Software Engineering Project 3

Team 19

Aman Raj

Shambhavi Jahagirdar

Pranjali Bishnoi

Soham Korade

Mohammad Zaid

Task 1: Requirements and Subsystems

☐ **Functional Requirements:**

1. **Subscription Plans**
    - FR1.1: Provide subscription plans with benefits including regular rates at peak hours, discounts, and security features.
    - FR1.2: Offer different subscription tiers with varying benefits and pricing.
    - FR1.3: Display subscription plans with details such as pricing, benefits, and duration.
    - FR1.4: Enable users to select and subscribe to a plan through the app.
2. **Reduced Waiting Time**
    - FR2.1: Aggregate data from multiple cab providers to minimise waiting times.
    - FR2.2: Prioritise cab assignment for subscribers during peak hours.
    - FR2.3: Provided estimated time arrival time based on location coordinates.
3. **Safety and Security**
    - FR3.1: Implement identity verification and background checks for drivers based on uploaded documents.
    - FR3.2: Assign drivers based on a certain number of rides/ratings for safety-concerned customers.
4. **Payment**
    - FR4.1**:** Facilitate secure payment transactions for subscription plans, ride bookings, and other services.
    - FR4.2**:** Provide options for users to manage and choose their preferred payment methods.

- FR4.3**:** Process payments for individual ride bookings, including fare calculations and adjustments.

5. **User Interaction**
    - FR5.1 : Provides separate interface for drivers as well as passengers , enabling passengers to book ride and drivers to accept pending(not assigned to driver yet) rides.
    - FR5.2: Display best available cab options with estimated fares and arrival times.
    - FR5.3: Enable profile management for users to update personal information, enabling drivers to add vehicles, and passengers to view ride history.
    - FR5.4: Provide feedback, rating, and reporting mechanisms for users.

☐ **Non-Functional Requirements:**

1. **Performance**
    - NFR1.1: The system should respond to user requests with minimal latency.
    - NFR1.2: The system should handle at least thousands of concurrent users.
2. **Security**
    - NFR2.1: All user private data such as password should be encrypted.
    - NFR2.2: Implement secure authentication and authorisation mechanisms.
3. **Usability**
    - NFR3.1: The user interface should be intuitive and user-friendly.
    - NFR3.2: It supports user initiatives such as canceling actions and undoing actions.
4. **Reliability**
    - NFR4.1: The system should have an high uptime.
    - NFR4.2: Ensure data integrity.
    - NFR4.3: Any system failures should be monitored and reported to the admin.
5. **Scalability**
    - NFR5.1: The system should be scalable to accommodate an increasing number of users and cab providers.
    - NFR5.2: Design the system to handle future expansion, including adding new features or cab types.

☐ **Architectural Significance**

We list down below the architectural significance of the above functional and non-functional requirements :

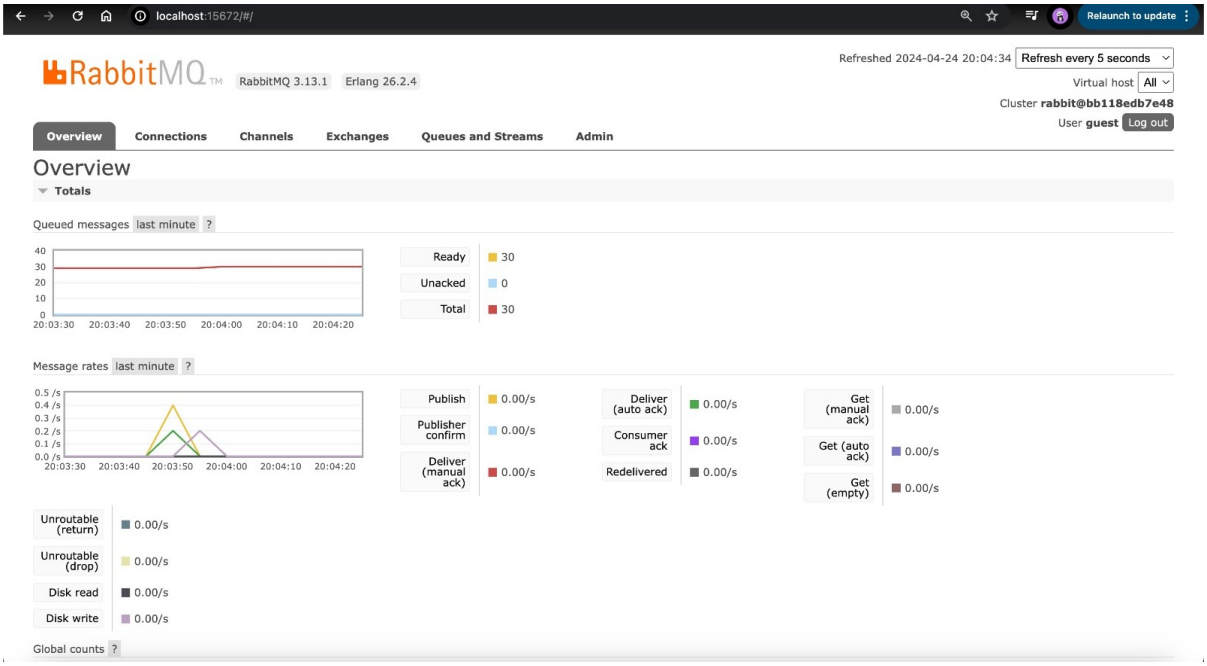
1. Performance : `Redis caching` optimises the response time for frequent read requests . By caching frequently accessed requests data, we reduce database hits and improve the overall system performance.We also ensure that Redis caches are invalidated on data modification.

```python
redis_host = 'localhost'
redis_port = 6379
redis_db = 0
redis_client = redis.StrictRedis(host=redis_host, port=redis_port, db=redis_db)


def fetch_from_redis(key):
    try:
        vehicle_data = redis_client.get(key)
        if vehicle_data:
            return json.loads(vehicle_data)
        return None
    except redis.RedisError as e:
        print(f"Redis Error: {e}")
        return None
```

```python
@app.route('/driver/current_ride/<driver_id>', methods=['GET'])
def get_current_ride_driver(driver_id):
    try :
        ride = fetch_from_redis('driver_current_ride_' + driver_id)
        if ride:
            return ride
        ride = RideDAO.RideDAO.get_current_ride_driver(driver_id)
        if not ride :
            return jsonify({'ride':None,'error':'error occured'})
        cache_in_redis('driver_current_ride_' + driver_id, ride)
```

2. Scalibility : Adopting the `microservices` architecture allows us to independently scale services handling cab aggregation and assignment. This ensures that we can handle varying load demands efficiently, especially during peak hours, without affecting the entire system's performance.This also allows us to choose individual databases for each microservices, depending on the need and the context.Each microservice can use a database technology that's optimised for its specific needs.
3. Reliability : To ensure service reliability, each microservice is `continuously monitored` .The continuous monitoring is done by a script , which calls the notification microservice to notify the admin on high error throw rates. The notification service is based on event driven architecture and uses pub-sub pattern - `rabbitmq` . This proactive monitoring helps us quickly identify and rectify any service disruptions or failures, minimising downtime and ensuring consistent service availability.



4. Usability : `Separating the frontend and backend` enhances modularity and maintainability, providing better usability . With distinct driver and passenger views, we optimize user experience by providing interfaces to specific user roles. This design choice also supports concurrent user sessions, allowing drivers and passengers to interact with the platform simultaneously without conflicts.

   We also use `State Driven Approach`, where we model the ride state as a state , and maintain the `current state of the ride` in our database to provide `usability features like cancellation of rides`. All the checks are made at backend to ensure that those transitions are valid and done by authorised users.

```python
@staticmethod
def passenger_cancel_ride(ride_id):
    ride = session.query(Ride).filter(Ride.ride_id == ride_id).first()
    if not ride:
        return None
    ride_metadata = session.query(RideMetadata).filter(RideMetadata.ride_id == ride_id).first()
    if ride_metadata.ride_status not in [1,2] :
        return None
    ride_metadata.ride_status = int(RideStatus.PASSENGER_CANCELLED)
    session.commit()
```

5. Security : Implementing a `centralized API gateway` handles all incoming requests, taking care of authentication and authorisation. This architecture ensures that the backend micro -services remain **isolated from direct exposure**, reducing potential security risks and enforcing consistent security policies across services.

```python
@app.route('/driver/add_vehicle', methods=['POST'])
def add_vehicle():
    data = request.get_json()
    response = requests.post(BASE_URL_ENTITY + 'driver/add_vehicle', json=data)
    record(response.json())
    return jsonify(response.json())      You, 6 days ago • gateway added
```

☐ **Subsystem Overview - Role and Functionality**

1. User Management Subsystem

Description:

Note that RideMerge includes two types of users : `passengers` and `drivers`.
The User Management Subsystem handles all aspects related to user registration, authentication, vehicle addition(by drivers) and profile management within the RideMerge platform.

Role:

- Manages user accounts and authentication.
- Ensures the security and privacy of user data.
- Facilitates user profile creation and management.

Functionality:

- User registration and login mechanisms.
- Profile creation and editing.
- Vehicle addition by drivers.
- Verification of driver’s details such as driving licence.

2. Subscription Management Subsystem

Description:
Manages subscription plans, benefits, and user subscriptions to enhance user experience and loyalty.

Role:

- Defines and manages various subscription plans.
- Tracks and manages user subscriptions and benefits.

Functionality:

- Subscription plan creation, modification, and deletion.
- Benefits allocation and management for subscribers.

3. Ride Booking Subsystem

Description:
Facilitates the booking, scheduling, and management of rides for users.

Role:

- Provides a seamless booking experience for users.
- Manages cab assignments, scheduling, and ride status updates.

Functionality:

- Real-time availability and fare estimation.
- Automatic scheduling for regular commuters.
- Ride cancellation and modification capabilities.

4. Payment Subsystem

Description :

The Payment Subsystem is responsible for managing all financial transactions within the RideMerge platform, ensuring secure and seamless payment processes for users.

Role:

- Manages and processes payments for subscription plans, ride bookings, and other services offered within the platform.
- Manages payment methods, providing users with options to perform payment with their preferred methods.
- Notify Users and send mails/messages to users upon successful transactions.

Functionality:

- Integrates with secure payment gateways to process credit/debit card payments, digital wallets, and other payment methods.
- Allows users to choose preferred payment methods while making a payment.
- Validates and verifies payment methods for accuracy and security.
- Calculates and processes payments for individual ride bookings, including fare estimation and adjustments
- Sends payment confirmations to users upon successful transactions.

5. Notification and Alert Subsystem

Description:
Handles real-time notifications and alerts to keep users informed about ride statuses, updates, and emergencies.

Role:

- Sends timely notifications and alerts to users as well as admin.
- Ensures users are updated about important information and events such as ride-booking/cancelling etc.

Functionality:

- Real-time notifications for ride assignments, updates, and alerts.
- Emergency alerts and notifications for immediate assistance.
- Subscription and promotional notifications.
- Sending alert to admins if system is behaving unexpected , throwing errors.

Task 2: Architecture Framework

Note :

The Architecture Decision Records(ADR) is already pushed to github.

Stakeholders

The users or stakeholders effected by this problem:

1. **Passengers** :
    - **Office-goers**: Individuals who commute to and from their workplace regularly.
    - **Students**: Students travelling to schools, colleges, or universities.
    - **General Commuters**: Individuals who frequently use cab services for daily commutes or occasional travel within cities.
2. **Drivers**:
    - **Cab Drivers**: Individuals who provide cab services to commuters using platforms like Uber, Ola, and others.
    - **Independent Drivers**: Individuals who operate independently without affiliations to major cab service providers but use digital platforms for bookings
3. **Cab Service Providers**: Major cab service providers like Uber, Ola, and others operating in various cities and regions.

Concerns:

1. **Commuters' Concerns:**
    - Seamless transportation experience through easy booking of cabs.
    - Efficient and reliable service.

- Ease of booking and payment.

2. **Drivers' Concerns:**
   - **Increased Earnings**: Opportunities to maximize earnings through more bookings.
   - **Fair Treatment**: Fair algorithms and policies by cab service providers.
3. **Cab Service Providers' Concerns:**
   - Increased user base.
   - Operational efficiency.
   - Integration with the platform.

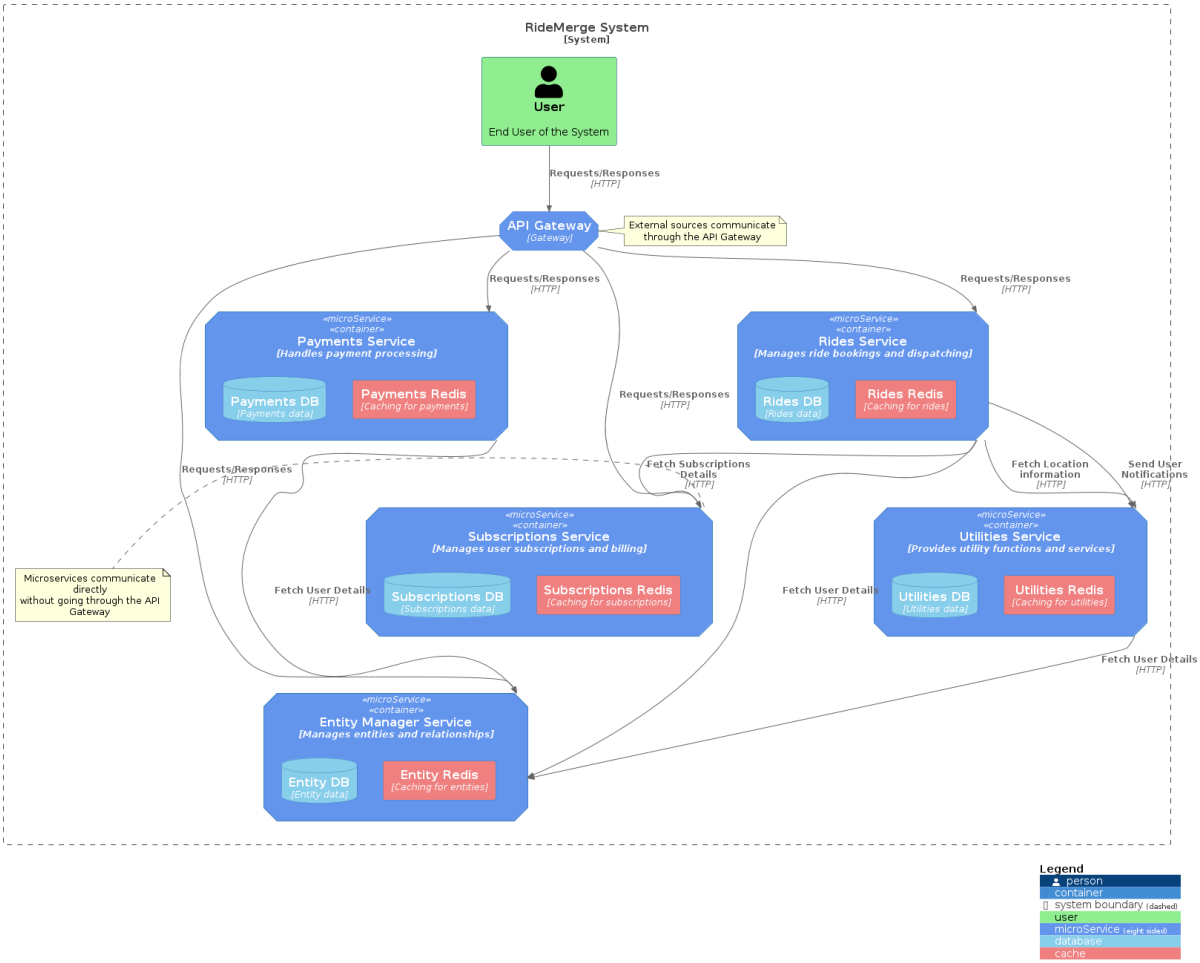
Viewpoints and Views:

1. **Commuters' Viewpoints:**
   - **Efficiency Viewpoint:** Addressing the efficiency of the transportation service. This is achieved by comparing and fetching the best available cabs from all service providers. It reduces the time commuters spend looking for a cab on different platforms and ensures heightened availability and reliability.
   - **User Experience Viewpoint:** Focusing on the ease of use and overall experience for commuters.
2. **Drivers' Viewpoints:**
   - **Earnings Viewpoint:** Pooling users across all other apps into a single one, thus expanding the user base and demand for rides.

Task 3: Architectural Tactics and Patterns

Implementation Details:

Architecture Diagram Model



We chose a microservices architecture to modularise our system into independently deployable services, each handling specific business functionalities. This approach aligns with our identified context, ensuring flexibility, scalability, and maintainability.

- **subscriptions_service**: The **subscriptions_service** takes charge of managing subscription plans, catering to the diverse needs and preferences of our users. It offers various subscription options, each with its own benefits. These benefits offers relaxation on surge fees and access to premium vehicles and also priority in ride bookings, ensuring subscribers enjoy an enhanced ride-sharing experience. Additionally, the service tracks the expiry dates of subscriptions.
- **entitymanager-service**: The **entitymanager-service** manages user, driver, and vehicle entities within our platform. It offers a functionalities that enable both passengers and drivers to register, login and allow drivers to onboard their vehicles by providing necessary information. Beyond authentication, the service issues JWT tokens in response to authenticated requests, ensuring secure and access to services. It also provides end point for jwt verification and decoding.

- **payments-service**: Handles payment transactions, supporting multiple payment methods for various use cases - such as when completed , or if purchase of a subscription plan is done.
- **utilities-service**: One of the core functionalities of the **utilities-service** is its robust notification system. It provides timely alerts to administrators in scenarios where the server is receiving responses at rates higher than expected or if any of the microservices experience downtime. This proactive monitoring ensures swift response to potential issues, maintaining system reliability and availability.

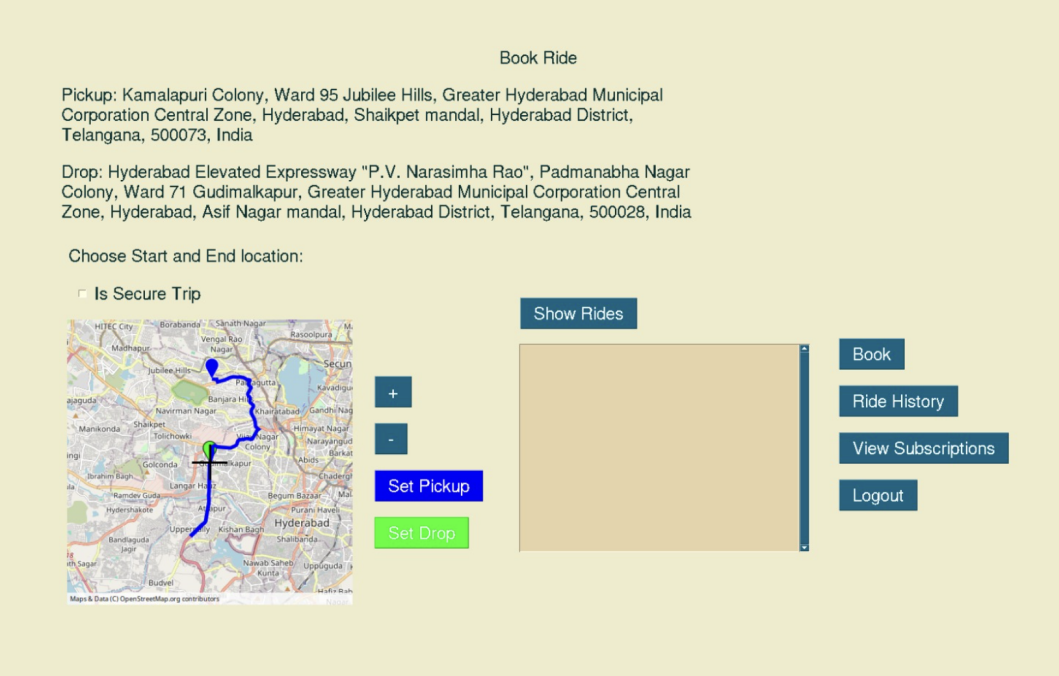
  The **utilities-service** is also responsible for keeping passengers informed about their ride status through timely email updates. It updates on ride confirmation, driver assignment, or arrival time estimates, passengers receive relevant and timely information.Utility services uses pub-sub architecture to provide robust notification service(both SMS and EMAIL).

  Beyond notifications, the **utilities-service** offers APIs for location-based functionalities. Users can obtain distance and duration estimates for their ride.We use third party APIs to get distance estimates between any two coordinates.
- **rides-service**: The **rides-service** orchestrates the entire ride booking process, managing the flow from a passenger creating a ride request to a driver completing the ride. The ride journey is divided into distinct stages, starting with **RIDE CREATION**, where passengers initiate ride requests through the platform.

  Both passengers and drivers have the flexibility to cancel rides before they are picked up, providing them with the necessary APIs.

  To ensure users have access to the best available options, the **rides-service** also fetches data from multiple cab providers. This data aggregation allows us to present users with a variety of choices, including different cab types, fare estimates, and arrival times estimation.

  Furthermore, the **rides-service** facilitates seamless interactions between passengers and drivers. Passengers can book rides based on their preferences and requirements, while drivers can accept pending ride requests and complete rides, ensuring efficient ride allocation and timely service delivery.



Given the use of multiple independently deployable microservices, we integrated an API Gateway to streamline interactions between clients and our microservices.

API Gateway:

Role:

The API Gateway serves as a unified entry point for external clients, handling request routing, aggregation, and providing additional functionalities like authentication and rate limiting.

Implementation:

- **Routing**: The API Gateway directs incoming requests to the appropriate microservice based on the endpoint or service URL, ensuring efficient request handling and response generation.
- **Authentication and Authorization**: Leveraging the Entity-Service, the API Gateway handles authentication and authorization. It validates JWT tokens before forwarding requests to microservices, ensuring only authorized users can access specific services.
- **Request Aggregation**: The API Gateway aggregates data from multiple microservices to fulfill a single client request, separating the client from the individual microservices, optimizing system performance, and enhancing user experience.

Architectural Tactics

Some of the architectural tactics we employed in our implementation are :

i. Performance Tactics → Caching by Redis

To enhance system performance and response times, we've implemented caching using Redis. By storing frequently accessed data in Redis, we reduce the load on our databases and accelerate data retrieval, resulting in faster response times and improved overall system performance.

We leverage Redis to cache subscription plans, user profiles, and frequently accessed ride data. This caching mechanism ensures that data retrieval for these entities is faster, contributing to a seamless user experience.Caching invalidation is also done on data modification.

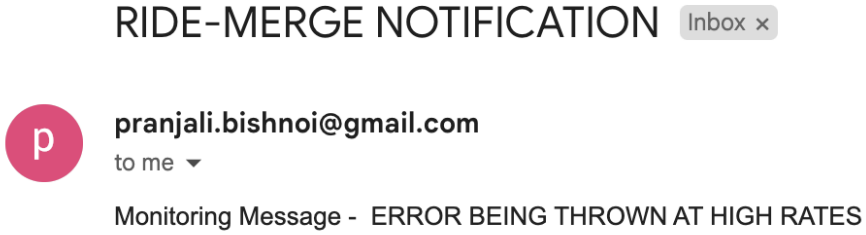
ii. Availability Tactics → Monitoring of Microservices and Alerts

Ensuring high availability is paramount to maintaining a reliable ride-sharing platform. To achieve this, we've implemented continuous monitoring of our microservices and set up alerting mechanisms to notify us of any service disruptions or performance anomalies.

```
def record(resp) :
    status = resp.get('status')
    if resp.get('error') or status !=200 :
        print("Error occured")
        cnt_errors = redis_client.get('cnt_errors')
        if cnt_errors:
            redis_client.set('cnt_errors', int(cnt_errors)+1)
        else:
            redis_client.set('cnt_errors', 1)
```

```
def test_monitoring():
    for i in range(100):
        data = {
            'password': 'driver1',
            'email': 'driver1',
            'phone' :'7992381519',
            'driving_license': '123456789'
        }
        response = requests.post(BASE_URL_ENTITY + 'register/driver', json=data)
    return
```

For simulation when we try to register the same driver multiple times , it throws error , and the admin is notified since the error thrown rate is high(we can set the limit on which admin should be notified ourself based on the expected load).



iii. Security Tactics → Authentication by Gateway

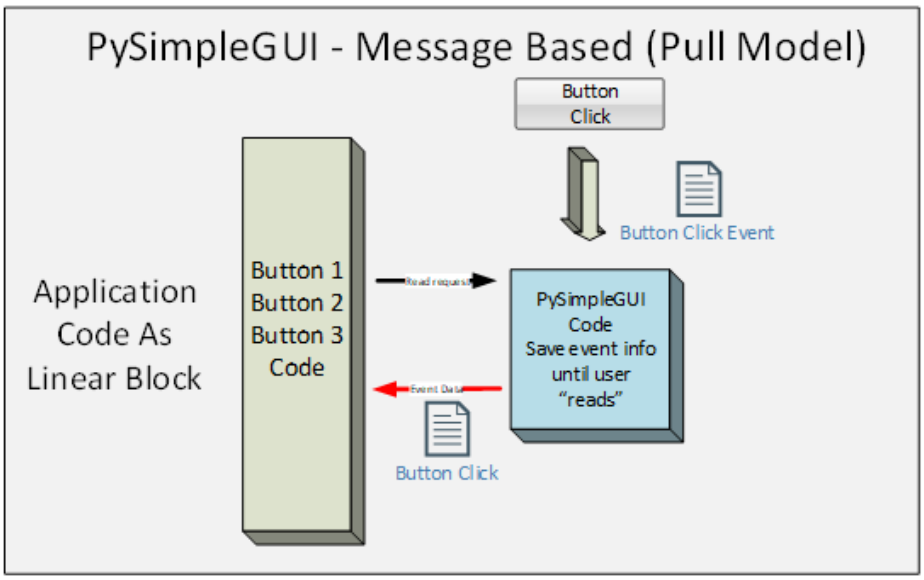
Our API Gateway handles authentication by validating JWT tokens issued by the **entitymanager-service**. Only requests with valid tokens are forwarded to the respective microservices, ensuring secure access and protecting our platform from unauthorized access and potential security threats.

```
def authenticate(request):
    jwt = request.get_jons().get('Authorization')
    response = requests.post(BASE_URL_ENTITY + 'verify/token', json={'token': jwt})
    if response.status_code == 200:
        return True
    else:
        return False       You, now • Uncommitted changes
```

iv. Usability Tactics → Providing User Features like Ride Cancelling

To enhance usability, we've incorporated user-centric features such as ride cancellation options to provide users with greater control over their ride bookings and minimising impacts of errors at run time.

GUI Architecture



PySimpleGUI and React are both popular frameworks used for building graphical user interfaces (GUIs), but they serve different purposes and follow different architectural patterns.

The app is built by defining a layout of GUI elements like buttons, text inputs and tables, etc. After defining the layout, the application enters an event loop where it waits for user interactions (which are raised as events) like button clicks, text input changes, etc. When an event occurs, the corresponding event handler is executed to handle the event.

PySimpleGUI does not have a built-in state management mechanism like React. Instead, we managed the application state using Python variables. When an event occurs, the event handler updates the state variables and then updates the GUI elements based on the new state (within the loop). This is vastly different from React as it uses a virtual DOM to optimize the rendering process.

Implementation Patterns

We will provide the design patterns we employed in each of the microservices :

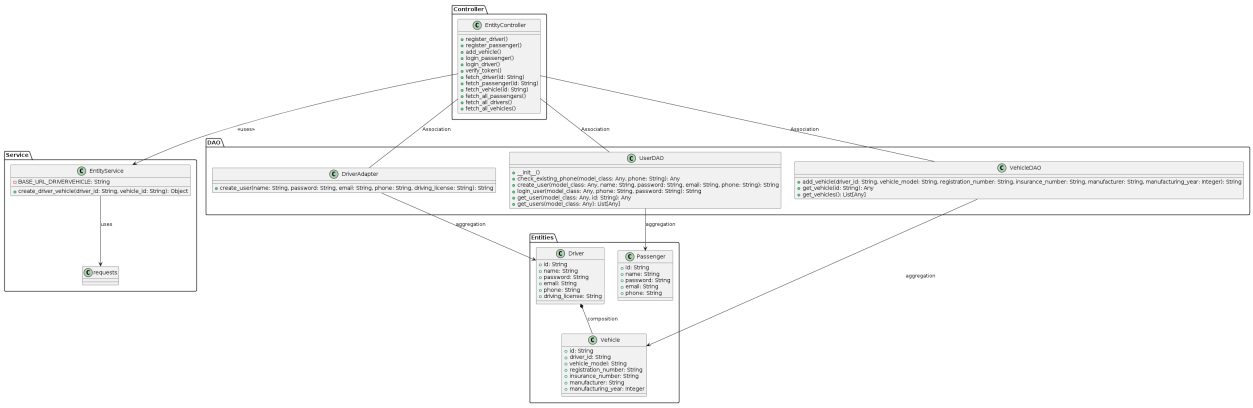
(i) EntityManager Service

Pattern Used : Adapter Pattern

In the EntityManager microservice , the Adapter pattern is being used in the `DriverAdapter` class. The purpose of the Adapter pattern here is to adapt the interface of the **UserDAO** class to a new interface that the client (or the caller) , which in our case is `EntityController` expects. In this case, the `DriverAdapter` class provides a method to create a driver user, which internally uses the `UserDAO` class to manage user-related operations.

1. **Adaptee**: The `UserDAO` class is the existing class with an interface that needs to be adapted.
2. **Adapter**: The `DriverAdapter` class acts as an adapter between the `UserDAO` class and the EntityController.

UML Diagram

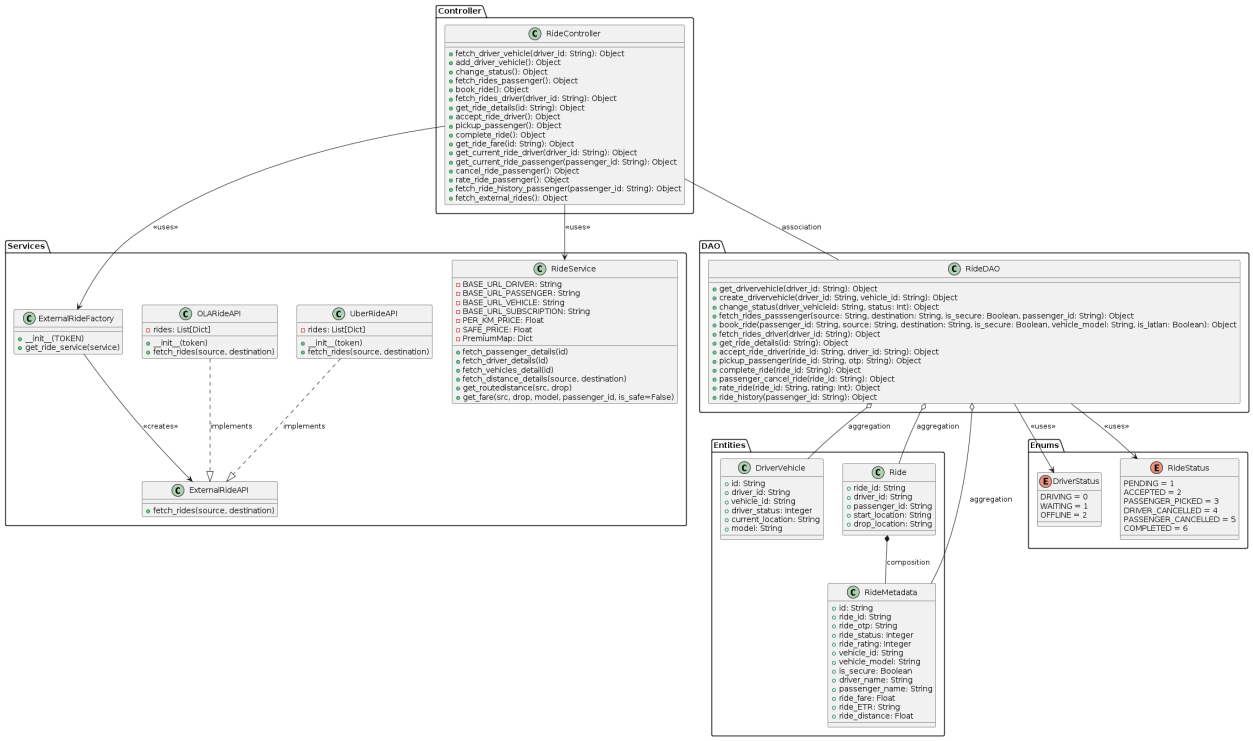


(ii) Rides Service

Pattern Used : Factory Pattern

In the Rides Service , we are using the factory pattern to create External Ride APIs such as OLA and UBER. RideController uses those APIs to fetch available rides for the given source and destination addresses by the user.

UML Diagram

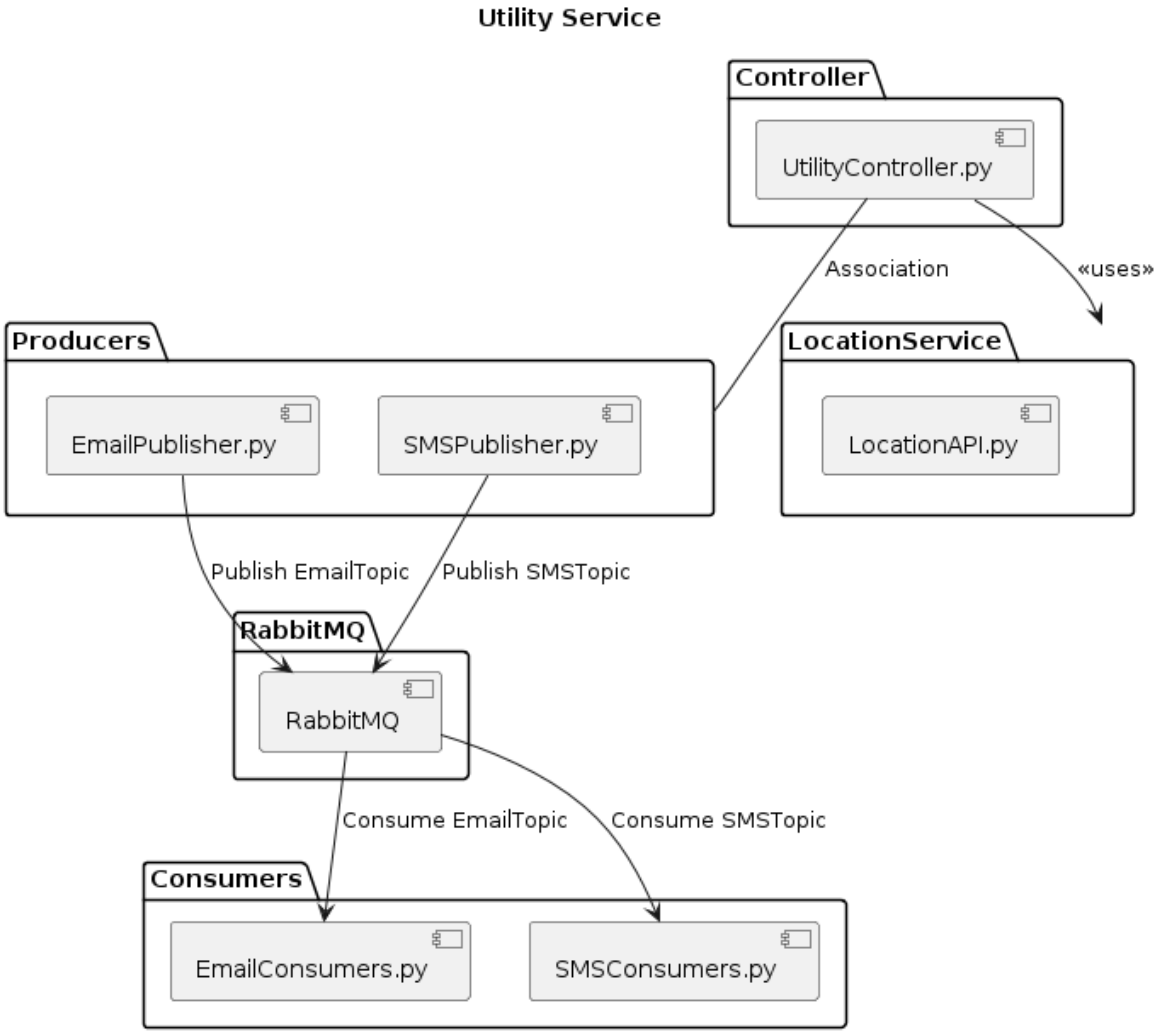


(iii) Utility Service

Utility Service provides two major functionalities : fetching distance/duration information from external API like distance-matrix , and also sending notification via email/message to the admin in case of high rate of server errors.It also sends notification to users on ride status change.

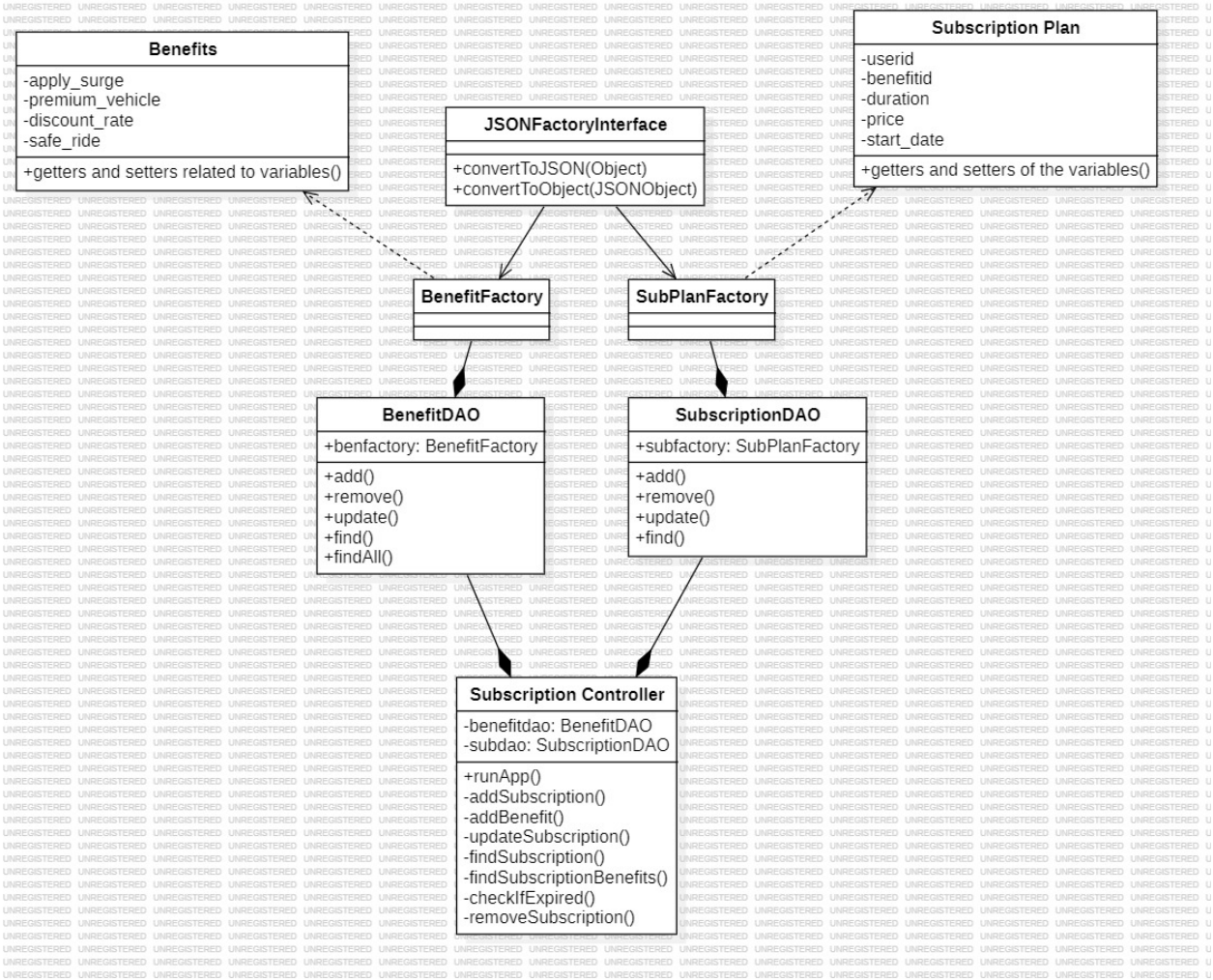
Pattern Used : Pub-Sub Pattern Using RabbitMQ.

UML Diagram



(iv) Subscriptions Service

UML Diagram



Description

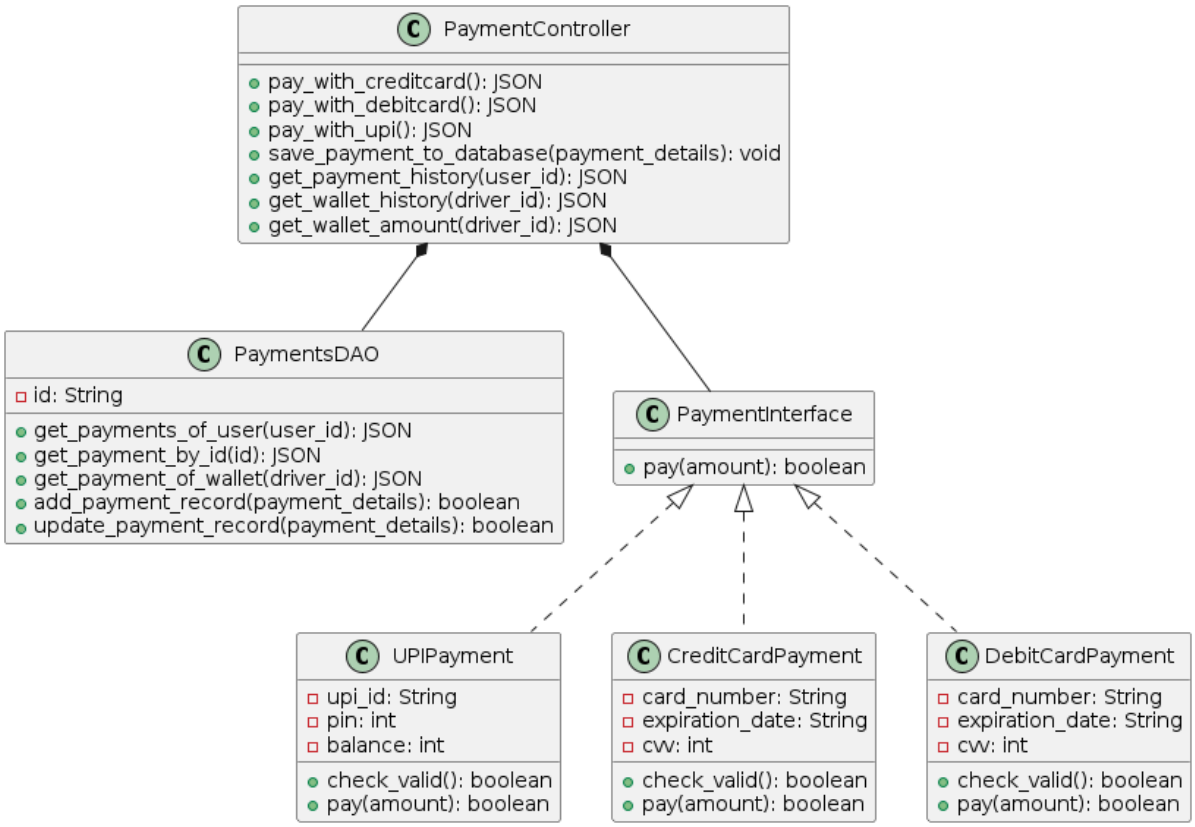
- Benefit objects stores the subscription plans that user can take. It includes all the benefits that the platform can offer.
- Subscription Plan objects stores the subscription taken by the user along with the price and the duration of the Plan as well as the start date of the plan. This takes into account the fact that Subscription Plan can also expire.
- DAO objects handle the database management of the two entities.

The pattern that is implemented in the Subscription Microservice is **Factory Pattern.** As shown above, in the process of converting objects of **Benefit** and **Subscription Plans**, we have replaced the direct object conversion with a factory called **BenefitJSONFactory** and **SubPlanJSONFactory** that handles the conversion of respective object(s) to JSONs and vice versa.

The Subscription Controller controls all the REST requests and is the front end for the Subscription Microservice.

(v) Payments Service

UML Diagram



Description

Payment system is implemented as follows. The passengers can pay via one of the 3 methods - UPI, Debit Card, Credit Card. For each payment made, an SQL database recording the payments is accessed to store its corresponding user id, ride id, driver id, amount paid, payment method, etc.

Payment for subscriptions is also made via the same system.

The payment microservice has been implemented using strategy pattern using the following:

- Separate classes for each payment method, implementing a common Payment Interface.
- Payment Controller class to control all the REST requests.
- Payment DAO class to access the corresponding objects in the database.

Task 4: Prototype Implementation and Analysis

Prototype Development

We have developed a working prototype and uploaded the code on github. Due to resource constraints , we took the following two assumptions :

(i)- Since OLA/Uber APIs are not available publicly, we fetch dummy external rides for them.

(ii)- Current Vehicle Location is not shown. Note that distances and expected time durations are shown using third party API(https://distancematrix.ai/).

(iii)- Information verification for drivers is not implemented for now, since that involves multiple components including manual verification by the service provider.

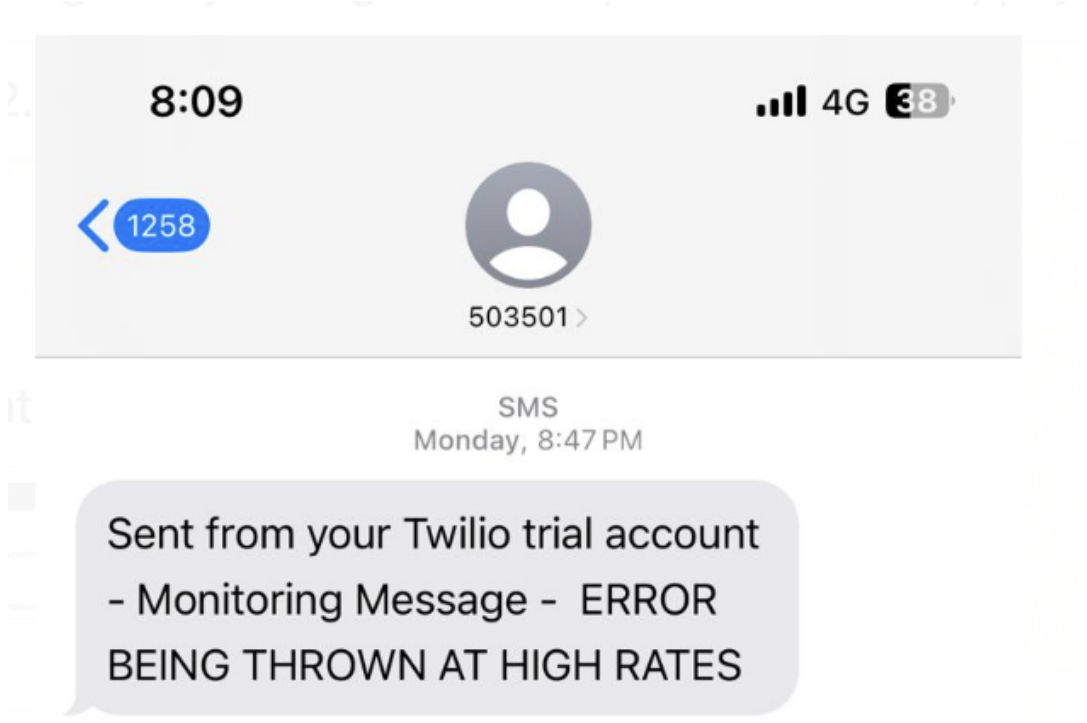
All other features for both drivers as well as passengers are fully implemented.

Frontend Implementation :

TODO

Architecture Analysis

Example : Notification by Mobile to ADMIN.



We implemented the utilitity microservices(notification part) in two ways : by using Layered Architecture(monolithic) and by using event driven pub-sub architecture.

Event-Driven Pub-Sub Architecture Implementation

Components

1. **Controller**
    - **UtilityController.py**: Manages the overall flow and interaction between publishers and consumers.
2. **Consumers**
    - **EmailConsumers.py**: Listens to the `EmailTopic` queue for incoming messages and sends emails to the specified receivers.
    - **SMSConsumers.py**: (Assumed based on the directory structure) Listens to the `SMSTopic` queue for incoming messages and sends SMS messages to the specified recipients.
3. **Publishers**
    - **EmailPublisher.py**: Publishes messages to the `EmailTopic` queue, triggering email notifications.
    - **SMSPublisher.py**: (Assumed based on the directory structure) Publishes messages to the `SMSTopic` queue, triggering SMS notifications.

For sending phone messages , we are using Twilio API.

Comparasion

We get the following result on testing the utility service(notification) :

PubSub Architecture

Throughput	32.12696661917674 requests/second
Average latency	0.031125998497009276 seconds

Layered(Monolithic Architecture)

Throughput	0.2509079544601211 requests/second
Average latency	3.985519766807556 seconds

Trade-offs Involved

- **Scalability**:
    - **Event-Driven Pub-Sub Architecture**: The pub-sub architecture shines in scalability, especially for systems with unpredictable workloads. Its distributed nature and asynchronous processing allow for seamless scaling by adding more subscribers or

message brokers. This makes it particularly suitable for high-volume and dynamic environments.

    - **Layered (Monolithic) Architecture**: On the other hand, monolithic architectures often struggle with scalability. As the application grows, scaling becomes a challenge due to the centralized nature of the codebase and database. The lack of `asynchronous processing` in many monolithic systems limits their ability to handle concurrent requests efficiently.
        - **Note**: `Asynchronous processing` is extremely beneficial in scenarios where tasks can be executed independently and don't require immediate results. For instance, in notification systems, sending a notification doesn't necessarily need to be done synchronously with the user action. However, in scenarios requiring tight coupling or immediate response, asynchronous processing may not be the best fit.

- **Complexity**:
    - **Event-Driven Pub-Sub Architecture**: Pub-Sub Architecture might introduce complexity in terms of message ordering, event consistency, and error handling. However, it offers greater flexibility by decoupling components, making it easier to maintain and evolve the system over time.
    - **Layered (Monolithic) Architecture**: Monolithic architectures are generally easier to grasp initially, as all components are tightly integrated within a single codebase. However, as the application grows, the complexity also grows exponentially. Changes in one part of the system can have unintended consequences elsewhere, making it challenging to maintain and extend the application.
- **Latency**:
    - **Event-Driven Pub-Sub Architecture**: The pub-sub architecture excels in delivering low-latency responses by leveraging message queues. Messages are processed independently and can be handled in parallel, leading to faster response times, which is crucial for real-time notification delivery.
    - **Layered (Monolithic) Architecture**: Due to its centralised nature, monolithic architectures often suffer from higher latency, especially under heavy loads. The synchronous processing of requests can lead to queuing delays and longer response times, making it less suitable for real-time applications where immediate feedback is required.

Monolithic (Layered) Implementation

Components

1. **Controller**
    - **UtilityController.py**: Manages the overall flow and interaction between services and utilities.
2. **Services**

- **NotificationService.py** : Contains common interface for both email and message.
- **EmailService.py** : Manages email notifications.
- **SMSService.py** : Manages SMS notifications

3. **Models**
    - **NotificationModel.py**: Defines the data structure for notifications.

Similar to the Event-Driven Pub-Sub Architecture Implementation, we used `smtplib` Python library is used to send emails and Twilio APIs for sending SMS.

Conclusion

Both architectural approaches have their strengths and weaknesses, and the choice between them should be based on the specific requirements, constraints, and future growth plans of the system. While the Event-Driven Pub-Sub Architecture offers better scalability, lower latency, and greater flexibility, it comes with the complexity of asynchronous messaging and event-driven design. On the other hand, the Layered (Monolithic) Architecture may be easier to understand and develop initially but can become a bottleneck in terms of scalability and latency as the application grows.

Lessons Learnt

1. Designing a real-time ride booking platform presented a higher level of complexity than we expected. Real-time systems includes multiple variables and cases which needs to be taken care of.
2. Utilising microservices and event-driven architecture added flexibility and scalability to the system. However, it also introduced challenges in terms of service orchestration, data consistency, and communication between services.
3. Working as a team of five required effective communication and collaboration. The major issue was integrating all of the microservices, which took us some time.
4. We underestimated the amount of effort and time required to implement the various features and ensure the stability and correctness ,majorly for the ride creation and completion part.

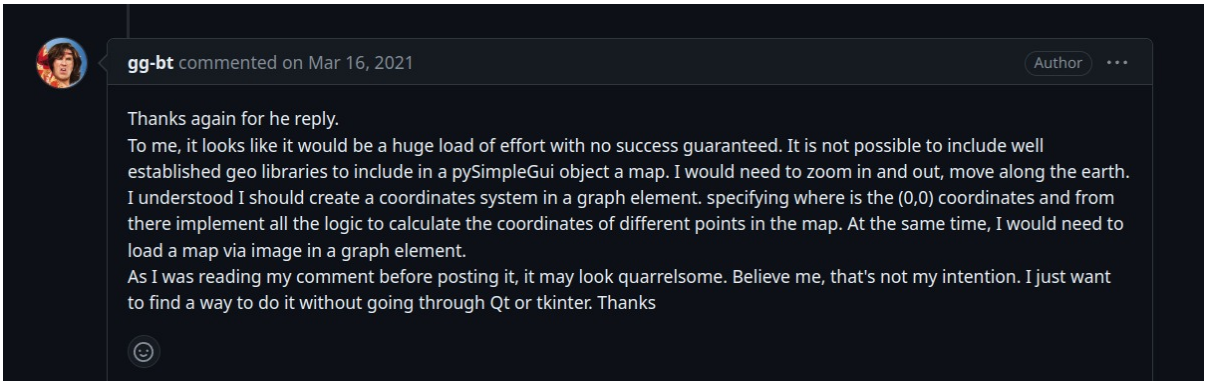
Achievement

1. For frontend , we decided to use **PySimpleGUI** , which is a thin wrapper around tkinter . For fetching the map , there was no API available which provides feature for scrolling. So , we decided to implement it on our own . This was mainly done by Soham. The tiles for maps are fetched from OpenStreetMap , and its being merged by our algorithm at frontend. Then our

algorithm calculates optimal number of tiles to be downloaded based no the parameters (centre location,zoom level). The tiles are cached locally to reduce multiple requests to OSM API.

We are planning to push it to PIP repository for python , since no such library is available right now, and there are multiple demands for this.

Github Comment for reference



https://github.com/PySimpleGUI/PySimpleGUI/issues/4044

https://github.com/PySimpleGUI/PySimpleGUI/issues/6276

Team Contributions

Member	Contribution
Mohammad Zaid	Rides Microservice , EntityManager Microservice
Pranjali Bishnoi	Utility Microservice
Aman Raj	Subscription Microservice
Shambhavi Jahagirdar	Payment Microservice
Soham Korade	Frontend designing and integration