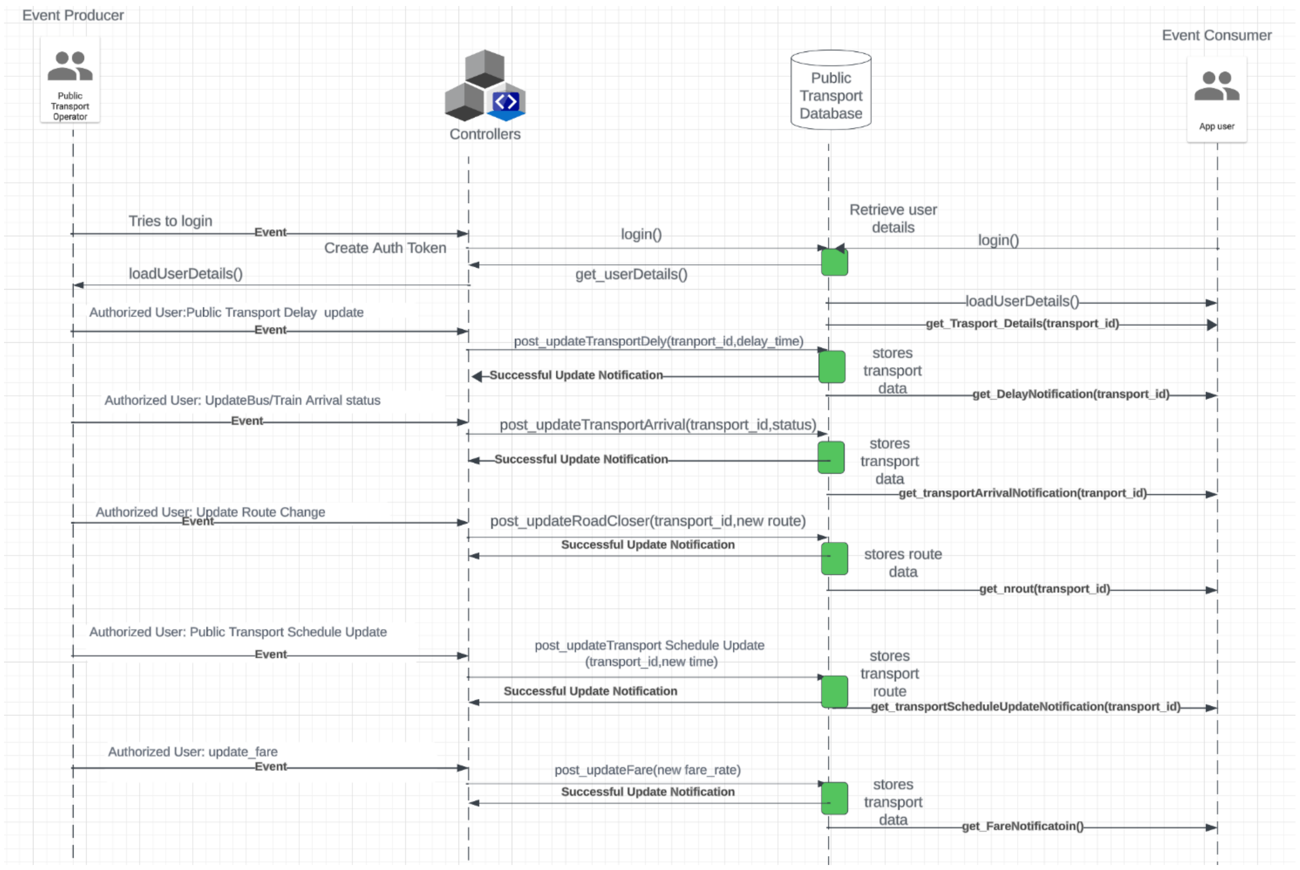
A proper sequence diagram below will explain the event sequence more clearly based on the traffic control event types:


Traffic Monitoring and Control Subsystem
Event Producer
User
IoT sensor
Controllers
Trafiic Database
Event Consumer
User
AuthorizedSensor:Traffic Congestion Detected
Event
post_updateTrafficeCongestion(rount_name,delay_time)
Successful Update Notification
Stores traffic details
get_trafficConhestionReport()
Tries to login
Event
Create Auth Token
login()
Retrieve user details
loadUserDetails()
get_userDetails()
get_userDetails()
Authorized User:Traffic Signal Change
Event
post_updateSignalChange(rount_name,new_signal)
Successful Update Notification
Stores traffic details
get_updateSignalReport()
Authorized User:Speed Limit Change
Event
post_speedLimitChange(road_name,new speed_limit)
Successful Update Notification
Stores traffic details
get_limitChangeNotification
Authorized User: Road Closure update
Event
post_updateRoadCloser(road_name.status)
Successful Update Notification
Stores traffic details
get_RoadCloserNotification()
Authorized User: Vehicle Accident Reported
Event
post_accidentReport(rount_name,delay_time)
Successful Update Notification
Stores traffic details
get_accidentNotifcation()

2. Public Transport Events Processor:

- Public Transport Delay
- Bus/Train Arrival Notification
- Route Change Request
- Public Transport Schedule Update
- Fare Adjustment Announcement

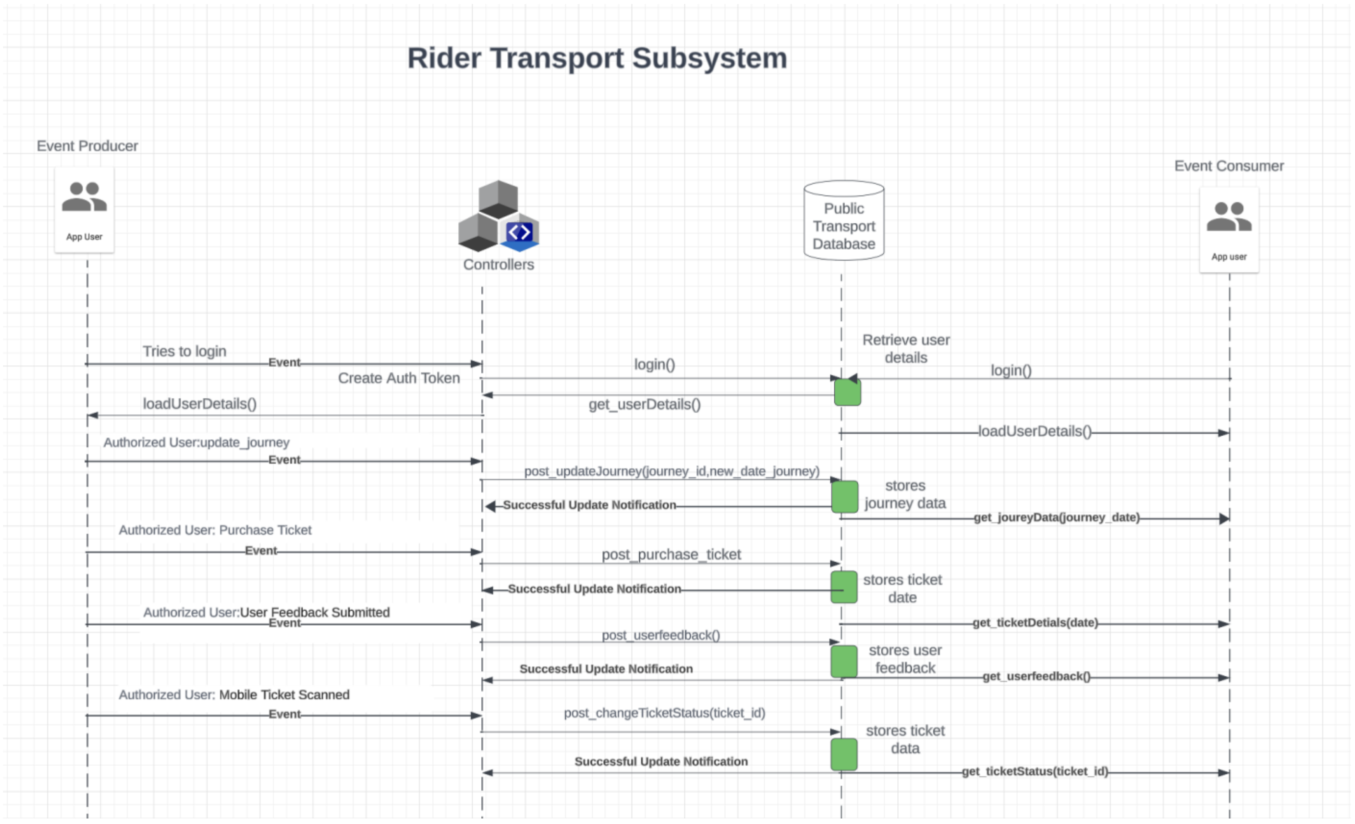
A proper sequence diagram below will explain the event sequence more clearly based on the public transport subsystem's event types:


Event Producer
Public Transport Operator
Controllers
Public Transport Database
Event Consumer
App user
Tries to login
Event
Create Auth Token
login()
Retrieve user details
login()
loadUserDetails()
get_userDetails()
loadUserDetails()
get_Trasport_Details(transport_id)
Authorized User:Public Transport Delay update
Event
post_updateTransportDely(tranport_id,delay_time)
Successful Update Notification
stores transport data
get_DelayNotification(transport_id)
Authorized User: UpdateBus/Train Arrival status
Event
post_updateTransportArrival(transport_id,status)
Successful Update Notification
stores transport data
get_transportArrivalNotification(tranport_id)
Authorized User: Update Route Change
Event
post_updateRoadCloser(transport_id,new route)
Successful Update Notification
stores route data
get_nrout(transport_id)
Authorized User: Public Transport Schedule Update
Event
post_updateTransport Schedule Update (transport_id,new time)
Successful Update Notification
stores transport route
get_transportScheduleUpdateNotification(transport_id)
Authorized User: update_fare
Event
post_updateFare(new fare_rate)
Successful Update Notification
stores transport data
get_FareNotificatoin()

3. Rider Experience Events Processor:

- User Journey Request
- Ticket Purchase Request
- Journey Plan Update
- User Feedback Submitted
- Mobile Ticket Scanned

A proper sequence diagram below will explain the event sequence more clearly based on the public transport subsystem's event types:


Rider Transport Subsystem
Event Producer
App User
Controllers
Public Transport Database
Event Consumer
App user
Tries to login
Event
Create Auth Token
login()
Retrieve user details
login()
loadUserDetails()
get_userDetails()
loadUserDetails()
Authorized User:update_journey
Event
post_updateJourney(journey_id,new_date_journey)
Successful Update Notification
stores journey data
get_joureyData(journey_date)
Authorized User: Purchase Ticket
Event
post_purchase_ticket
Successful Update Notification
stores ticket date
get_ticketDetials(date)
Authorized User:User Feedback Submitted
Event
post_userfeedback()
stores user feedback
Successful Update Notification
get_userfeedback()
Authorized User: Mobile Ticket Scanned
Event
post_changeTicketStatus(ticket_id)
stores ticket data
Successful Update Notification
get_ticketStatus(ticket_id)

2.5. API-First Design:

Based on the Event Driven diagram if we want to design all the APIs that we need to meet the functional requirements, here is the list of APIs endpoint, methos and error messages for each subsystem:

Traffic Monitoring and Control Subsystem:

Endpoints:

Real-time Traffic Data:

- GET /traffic-data: Retrieve real-time traffic data.
- GET / traffic-data: Road Closer Notification
- GET /traffic-data: Vehicle Accident Reported
- GET /traffic-data: Traffic Signal Change
- POST /traffic-data: Update traffic data (for authorized IoTs and users).
- POST /traffic-data: Vehicle Accident Reported
- POST /traffic-data: Road Closure Alert
- POST /traffic-data: Traffic Signal Change
- POST /traffic-data: Speed Limit Change

Methods:

- post_updateTrafficCongestion(road_name, delay_time)
- post_updateSignalChange(road_name, new_signal)

- get_userDetails(user_id)
- get_trafficConhestionReport()
- post_updateSignalChange(road_name, new_signal)
- get_updateSignalReport()
- post_speedLimitChange(road_name,new_speed_limit)
- get_updateSignalReport()
- post_updateRoadCloser(road_name, status)
- get_roadCloseNotification()
- post_accidentReport(road_name,delay_time)
- get_accidentNotifcation()

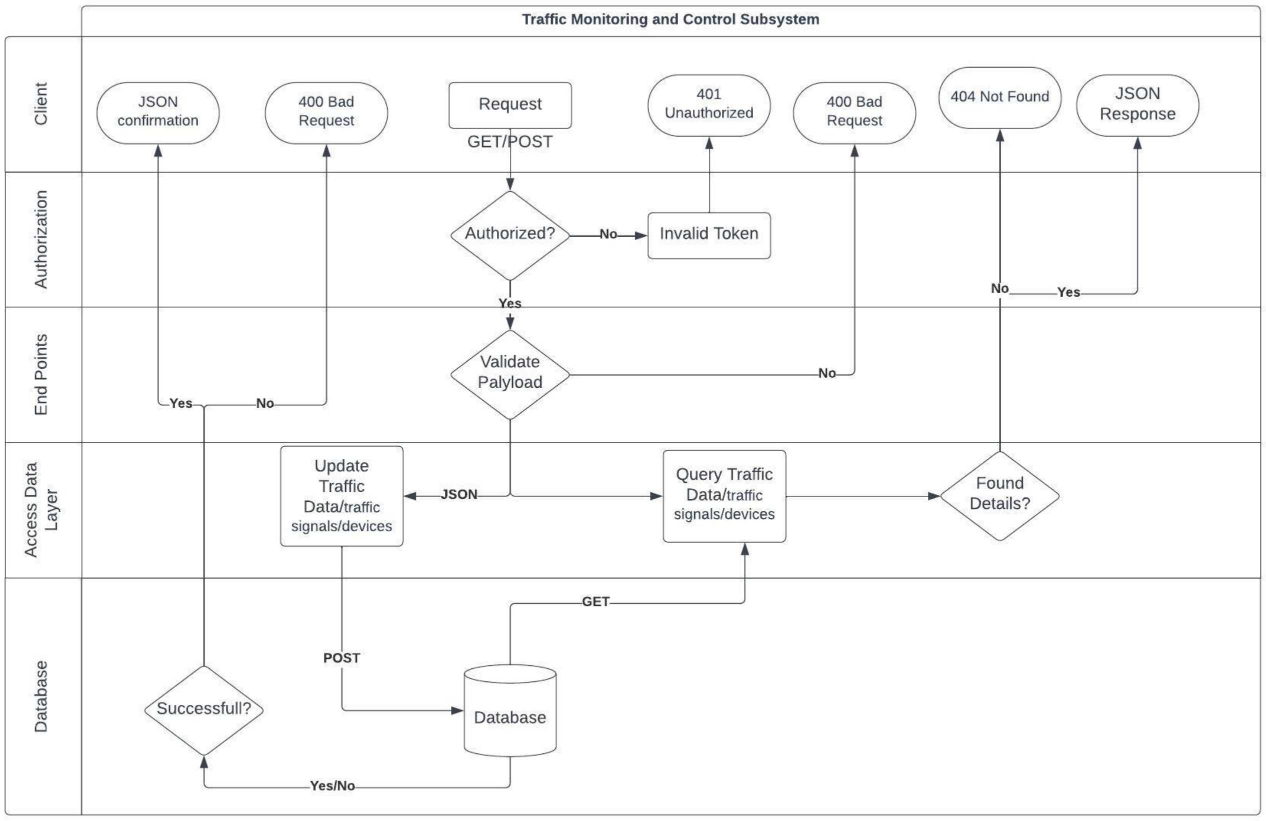
Request/Response Formats:

- GET /traffic-data:
  Request: None
  Response: JSON format with real-time traffic data.
- POST /traffic-data:
  Request: JSON format with data to update traffic conditions.
  Response: JSON confirmation message.
- POST /traffic-control:
  Request: JSON format to control traffic signals/devices.
  Response: JSON confirmation message.

Error Messages:

- 400 Bad Request: Invalid input.
- 401 Unauthorized: Lack of authorization.
- 404 Not Found: Endpoint not found.
- 500 Internal Server Error: Server issues.
- 200 Response Success

Showing a general workflow diagram for the Traffic Monitoring and Control Subsystem using the API Frist design concept:



Public Transport Management Subsystem:

Endpoints:

Public Transport Information:

- GET /public-transport-info: Retrieve public transport information.

Route, schedule, transportation Management:

- GET /routes: Retrieve available public transport routes.
- GET/ schedule: Retrieve transportation schedules
- POST /routes: Public Transport Delay
- GET /schedule: Bus/Transport Arrival Notification
- POST /routes: Route Change Request
- POST /schedule: Public Transport Schedule Update
- POST /transportation: Fare Adjustment Announcement

Methods:

- post_updateTransportDelay(transport_id, delay_time)
- get_DelayNotification(transport_id)
- post_updateTransportArrival(transport_id, status)
- get_transportArrivalNotification(tranport_id)

- post_updateRoadClose(transport_id, new_route)
- get_route(transport_id)
- post_updateTransportScheduleUpdate (transport_id, new_time)
- get_transportScheduleUpdateNotification(transport_id)
- post_updateFare(new_fare_rate)
- get_FareNotification()

Request/Response Formats:

- GET /public-transport-info:
  Request: None
  Response: JSON format with public transport information.
- GET /routes:
  Request: None
  Response: JSON format with available routes.
- POST /routes:
  Request: JSON format to create or update routes.
  Response: JSON confirmation message.

Error Messages:

- 400 Bad Request: Invalid input.
- 401 Unauthorized: Lack of authorization.
- 404 Not Found: Endpoint not found.
- 500 Internal Server Error: Server issues.
- 200 Response Success

Rider Experience Subsystem:

Endpoints:

Journey Planning:

- POST /plan-journey: Plan a journey for a user based on preferences and real-time data.
- GET /my-plan: User Journey Request

Mobile Ticketing:

- POST /purchase-ticket: Allow users to purchase tickets.
- GET /my-tickets: Retrieve purchased tickets for a user.
- POST /my-feedback: User Feedback Submitted
- POST /my-tickets: Mobile Ticket Scanned

Methods:

- post_updateJourney(journey_id, new_date_journey)
- get_journeyData(journey_date)
- post_purchase_ticket(ticket_details, date)
- get_ticketDetails(date)
- post_userfeedback(user_feedback)
- get_ticketDetails(date)

- post_changeTicketStatus(ticket_id)
- get_ticketStatus(ticket_id)

Request/Response Formats:

- POST /plan-journey:
  Request: JSON format with user preferences and origin-destination.
  Response: JSON format with journey details and routes.
- POST /purchase-ticket:
  Request: JSON format with ticket details.
  Response: JSON confirmation message.
- GET /my-tickets:
  Request: None
  Response: JSON format with purchased tickets.

Error Messages:

- 400 Bad Request: Invalid input.
- 401 Unauthorized: Lack of authorization.
- 404 Not Found: Endpoint not found.
- 500 Internal Server Error: Server issues.
- 200 Response Success

3. User Experience

3.1. User-Centered Design:

Traffic Monitoring and Control Subsystem:

Administrator Dashboard: used by system administrators

- To manage user accounts-
- To access system settings
- To configure and monitor the Traffic Monitoring and Control

Traffic Control Center UI: Used by traffic control operators

- To monitor real-time traffic data
- To control traffic signals and devices and respond to traffic incidents.

Traffic Data Analytics Dashboard: Utilized by data analysts

- To visualize and analyze traffic data
- To identify trends and make informed decisions for traffic management and optimization.

Traffic Camera Monitoring UI: Used by traffic camera operators

- To monitor live camera feeds and assess traffic conditions.

Public Transport Management Subsystem:

Public Transport Operator Dashboard: Utilized by public transport operators

- To manage routes, schedules, and vehicles.
- It may include features for assigning routes, monitoring vehicles, and handling passenger data.

Driver Console (Public Transport): used by the transport drivers

- to access route information, report issues, and communicate with passengers.

Rider Experience Subsystem:

Rider Experience Mobile App: used by the residents

- To plan journeys, purchase tickets, receive notifications,
- To access real-time information about public transport and traffic.
- A mobile application for citizens to report traffic incidents, accidents, or road hazards.

3.2. Responsive Design

Since the majority of the passengers will be using the Rider Experience app on the go, it is essential that the application is displayed properly on both desktop and mobile devices. To achieve that goal, we use the Responsive Design for the interface on mobile devices can be simpler and focused, supporting touch screens, whereas the desktop application contains more functionalities for power users.

Our consideration for desktop devices:

- Reduce the bundle size to be transmitted over the internet so that the users can see and interact with the page quicker.
- Ensure compatibility among the major browser vendors such as Chrome, Firefox and Safari.
- The page must have a reasonable layout. For instance, a reporting page where tabular data is used may need to support both vertical and horizontal scrolling. Meanwhile, a dashboard page does not need to support horizontal scrolling.
- Ensure validation logic is implemented both on the client side and server side to increase the responsiveness to errors of the system.
- Navigational links and URLs should reflect the purpose of the screen. For example, a report page should have an URL like /report, not /testing-page.

Our consideration for smartphones and small mobile devices:

1. **Layout and Formatting:**
   - Each screen only serves a single purpose.
   - Support both portrait and horizontal screen layout.
   - UI controls on the screen can be manipulated by touch actions easily.
   - Provide intuitive navigation controls in the app by placing a navigational bar at the bottom of the screen.
   - Support two-dimension scrolling, vertical scrolling for the content, while horizontal scrolling for quick navigational actions.
   - Add small pop-up screens appropriately when extra small information is needed to display.

2. **Data Entry and User Actions:**
   - Support different types of inputs depending on the hardware capability of the mobile device: virtual keyboard, voice input, touches, etc.

- Create the UI controls at a reasonable size to avoid the fat-finger problem where users cannot perform actions due to the inability to select the UI components.
- Implement a mechanism to make reverting actions easy in the case of accidental touches.

3. **Navigation and Visibility:**
   - Add a dashboard/home screen to show the general layout of the apps and link to all critical functions. Allow returning to this screen from any actions.
   - Implement a list-like or grid-like layout to display the tasks, so that they can be located easily.
   - Implement a navigational bar that can be collapsible but can be triggered to open easily.
   - Ensure the Back action will always work regardless of the location of the button (in-app or OS-level)

4. Data Management

4.1. Storage Technologies

- Video feeds coming from street cameras are binary objects, so it is ideal to store them in a high-performance object storage system. A database can store such data in serialized string format but it is inefficient and has a size limit.
- Given that the data coming from various sensors can differ from each other in terms of shape, NoSQL databases should be used to store street sensor data.
- The remaining system can use either a NoSQL database system or a relational database system, depending on the use case. Since our system prioritizes high availability over consistency, NoSQL database solutions should be preferred. Some structured data, such as user information, can be chosen to store in the relational database.

4.2. Approaches for a secure and scalable data management

Automation Pipelines:

Footage coming from street cameras can be large in terms of storage size. Meanwhile, traffic officers would tend to display video footage of multiple streets on the same screen. Therefore, it would be better to compress the file and reduce it before sending it to the end-users. To do this, the video footage can b passed through several data pipelines to reduce the dimensions and color bitrates to an acceptable level that cannot be differentiated by human eyes.

Encryption and Access Control:
Adopt strong encryption mechanisms to secure traffic data and user information. Limit database access with to authorized personnel with the role and permission system such as Access Control Level (ACL).

Optimized Read/Write Operations:
Since TMS will receive the traffic data constantly, the write operations will cause a burden on the system. The same is also applicable for read operations since both the operators and passengers will access the system for particular businesses. Therefore, our database and storage design will adopt the CQRS and Materialized View patterns to avoid complex queries and provide cached data for end-users. To that end, we can split the database into two main structures:

- One part for read operations: SQL Database
- One part for write operations: NoSQL Database

Scalability Techniques:
The TMS should allow horizontal scaling of databases to handle growing data volumes. We can perform sharding to store smaller parts of the database on different database instances. Meanwhile, we will replicate the database to ensure that all instances will reach eventual consistency via the primary-secondary model.

4.3. Data-Driven Design approach for the system

Data-Driven Design (DDD) places an emphasis on the collection and analysis of empirical data to make design decisions. In TMS, we can use DDD to improve the traffic prediction system and create personalized experiences for the end-users. The process for this approach is below:

Data Collection:
The data can be collected from different sources such as:
- IoT Devices, traffic sensors and cameras about the current traffic conditions
- GPS data about the latest position of the public transport vehicles
- User preferences, historical travel patterns, and feedback

Data Analysis:
From a series of data samples, conduct a thorough analysis to provide an initial report about the usage pattern and trends. Use these results to create the necessary solutions for giving decisions or suggestions to end-users. For example: From collected user data such as travel patterns, preferred modes of transportation, and frequently visited locations, TMS can give suggestions for passengers about optimal routes, predict transit arrival times, and offering alternative transportation options based on user preferences.

Implementation:
Apply data pipelines with machine learning algorithms to analyze patterns, identify trends, and predict future traffic conditions. This may allow TMS to automatically optimize traffic signal control, reroute public transport, and provide proactive information to users about potential congestion or delays. Implement the necessary heuristics based on the given user data to create suggestions for passengers on the server side.

Testing and Iteration:
Gather more data to ensure the correctness of the data analysis pipelines over each iteration.

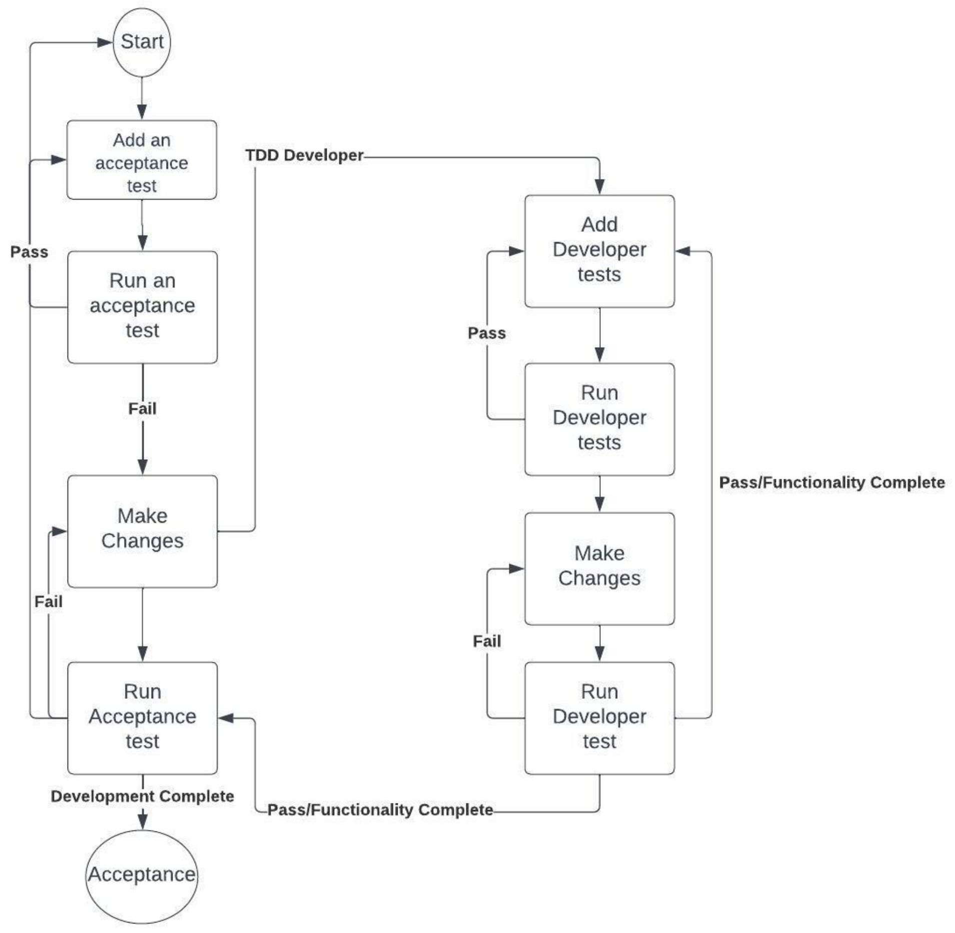
5. Development Practice

5.1. Test-Driven Design:

Before we start coding, we should create some acceptance test cases based on the functional requirements, and our development should follow the test cases. Once the test cases are done before code implementation all the test cases should fail. After implementation the test cases should be successfully passed to meet the user acceptance criteria. If they do not pass, then TDD developer should make adjustment on the code. The process will continue until we meet the requirements.

For our project here is a list of test cases based on functional requirements before code implementation. To follow the TDD design all the test cases listed should fail.

A typical work-flow of the TDD:



Fail test cases before code implementation:

Traffic Monitoring and Control:

Test Case: Real-time Traffic Data Storage Failure

Scenario: When real-time traffic data is received from IoT devices, it should be stored in the system for analysis.

Test Steps:

Simulate the arrival of real-time traffic data.

Attempt to retrieve and verify the stored data.

Expected Result: The test should fail since the storage mechanism is not yet implemented.

Public Transport Management:

Test Case: Route Optimization Failure

Scenario: The system should optimize public transport routes based on real-time traffic conditions.

Test Steps:

Provide a set of routes and current traffic conditions.

Request the system to optimize the routes.

Validate that the optimized routes match expectations.

Expected Result: The test should fail as the route optimization algorithm is not yet implemented.

Rider Experience:

Test Case: Mobile Ticketing Validation Failure

Scenario: Users should be able to purchase mobile tickets and have them validated during a journey.

Test Steps:

Simulate the purchase of a mobile ticket.

Attempt to validate the mobile ticket during a journey.

Verify that the validation process succeeds.

Expected Result: The test should fail since the mobile ticketing validation process is not implemented.

Now we have assigned a TDD developer to implement code based on the fail cases. Once the implementation is done below should be the pass test cases.

<u>Successful test cases after code implementation:</u>

Traffic Monitoring and Control:

Test Case: Real-time Traffic Data Storage Success

Scenario: After implementing the storage mechanism for real-time traffic data, ensure that data can be successfully stored and retrieved.

Test Steps:

Simulate the arrival of real-time traffic data.

Store the data in the system.

Retrieve and verify the stored data.

Expected Result: The test should pass, indicating that the real-time traffic data is successfully stored and retrievable.

Other test cases:

- Vehicle Accident reported update the traffic data
- Road Closure happens update the data
- Traffic Signal Change update the data
- Speed Limit Change happens update traffic data

Public Transport Management:

Test Case: Route Optimization Success

Scenario: After implementing the route optimization algorithm, verify that the system optimizes public transport routes based on real-time traffic conditions.

Test Steps:

Provide a set of routes and current traffic conditions.

Request the system to optimize the routes.

Validate that the optimized routes align with expectations.

Expected Result: The test should pass, confirming that the system optimizes routes according to real-time traffic conditions.

Other test cases:

- Public Transport Delay happens recalculate the schedule
- Route Change Request happens recalculate the route and provide alternate route
- Public Transport Schedule Update happens provide updated schedules
- Fare Adjustment happens to adjust fare while purchasing tickets and send notifications for the fare adjustment announcement.

Rider Experience:

Test Case: Mobile Ticketing Validation Success

Scenario: After implementing the mobile ticketing validation process, ensure that purchased mobile tickets can be successfully validated during a journey.

Test Steps:

Simulate the purchase of a mobile ticket.

Validate the mobile ticket during a journey.

Verify that the validation process succeeds.

Expected Result: The test should pass, indicating that the mobile ticketing validation process is successfully implemented.

Other test cases:

- Plan a journey for a user based on preferences and real-time data
- Retrieve journey data for the user
- User Feedback Submitted, update the database
- Mobile Ticket Scanned then change the ticket status to used.

5.2. Agile Practices to Accompany Component Breakdown:

- **Backlog Refinement:** Regularly revisit and refine the backlog to add, remove, or reprioritize components based on feedback and changing requirements.
- **Sprint Planning:**
  - Plan sprints based on the priority of components and their dependencies.
  - Select a set of components that can be completed within a sprint.
- **Continuous Integration:** Integrate and test components continuously to identify and address issues early in the development process.
- **User Feedback Loops:** Gather user feedback on implemented components to make iterative improvements.
- **Refactoring Sessions:** Allocate time for refactoring sessions to enhance the design and maintainability of components.
- **Collaborative Design Sessions:** Conduct design sessions involving cross-functional teams to discuss and adapt components as needed.
- **Continuous Monitoring:** Implement monitoring for critical components to identify performance issues and improve reliability.

6. Scalability and Performance

6.1. Insights

It is important that TMS should adopt the best practices in terms of scalability and performance, since the system requires to be resilient to serve high traffic loads. Below are some social media platform insights that can be applied to TMS to handle extensive real-time data and user interactions.

Microservice Architecture:

- Breaking down the system into smaller, independently deployable services. This allows for individual scalability of components, making it easier to handle varying loads on different functionalities.

Load Balancing:

- Load balancing mechanisms should be implemented to distribute incoming traffic across multiple servers or different services evenly.
- The load balancers should support scaling dynamically to adapt to varying workloads in different situations.

Database Replication and Sharding:

- System databases should be replicated across multiple regions to ensure redundancy, which leads to data availability and better disaster recovery.
- For a high-traffic system like the Smart City TMS, it is challenging to store all data inside a single database server. Therefore, it is important to consider sharding the database to partition data and distribute it across multiple database instances for improved performance.
- Techniques such as CQRS and Materialized View will be adopted in the system to improve the read/write operations.

API Gateway:

- To decouple the front-end from the back-end, we design the TMS to have an API Gateway. The API Gateway service will receive the requests coming from the client-side and redirect it to the appropriate resources.

Considering Cloud-Based Infrastructure:

- If using on-premise infrastructure is too costly in terms of building and maintaining, the Smart City TMS can consider using fully or partially some services on the cloud. Major vendors such as AWS, Azure or Google Cloud should be considered because they have a high service level agreement (SLA) that satisfies the requirements of a high, real-time traffic system of TMS.
- By using cloud services, it is easy to manage storage, computing resources and database. Scaling the system is made easy because the cloud vendors usually have set up rules for scaling based on a specific set of criteria.

Adopt Content Delivery Network (CDN):

Implement CDN to serve static assets such as images, HTML, CSS, JS files to end-users in a minimal amount of time.

6.2. Domain-Driven Design

Given the nature of TMS, the system contains various complexities that should be treated carefully. In this case, we should apply the domain-driven design (DDD) methodology to identify and understand the core domain concepts, define boundaries, and structure the system to reflect the real-world complexities of urban transportation accordingly. The steps to apply DDD is listed below:

Domain Exploration

- Finalize Domain Concepts: Work with transportation planners, stakeholders and transportation domain experts to identify relevant concepts and terminologies to urban transportation. They could be entities such as vehicles, routes, passengers, and user itineraries.
- Define Ubiquitous Language: Set up a common framework to include shared language to be used consistently among team members. This ensures a consistent understanding of the transportation domain.

Bounded Context

- Identify and define bounded contexts to encapsulate aspects of transportation domain. For instance: Traffic management, public transport, and rider experience can be three separate bounded contexts.

Aggregate Design

- Aggregates are clusters of entities and value objects treated as a single unit. Find aggregate roots within each bounded context in TMS. For instance, a Journey aggregate might include entities like Passenger, Route and Ticket.

Repositories

- Create repositories to store and retrieve aggregate roots within the transportation domain. For example, create a JourneyRepository for accessing and persisting journey-related aggregates.

Services

- Build a domain service to address complex operation that involves coordinating multiple entities within the transportation domain. For instance, using a RoutingService may handle the optimization of routes based on real-time traffic data.

Factories

- There are complex objects or aggregates that require special logic for initialization in TMS. In that case we can use Factories to make the creation process easier. For instance, we can use the Factory design pattern to create JourneyFactory, which could encapsulate the logic of creating a Journey aggregate, considering factors like user preferences, real-time traffic conditions, and available transportation options.

Domain Events

- In TMS, events such as traffic incidents, route changes, service disruptions, or changes in public transport schedules could happen. We can use mechanisms such as an event bus or a dedicated event publishing service to publish domain events to the corresponding subscribers.

7. Deployment Strategy

To ensure that the system has a high availability and disaster recovery preparedness, we can implement the following strategies:

Multi-Region deployment:

- We should deploy the Smart City TMS across multiple geographic regions to distribute traffic load and enhance fault tolerance.
- The data center locations should be chosen strategically to minimize latency and optimize performance for city-dwellers.
- In the context of the city of Technopolis in the given project scope, we should choose at least two locations that are inside or close to Technopolis to be the data center. For example, if Technopolis resides in North America, one major data center can be on the eastern side, while the other one will be on the western side of the continent.

Containerization and Orchestration:

- To deploy the program, we should consider utilizing a containerization platform like Docker to package the application and its dependencies. It reduces the complication of having to set up everything on every server.
- For autoscaling, developers should evaluate container orchestration solutions such as Kubernetes to automate deployment, scaling, and management of containerized applications.

Continuous Deployment and Integration (CI/CD):

- In the deployment process, CI/CD pipelines can be integrated to perform automation tests, building and deploying new features or updates.
- The CI/CD pipeline could prevent unqualified code by applying automated linters to check the code against the existing conventions.

Gradual Deployment:

- Deployments should mostly be made when the level of service usage is low, for example, mid-nights. Gradually update instances or microservices may help keeping system operational during the update process.
- Rolling deployments can minimize service interruptions and maintain continuous service availability.

Monitoring and Alerting:

- The system should implement monitoring tools to track system performance, resource utilization, and user interactions.
- Alert mechanisms should be in place to notify the operation group of anomalies or issues in real-time or near real-time.

System Backup:

- Regularly backup databases and replicate them in different zones and regions.
- For each backup, at least two copies should be made on different physical media to ensure redundancy.

Disaster Recovery Plan:

- A comprehensive disaster recovery plan should be developed to define the goals, personnel involved, the backup process, and procedures for quick system restoration in case of failures. For TMS:
    - The recovery time objective (or downtime) should be at most 3 hours in peak time or 12 hours in off-peak time.
    - The recovery point objective (or data loss) should be only within the same day.
    - The Operation/Dev-Ops teams should be on-call to ensure the availability of services.
    - Backups should be made everyday, at 2 AM when the traffic is at the lowest.
    - If a catastrophic failure occurs, the on-call team must be involved in the database recovery process. The recovery process starts by using the latest version of the full database backup among all regions.
- Conduct disaster recovery practices to ensure the effectiveness of the recovery plan.

Security Measures:

- Different security measures should be implemented at each layer to combat against external threats. For example, the firewall should be implemented in the network layer. For users with access to critical assess, it is imperative that multi-factor authentication should be used.
- Perform penetration testing and security auditing to identify and mitigate vulnerabilities.

8. Agile Project Management Timeline

To provide a credible Agile project management timeline, several factors need to be accounted for:

- The number of team members and their composition. More people may not guarantee a faster development velocity, but it is necessary to identify and hire people with the desired skill sets to complete the project.
- The budget for the development. This is important because it affects the quantity and quality of team members hired.

- The general roadmap that the stakeholders would like to demonstrate the in-progress product. This affects how the development should be planned.
- Whether the team can be split into smaller sub-teams to work on the multiple features simultaneously

The timeline proposed by our team is made with the assumed team composition as below:

Position	**Headcount**	**Main Responsibilities**
Project Manager	1-2	Overall project planning, coordination, and execution. Stakeholder communication and management. Risk assessment and mitigation. Budgeting and resource allocation.
Solution Architect	2	Provide the overall system architecture. Develop high-level design and integration strategies. Oversee technical aspects of the project.
Business Analyst	2	Gather and document functional and non-functional requirements. Conduct user interviews and workshops. Collaborate with stakeholders to refine project goals.
UI/UX Designer	2	Create wireframes and prototypes. Improve the design based on internal and external feedback. Work closely with front-end developers.
Software Developers (backend & frontend)	10-12	Develop the database; implement backend and frontend functionalities for all three subsystems. The developers will work on all subsystems to avoid integration issues happening late in the project.
Data Engineer	1-2	Create and execute database design. Handle data mediation coming from different sources. Create data analysis pipelines to provide business insight for the users.
QA Engineer	5	Develop and execute test plans in every sprint. Perform regression, integration and acceptance testing. Work in concert with developers to build the plan to carry out load testing, and security testing. Identify and report bugs for resolution.
Dev-Ops	2	Build and maintain the system infrastructure. Implement CI/CD process. Ensure the scalability, high availability, reliability and security of the system.

The projects will be developed in iterations, each lasting 3 weeks. Programming is done concurrently with the design to deliver new values at the end of the iteration. The development effort in each sprint will be distributed as below:

- 50% - 60% for feature development. This includes incorporating customers' feedback into the sprint.
- 30% - 40% for bug fixing to address the highest blocking issues for the end-users.
- 10% for system refactoring to avoid code degradation that would require more effort later to fix.

With the given details, the timeline plan is divided into different phases as below:

Phase 1: Project Kickoff and Planning (~1-2 iterations)

Goals:

- Define initial project scope, short-term and long-term objectives, and initial backlog.
- Identify stakeholders and establish communication channels.
- Set up core project infrastructure and tools.

Deliverables at the end of phase:

- Project vision document and scope document.
- Stakeholder list and communication plan.
- Initial project timeline.
- Sprint backlog for the first three sprints.
- Brief report on the responsibilities of team members in the development of the project.

Note: Some activities can be conducted before the project commences, depending on the availability of resources.

Phase 2: Requirements Analysis, UML Diagram Development, and Core Architecture (1-2 iterations)

Goals:

- For each subsystem and the system integration, conduct the system design and analysis to define the use cases.
- Develop initial UML diagrams outlining system structure.
- Define the technology solution and establish the core architecture and integration points.

Deliverables at the end of phase:

- Initial requirements document specification.
- Use case diagrams.
- UML class and sequence diagrams.
- Technology report: the solutions chosen for developing the system.

Phase 3: Core Functionality Development (2 iterations)

Goals:

- Design the database system and entities
- Develop core functionalities for Traffic Monitoring and Control, Public Transport Management, and Rider Experience.

Deliverables:

- Database design document.
- Core features for each subsystem.
- A pre-production environment of the system, ready for verification in subsequent iterations.

Phase 4: Feature Development for Subsystems and Enhanced UML Diagrams (3-5 iterations)

Goals:

- Simultaneously develop specific features for each subsystem.
- Enhance UML diagrams to reflect evolving system structure.

- Prioritize and implement user stories for each subsystem.

Deliverables:

- Subsystem-specific features.
- Updated UML diagrams.
- User feedback on new features.

Phase 5: Enhanced Integration, Cross-Subsystem Features, and Usability Testing (3 iterations)

Goals:

- Strengthen integration between subsystems.
- Implement cross-subsystem features and functionalities.
- Conduct usability testing with end-users.

Deliverables:

- Cross-subsystem features.
- Usability testing results and feedback.

Phase 6: Coding Refinement, Implement Non-Functional Requirements (3 iterations)

Goals:

- Refine user interfaces based on usability testing results.
- Dive into detailed coding for enhanced functionalities.
- Implement security measures based on security requirements.
- Implement secure and scalable database design for managing real-time traffic data and user information.

Deliverables:

- Enhanced user interfaces.
- Detailed code implementations.
- Initial scalability and security measures.

Phase 7: Finalizing feature implementation. Intensive Testing (2 iterations)

Goals:

- Conduct comprehensive testing, including load testing, integration testing.
- Perform security audit
- Finalize the functional and non-functional requirements implementation

Deliverables:

- Testing results and bug fixes.

Phase 8: Final Integration, Deployment Preparation, and Documentation (1 iteration)

Goals:

- Conduct final checks for the integration of all subsystems.
- Prepare for deployment, including documentation and training materials.

- Address final feedback and bug fixes.

<u>Deliverables:</u>

- Fully integrated Smart City TMS.
- Comprehensive testing results.
- Deployment-ready system with documentation.

Phase 9: Deployment, User Training, and Feedback Analysis (2 sprints)

<u>Goals:</u>

- Deploy the Smart City TMS in the production environment.
- Conduct user training sessions.
- Support customer requests for the new system

<u>Deliverables:</u>

- Deployed Smart City TMS.
- User training materials and sessions.
- Analysis of initial deployment feedback.

Phase 10: System Maintenance mode, and Documentation (indefinitely)

<u>Goals:</u>

- Continuously monitor system performance and gather user feedback.
- Implement optimizations and address any issues.
- Finalize all documentation and lessons learned.

<u>Deliverables:</u>

- Optimized Smart City TMS.
- Improvement plan