Integrated Public Transportation System Project Report

Dharini Baskaran * Surya Rajendran†

December 13, 2023

1 Introduction and Motivation

In the evolving landscape of urban mobility, the need for seamless and efficient public transportation systems has become paramount. This project proposes the development and implementation of an Integrated Public Transport System—a pioneering initiative designed to unify various modes of public transportation into a cohesive and user-friendly framework. This system aims to streamline the travel experience for commuters by allowing them to travel around a lot more using multiple transportation modes with a 90 minute fare.

In a city like Boulder, public transportation offer users a range of options, including Day Passes, RTD MyRide cards, 3-hour Cash Passes for bus transport, participation in the Bcycle program, utilization of
Nightrides, engagement in RideShares, and more. However, the existing challenge lies in the fragmented nature of these services, where users are compelled to manage separate passes or applications to monitor their usage and make payments across various transportation modes. Our proposed project aims to introduce a unified payment solution—a singular card or payment mode—that seamlessly transcends all transport zones. This comprehensive system is designed not only to simplify the user experience but also to incentivise regular use of public transportation. By consolidating disparate modes of transport into a singular, user-friendly platform, our initiative seeks to encourage widespread adoption of public transit, offering an integrated and efficient solution that promotes the ease and attractiveness of utilizing the region's diverse transportation services.

The Integrated Public Transport System addresses this challenge by leveraging data scaling techniques to seamlessly integrate disparate modes of transportation. By employing advanced techniques, the system will facilitate a unified payment process, allowing users to pay for their entire journey with a single transaction. This innovative approach not only enhances user convenience but also has the potential to significantly increase public transport ridership.

Moreover, the project aligns with broader sustainability goals by contributing to the reduction of carbon emissions in urban environments. Encouraging the use of public transportation over individual vehicles is a key strategy in mitigating the environmental impact of urban commuting. Through the seamless integration of payment systems, the project aims to create a more attractive and user-centric public transportation experience, ultimately promoting eco-friendly modes of travel.

*dharini.baskaran@colorado.edu
†surya.rajendran@colorado.edu

2 Architectural Diagram

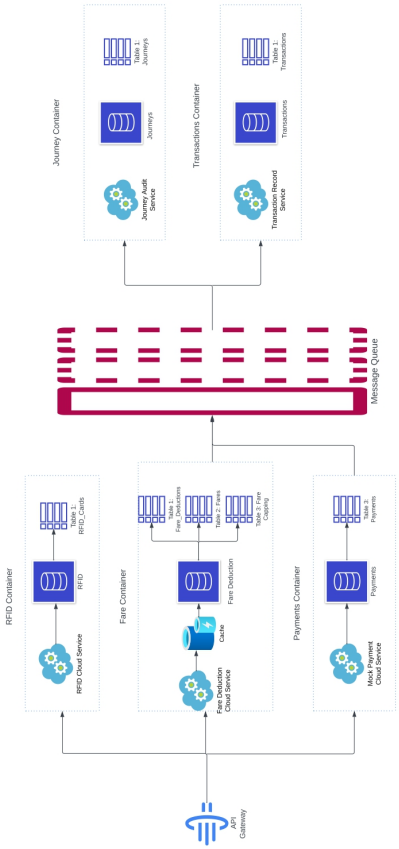

Figure 1: Architecture Diagram

3 Software Components

1. **SQL:**
   - We have used the relational tables and database predominantly for our project. We used the postgres database. It is used for RFID tag

management, Fare deduction services, for calculating and updating payments and top-up services.

- The main advantage is, its fully ACID compliant, thus ensuring data integrity and reliability. Disadvantage is lack of options in GUI.

2. **Key-Value Store:**

   - We have incorporated NoSQL database for our Transactions database. We have used DynamoDB in our system. DynamoDB is chosen to implement key-value store. This transaction table contains details of the current transaction such as the transactionId, the cardId, the amount calculated, the transactionType and timestamp details like created and modified time.
   - The main advantage is that MongoDB is designed to scale horizontally, making it suitable for handling large amounts of data and high traffic.

3. **Redis MQ:**

   - Our project incorporates Redis MQ to work asynchronously based on our EDD (Event-Drive
     n-Design). This is done mainly to facilitate real-time updates across our system. This Redis MQ calculates the fare to be calculated based on trasactionType, zone and more. And it also used to update the fare deduction details in the Fare-deduction table.
   - The main advantage is that Redis is an in-memory data store, which allows for fast message processing, which is essential for our usecase. The drawback is that Redis does not guarantee the ordering of messages, which might be essential in certain scenarios. But it doesn't affect our usecase.

4. **Redis Cache:**

   - Redis Cache is utilized to enhance our system performance by storing data in-memory. We share frequently used data like busId, fare rate for different zones and more.
   - The most advantageous reason is that Redis allows for partitioning and sharding, making it easy to scale horizontally. The disadvantage is that Redis is an in-memory store, leading the capacity to be limited by available RAM.

5. **Containers:**

   - We will be using Docker containers for easy scalability of our public system. Containers will play an important role in achieving a modular architecture for our project. This ensures that the system will provide reliability. Kubernetes will be used to orchestrate these containers.

- The advantage is the containerization, which enables our services and their dependencies to be packaged together in a consistent and isolated environment. This is really handy in our system. The disadvantage is that docker will introduce some resource overhead.

4 Software Interaction

Our system is strategically crafted following an API-first design approach. This intentional methodology has been adopted to streamline and define the project scope effectively. This approach aligns with best practices, thus facilitating a modular and scalable system architecture.

In the realm of backend software development, seamless interaction between various services is pivotal for a robust and responsive system. This section elucidates the software interactions within our integrated public transport system's service backend system, focusing on two primary REST endpoints: the Tag endpoint and the Top-up endpoint, both of which are crucial for the operation of a public transport card usage system.

Furthermore, we've established a schema-based database for Postgres, structuring our data storage systematically. The modularization of our project into distinct services, each encapsulating specific functionalities, enhances the organization and maintainability of our system. Leveraging Docker and Kubernetes, we've optimized the deployment process, ensuring efficiency and scalability. We have used Kubernetes to orchestrate all our containerized services. Additionally, the implementation of a messaging system with Redis, persistently listening in real-time, allows us to capture logs seamlessly, minimizing latency and enhancing the overall responsiveness of our logging system. This logging system is designed to work asynchronously.

As can be seen in our architecture diagram, all the communications between the containerized services takes place via the Gateway. This is done to ensure centralized control. This gateway can distribute incoming requests across multiple backend services to achieve load balancing.

The initiation of our system's workflow occurs when a user taps their card to facilitate payment for their ongoing journey. Presently, our system is configured to capture journeys in buses and trains, with the database designed to accommodate the addition of new modes seamlessly. Upon card tapping, the RFID tag is read, and the card details are recorded. The details include cardId, balance amount and the timestamp. These details are first searched in our Redis cache, only if it turns up not present, a query to RFID-Cards table is made to retrieve the details. This is done to ensure minimum latency. The payment process for a bus journey is straightforward, with users making a single payment upon entry. However, for train journeys, a tag-on and tag-off system is implemented. This system not only tracks the user's entry and exit stations but also records the journey time.

Our system promotes public transportation by offering features like fare-capping and a 90-minute window, regardless of the mode of transport. For

now, the fare is capped to 10 USD for a 7 day time period, wherein they can ride the next journeys free following the 10 USD spent in a 7 day time period. The user doesn't have to keep track of this. Our database contains all these information and simply on tap, doesn't deduct any fare amount from the passenger's card. This fare-capping mechanism encourages commuters to utilize public transportation efficiently.

Upon tapping the card, our system promptly verifies its validity. If the card is found to be invalid, an error is immediately triggered, akin to an invalid card scenario in our Boulder public transport system. Conversely, for a valid card, the system proceeds to calculate the fare to be deducted. Subsequently, the system checks the card's balance, and if the funds are insufficient, the card is declined. In cases where the card holds an adequate amount, the system deducts the fare and meticulously records this transactional data. The audit data, encompassing journey details, is then persistently stored in our Journey database housed in Dynamo DB, facilitated by the Redis Message Queue (MQ).

In instances of insufficient balance, our system offers users the flexibility to top up their cards independently. This is achieved through our top-up POST method, allowing users to load any desired amount via their preferred payment method. Despite the absence of a frontend, we have mockingly simulated this aspect of our system. Each card reload transaction is recorded in our Transaction table within Dynamo DB. The fare deduction table is contemporaneously updated each time a card is tapped to pay for tickets. To retrieve details from our Dynamo DB, we have implemented a journeyFunction cloud functionality. This is done, because a Google Cloud Function is a serverless compute service and allows us to run single-purpose functions without much/any management overhead.

The Audit service listens for incoming messages and is seamlessly connected to both the Journey and Transaction tables in Dynamo DB. Consequently, whenever a card is recharged or tapped, the Redis MQ efficiently relays messages to update transaction and journey details in Dynamo DB.

Furthermore, our system incorporates a feature to notify users of the success or failure of their card recharge via email services. This notification mechanism serves to keep users informed about their payment transactions, enhancing their ability to track and manage their cards effectively. Redis, once again, plays a pivotal role by serving as the message queue from which the Email Service retrieves the transaction details to notify the user.

In the case of train journeys, the system incorporates a dynamic tagging mechanism when a user taps their card at a train station. This tagging process involves checking the previous entry to toggle the tag value. When the user initiates a train journey, the tag is switched on, indicating the start of the journey. Conversely, when the user concludes the journey by tapping the card again, the tag is toggled off, signifying the journey's end.

Subsequently, if the user taps the card within a 90-minute window from the commencement of the journey, the system introduces a fare exemption for that particular leg of the journey and the timestamp is updated with the current time to keep track of the time left in the 90-minute window. During this specified

timeframe, no fare is deducted, promoting a user-friendly approach by allowing flexibility in travel plans without incurring additional charges for closely timed consecutive taps within the given time limit. This process is taken care by the Tag REST endpoint.

5 Debugging and Testing

In our project, we have implemented a logging system that integrates seamlessly with our earlier labs' logging service. The logging code is designed as an infinite loop, continuously monitoring a specific key in a Redis database for incoming log messages. This loop decodes and prints these messages to the console, ensuring a continuous flow of abstracted information. The use of Redis in this context serves as a pivotal component, establishing a distributed log processing system. This architecture allows logs originating from diverse sources to be centralized, facilitating efficient monitoring and in-depth analysis of system activities. Additionally, the implementation incorporates robust exception handling to ensure the continuous operation of the logging system.

We have tested our system in postman. We have done a load testing to check the latency. First once the containers were all deployed in Kubernetes, for 100 users we could see a very large latency. So we checked and changed our postgres database connection to pool-based one. Then with this change, we got next to no latency.

6 Future Scope

Our current project focuses exclusively on the backend and middleware components of a system, employing an API-First design approach. As a result, no frontend has been developed, necessitating the use of mock data for the demonstration of payment deduction processes. The forthcoming critical step involves the creation of a frontend capable of handling various modes of payment through different applications.

At present, our system diligently records unique journey details for each user. This encompasses information such as journey times, preferred modes of travel, and involves intricate fare calculations with additional fare capping considerations. As our data repository grows to a substantial volume, the logical progression of the project involves transforming it into a data mining initiative. This transition will enable us to extract meaningful patterns for each user, facilitating the provision of personalized suggestions on optimal travel times and preferred transportation modes. The overarching objective of this endeavor is to encourage frequent utilization of public transport among users.

The other extension will be updating our system to consider types of users and deduct the fares accordingly. The types may be like senior citizens, first responders, students and so on. This makes our system more robust and easy to adapt to any organization's use.

7 Inspirations

1. TFI - Transport for Ireland
2. TFL - Transport for London
3. RATP France
4. MARTA - Metropolitan Atlanta Rapid Transit Authority, Atlanta.
5. RTD - Regional Transportation District, Denver.
6. MTA - Metropolitan Transportation Authority, New York City.
7. MBTA - Massachusetts Bay Transport Authority, Boston.

8 Appendix

Attaching Database Schema and API details.

APPENDIX 1 : Database Schema

Service: RFID Card Management Service

```
CREATE TABLE rfid.Card_Detail(
   card_id VARCHAR(32) PRIMARY KEY NOT NULL,
   user_id INT NULL REFERENCES dev.Users(user_id),
   balance DECIMAL(10,2) NOT NULL,
   isActive BOOLEAN NOT NULL DEFAULT TRUE,
   created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Service: Fare Deduction Service

```
CREATE TABLE fare.Fare_Deduction(
   deduction_id SERIAL PRIMARY KEY ,
   card_id VARCHAR(36) NOT NULL REFERENCES rfid.Card_Detail(card_id),
   zone_no INT NOT NULL,
   mode_id INT NOT NULL,
   amount DECIMAL(10,2) NOT NULL,
   tagged_on_timestamp TIMESTAMP NOT NULL,
   expiration_time TIMESTAMP NOT NULL,
   created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE fare.Fare_detail(
   fare_id INT PRIMARY KEY NOT NULL,
   mode_of_transport VARCHAR(50) UNIQUE NOT NULL,
   amount DECIMAL(10,2) NOT NULL
);


CREATE TABLE fare.fare_mode(mode_id INT PRIMARY KEY, fare_type boolean DEFAULT FALSE);

CREATE TABLE fare.zone_detail(
   zone_id INT PRIMARY KEY,
   zone_no INT NOT NULL UNIQUE,
```

```
   fare_id INT NOT NULL REFERENCES fare.Fare_detail(fare_id),
   mode_id INT NOT NULL REFERENCES fare.fare_mode(mode_id),
);

CREATE TABLE fare.Fare_Capping(
   fare_capping_id INT PRIMARY KEY NOT NULL,
   fare_id INT NOT NULL REFERENCES fare.Fare_detail(fare_id),
   time_period INT NOT NULL,
   max_amount  decimal(10,2) NOT NULL
);
```

Service: Payment Service

```
CREATE TABLE pmt.Payment(
   payment_id INT PRIMARY KEY NOT NULL,
   card_id VARCHAR(16) FOREIGN KEY (RFID_Cards) NOT NULL,
   amount DECIMAL(10,2) NOT NULL,
   payment_method VARCHAR(50) NOT NULL,
   payment_status VARCHAR(20) NOT NULL,
   created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE public.Users(
   user_id INT PRIMARY KEY NOT NULL ,
   first_name VARCHAR(50) NOT NULL,
   last_name VARCHAR(50) NOT NULL,
   email VARCHAR(100) NOT NULL
);

INSERT into public.users ( user_id,first_name,last_name,email) values
(1,Surya,Rajendran,test1@gmail.com);
```

Appendix 2: API Gateway

RFID Card Management API

Overview

The RFID Card Management API provides functionality for handling RFID card interactions, specifically for recording user entry and exit onto modes of transport. This API is designed to be integrated with an RFID card reader system.

Base URL

```
https://api.example.com/rfid
```

Endpoints

1. Tag Endpoint

Description

Records the entry or exit of a user onto a mode of transport when they tap their RFID card. If the user has not started a journey, it tags ON. If the user is already in a journey, it tags OFF. If the reader is in a different mode of transport, it also tags ON.

Endpoint URL

```
POST /api/tag
```

Request

- **Card ID**: Unique identifier of the RFID card (string or integer).
- **Journey Status**: Boolean indicating if the user is already in a journey (True) or not (False).
- **Reader Mode**: Mode of transport of the reader (string).

```
{
"card_id": "100020000190",
"journey_status": false,
"reader_mode": "bus"
}
```

Response

- **Success**: Boolean indicating if the tagging process was successful.

```
{
"success": true
}
```

2. Top-up Endpoint

Description

Allows users to add value to their RFID card, increasing the card's balance for fare payments.

Endpoint URL

```
POST /api/topup
```

Request

- **Card ID**: Unique identifier of the RFID card (string or integer).
- **Amount**: Amount to top up (float or integer).

```
{
"card_id": "100020000190",
"amount": 20.00
}
```

Response:

- **New Balance**: New card balance after top-up

```
{
"new_balance": 35.00
}
```