RIDE-SHARING PLATFORM UBER LYFT
System Design Sketch

Functional Requirements
1. Rider gets estimated fare (by start and destination locations)
2. Rider requests a ride based on the estimates
3. Driver accepts/Denies request + Navigate to pickup/drop-off

Below the line(out of scope)
1. Rider should be able to rate their ride and drive post-trip.
2. Drivers should be able to rate passengers
3. Riders should be able to schedule rider in advance
4. Riders should be able to request different categories of ride(X,XL,Comfort).

Non-Functional Requirements
1. Low Latency for rider-driver matching (<1min)
2. Consistency of rider-driver 1:1 matching
- For one ride, only one driver will get request at a time
3. Highly available outside matching
4. Handle high-throughput (peak hours, popular events)

Below the line(out of scope)
1. The system should ensure the security and privacy of user and driver data, complying with regulation like GDPR.
2. The system should be resilient to failures, with redundancy and failure mechanisms in place.
3. The system should have robust monitoring, logging and alerting to quickly identify and resolve issues.
4. The system should facilitate easy updates and maintenance without significant downtime.

Core Entities
1. Rider
2. Driver
3. Ride
3.1. Fare: estimated fare for ride, include pickup and destination location, estimated fare and estimated time of arrival
4. Location: stores real-time locaiton of drivers.

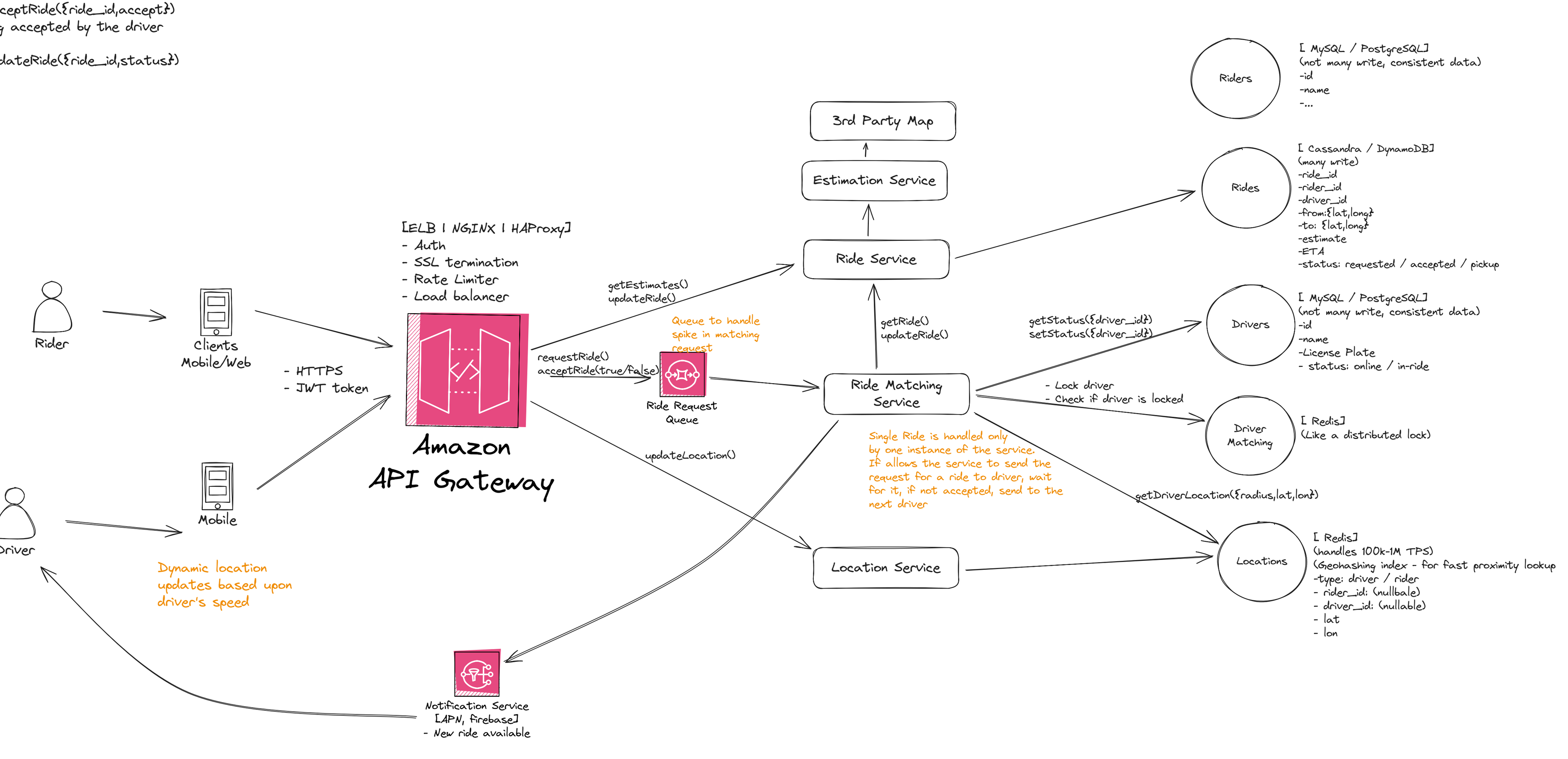
API:
- POST /getEstimate({from,to}) --> Partial<ride>
- POST /requestRide({ride_id}) --> Ride
- POST /getRide({ride_id})

- gets the latest ride entity to display the ride details for both rider and driver app

- POST /updateLocation({lat,lon})
- both driver and rider can update their locations which may be use to display their on the map
- drivers might update their location i.e. once every 5 seconds(it is needed for matching)
- however this can be optimized on the client side, where we might say that location update is propotional to driver speed:
for still drivers we update location rarely, for moving - more often

- POST /acceptRide({ride_id,accept})
-ride is being accepted by the driver

- POST /updateRide({ride_id,status})

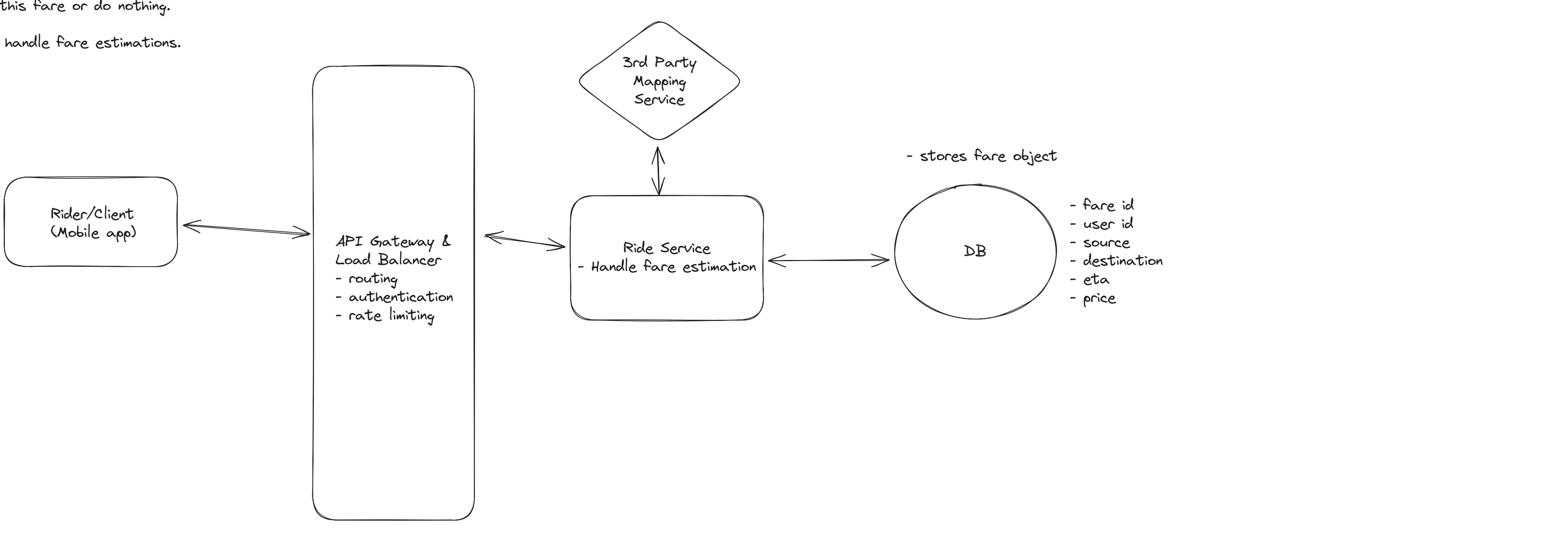


High level Design

1. Rider should be able to input a start location and a destination and get an estimated fare.

POST /fare -> Fare
Body: {
pickupLocation,
destination
}
Client will make a request to our service to get an estimated price for ride, the user will then have a chance to request a rider with this fare or do nothing.

Ride Service: handle fare estimations.



The core components necessary to fulfill fare estimation are:
1. Rider Client: This client interfaces with the system's backend services to provide and seamless user experience.
2. API Gateway: Acting as the entry point for the client requests, the API gateway routes requests to the appropriate microservices.
It will manages cross-cutting concerns such as authentication and rate limiting.
3. Rider Service: This microservice is tasked whith managing ride state, starting with calculating fare estimates. It interacts with third-party mapping APIs to determine the distance and travel time between locations and applies the company' pricing model to generate a fare estimates.
4. Third Party Mapping API: We use a third-party service to provide mapping and routing functionality. It is use by Ride service to calculate the distance and travel time between locations.
5. Database: The database is, so far, responsible for storing Fare entitiess. In this case, it create a fare with information about price, eta,etc.

Component interaction.
1. The rider enter their pickup locaiton and desired destination into the client app, which sends a POST request to our backend system via /fare.
2. The API gateway receives the request and handles any necessary authentication and rate limiting before forwarding the request to ride service.
3. The Rider service makes request to Third party Mapping API to calculate the distance and travel time between pickup and destination locations.
and then applies the company pricing model to distance and travel time to generate fare estimates.
4. The Ride service creates a new Fare entity in the database with the details about estimated fare.
5. The service then returns the fare entity to the API gateway, which forwards it to the Rider client so they can make a decision about whether accept the fare and request a ride.

As soon user accepts the request, client will make a request to create ride
Table: Ride
rideId
riderId
driverId
fareId
source
destination
etc

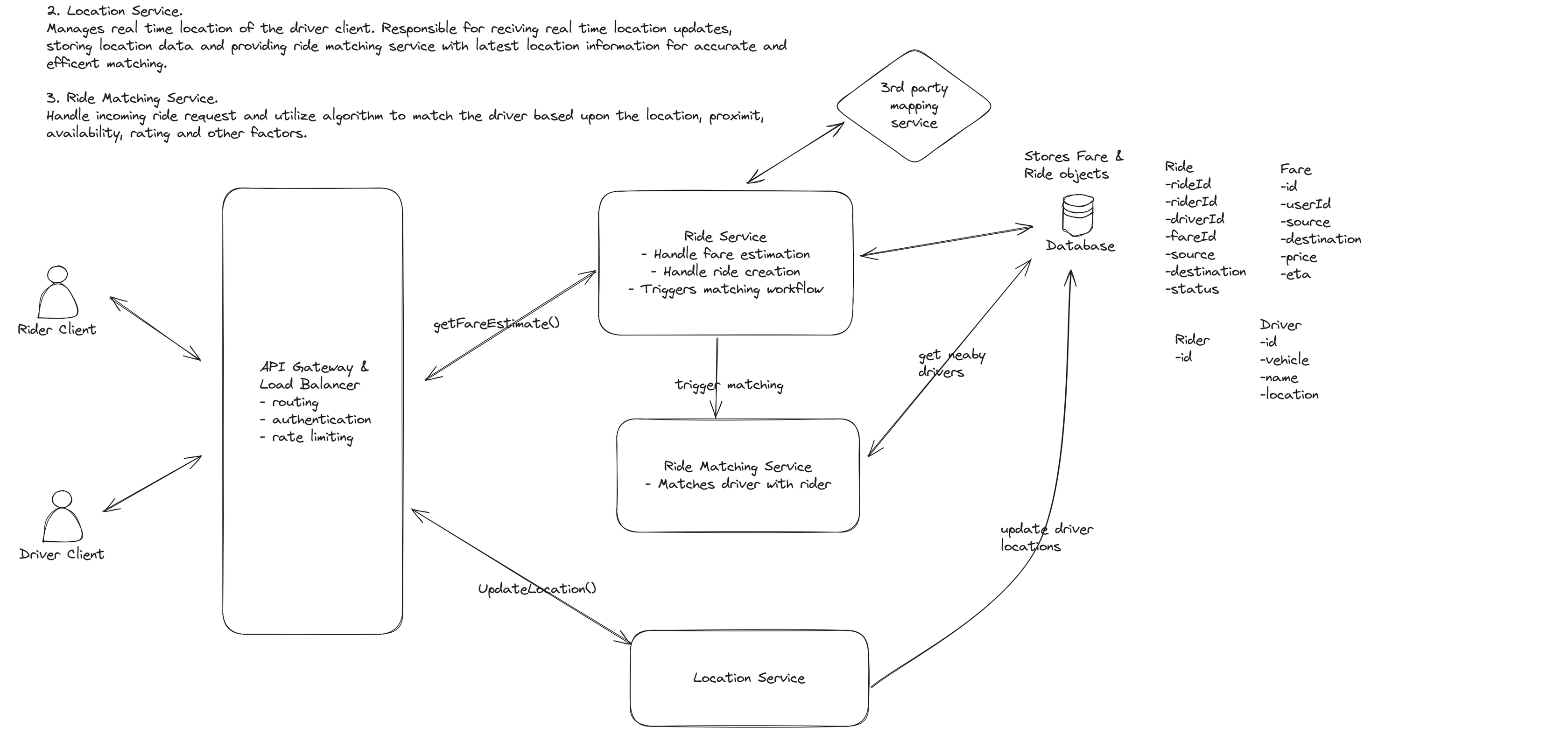
The user confirm their request in their client application, which sends POST request to our backend system,
with the fareId user has accepted.
The API gateway will perform the necessary authentication and rate limiting before forwarding request to ride service.
The ride service receives the request and create ride record, linking the fare id accepted by the user and set status as requested.
Next, it will trigger matching flow so that we can assign driver for the request.

Upon request, the rider should be matched with the driver who is near by and available.

1. Driver Client.
Similar to rider client, we will have driver client that will allow drivers to accept the ride request and share the real time location of the driver. The driver client will interact with Location service to provide real time updates.

2. Location Service.
Manages real time location of the driver client. Responsible for reciving real time location updates, storing location data and providing ride matching service with latest location information for accurate and efficent matching.

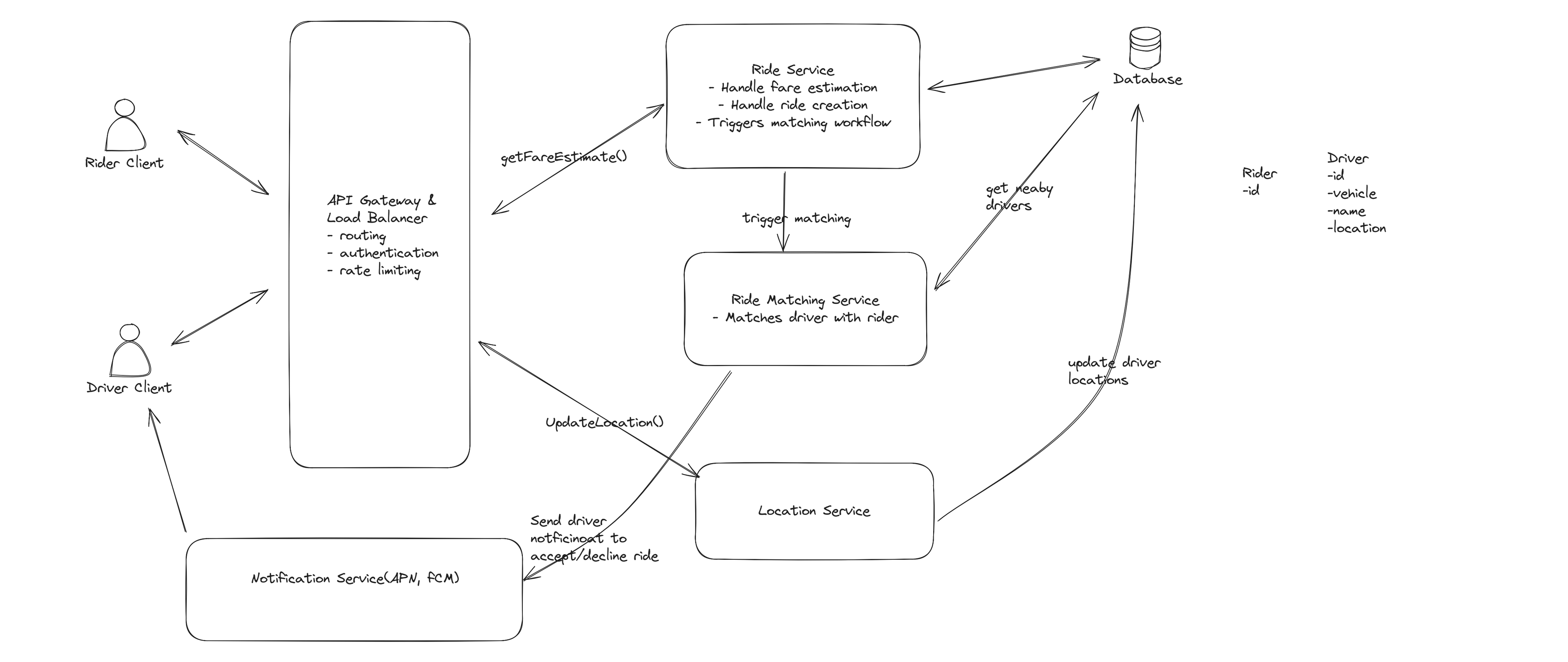
3. Ride Matching Service.
Handle incoming ride request and utilize algorithm to match the driver based upon the location, proximit, availability, rating and other factors.



Sequence of events that occurs when user requests a ride and ssystem matches them with a driver.
1. The user confirm the ride request in the client app, which sends a POST request to our backend system with the ID of the fare they are accepting.
2. The API gateway performs necessary authentication and rate limiting before forwarding the request to Ride Matching Service.
3. We create a ride object and then trigger the matching wrokflow.
4. Meanwhile, all the times, drivers are sending their current location to the location service, and we are updating our database with their latest location lat & log so w know where they are
5. The matching workflow then uses these updaed locations to query for the closest available driver in an attempt to find an optimal match.

Drivers should be able to accept/decline a request and navigates to the pickup/drop=off.

1. Notification Service:
Responsible for dispatching real-time notifications to drivers when a new ride request is matched to them. It enxure that drivers are promptly informed so they can ride requests in a timely manner, thus maintaining a fluid user experience.



Sequcne of events occur when a driver accepts a ride request and completes the ride:
1. After the Ride matching service determines the ranked list of eligible drivers, it sends a notification to the top driver on the list.
2. The driver receives a notification that a new ride request is available. They open the Driver Client app and accept the ride request, which send a PATCH request to the backen system with rideID.
If they decline the ride instead, the ysstem will send a notification ot the next driver in the list.
3. The API gateway receives the requests and routes it to the Ride Service.
4. The Ride Service receives the request and updates the status of the ride to "accepted" and updates the assigned driver accordingly. It then returns the pickup lcoation coodinates to the Driver client.
5. With the coordinates in hand, the Driver uses on client GPS to navigates to the pickup locations.

Potential Deep Dives

How do we handle frequent driver location updates and efficient proximity searches on location data?

Managing the high volume of location updates from drivers and performing efficient proximity searches to match them with nearby ride requests is a difficult task, and our current high-level design most definitely does not handle this well.
Wo main problem with our current design that we need to solve:
1. High Frequency of Writes.
Given we have 10M drivers, sending location roughly every 5 sec, that about 2M updates per second.
2. Query Efficiency:
Without any optimizations, to query a table based on lat/long we would need to perform a full table scan, calculating the distance between each, driver's location and rider's location. This extremely inefficient, especially with millions of drivers.

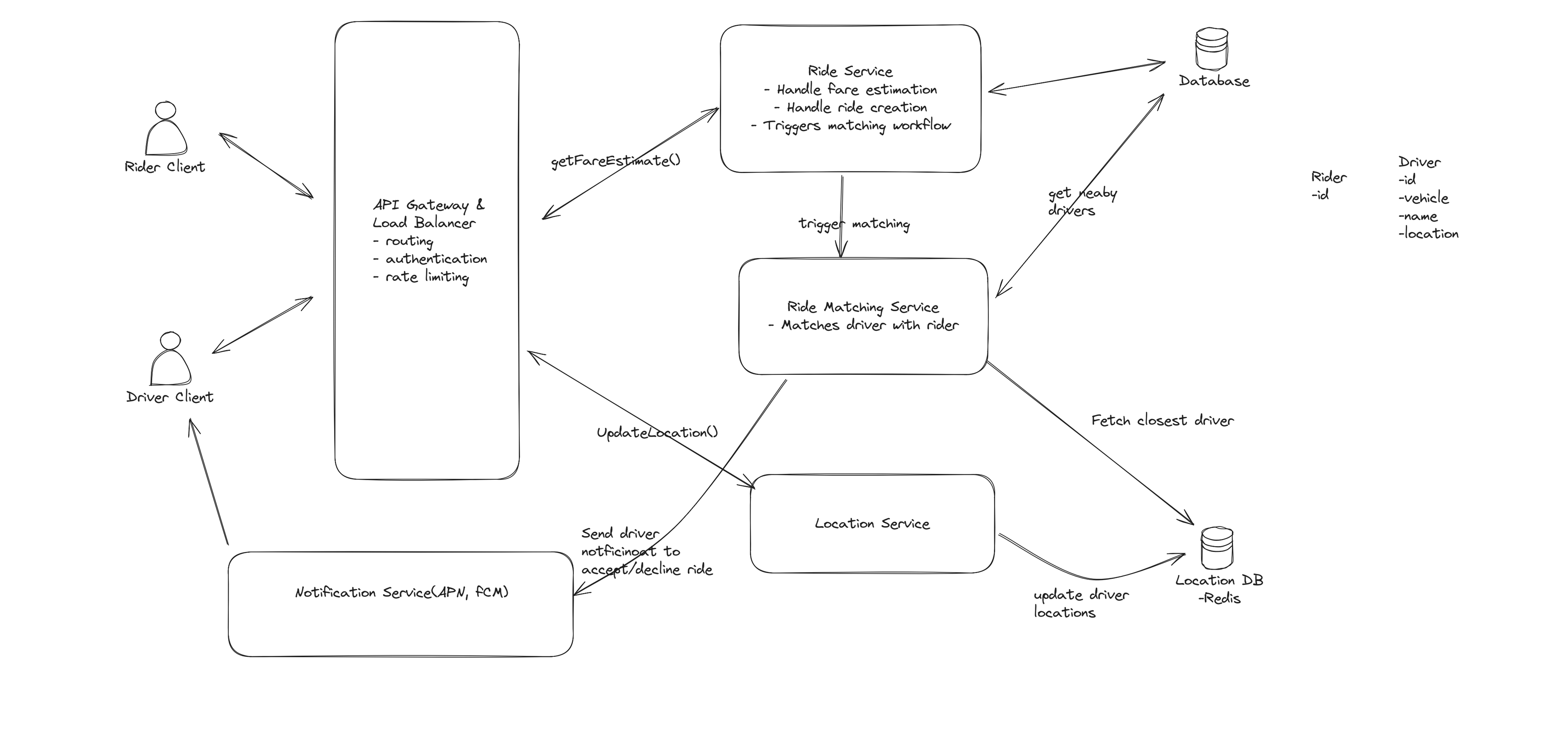
Efficient Solution: Real-Time In-memory Geospatial Data store.

Approach: We can address all the limitation of the previous solution by using an in memory data store like Redis, which supports geospatial data types and commands. This allow us to handle real-time driver location updates and proximity searches with high throughput and low latency while minimizing storage cost with automatic data expiration.

Redis is an in-memory data store that supports geospatial data types and commands. It uses geohashing to encode latitude and longitude cooridnates into a single string key, which is then indexed using sorted set. This allows for efficient storage and querying of geospatial data.
Redis provides geospatial commands like GEOADD for adding location data and GEOSEARCH for querying nearby locations within a given radius or bounding box.

We no longer have a need for batch processing since Redis can handle the high volume of location updates in real-time. Additionally, Redis automatically expires data based on the specified time to live, which allows us to retain only the most recent location updates, and avoid unnecessary storage costs.

Challenges
The main challenge with this approach would be durability. Since Redis is an in-memory data store, there's a risk of data loss in case of a system crash or power failure. However, this risk can be mitigated in a few ways:
1. Redis persistence: We could enable Redis persistence mechanisms like RDB(Redis Database) or AOF(Append only File) to periodically save the in-memory data to disk.
2. Redis Sentinel: We could use Redis Sentinel for high availability. Sentinel provides automatic failover if the master node goes down, ensuring that the replica is promoted to master.



How can we manage system overload from frequent driver location updates while ensuring location accuracy?

High-frequency location updates from drivers can lead to system overload, straining server resources and network bandwidth. This overload risks slowing down the system, leading to delayed location updates and potentially impacting user experience. In most candidates original design, they have drivers ping a new locaiton every 5 sec or so.
This follow up question is designed to see if they an intelligently reduce the number of pings while maintaining accuracy.

Approach: Adaptive Location update Intevals.
We can address this issue by implementing adaptive location update intervals, which dynamically adjust the frequency of location updates based on contextual factors such as speed, direction of travel, proximity of pending ride requests, and driver staus. This allow us to reduce the number of location updates while maintaining accuracy.

The driver's app uses on device sensors and algorightm to determine the optimal interval for sending lcoation updates.

CHallenges
The main challenge with this approach is the complexity of designing effective algorithms to determine the optimal update frequency. This requires careful consideration and testing to ensure accuracy and reliability. But, if done well, it will significanlty reduce the number of location udpates and improve efficiency.

How do we prevent multiple ride request from being sent to the same driver simultaneously?

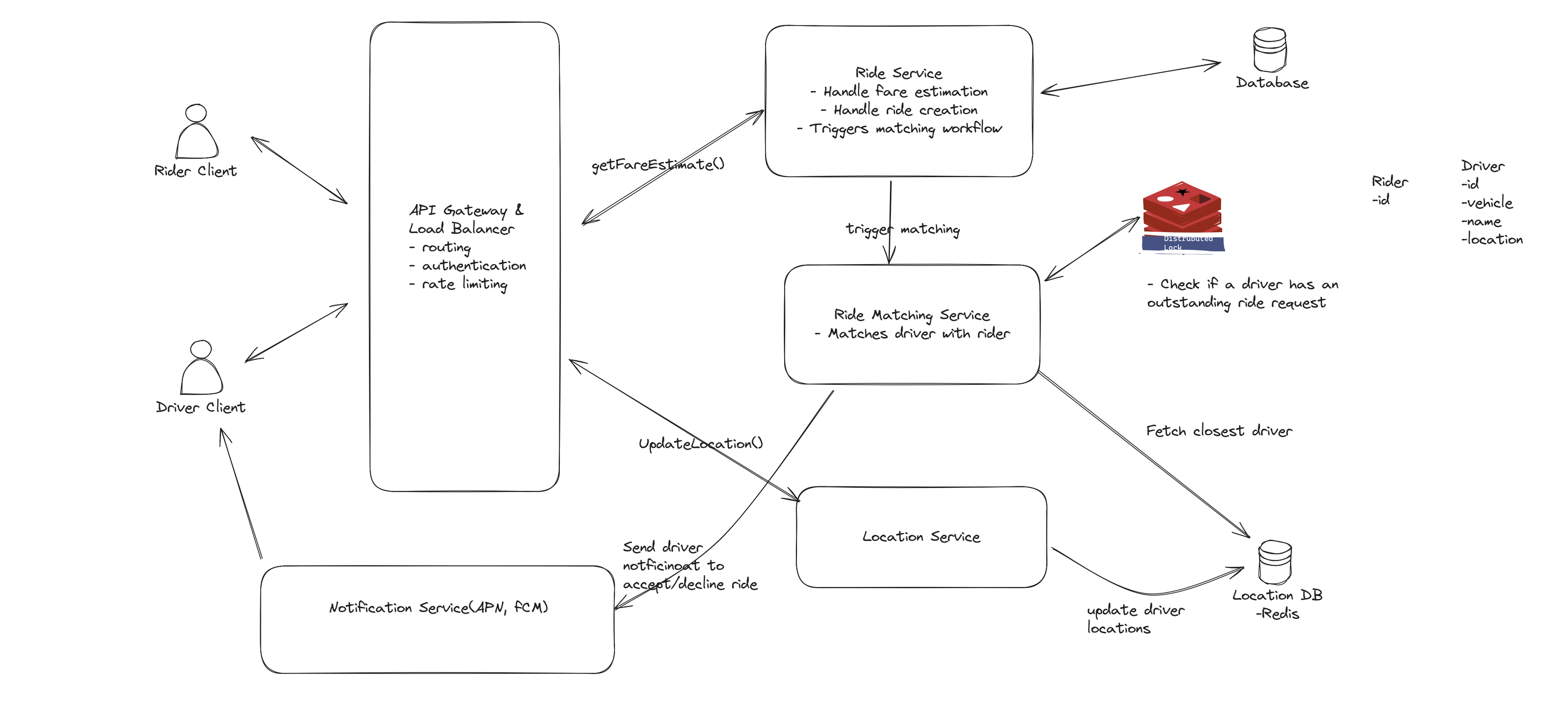
We defined consistency in ride matching as a key non-functional requirement. This means that we only request one driver at a time for a given ride request and that each driver only receives one ride request at a time. That driver would have 10 sec to accept or deny the request before we move on to the next driver.

Bad Solution : Applicaiton-leve location with Manual Timeout checks
Problems: - Lack of Coordination
- Inconsistent Lock state.
- Scalability Concerns

Good Solution : Database status UPdate with timeout handling
Problems: If ride service crashes or is restarted, the timeout will be lost and the lock will remain indefinitely.

Great Solution : Distributed Lock with TTL.
To solve the timeout issue, we can use distributed lock implemented with and in-memory data store like Redis. When a ride request is sent to a driver, a lock is created with a unique identifier and a TTL set to acceptance window duration of 10 sec. The Ride matching service attempts to acquire a lock on the driverID in Redis. If the lock is successfully acquired, it means no other service instance can send a ride request to the same driver until lock expires or is released. If the driver accept the ride within TTL window, the ride matching service updates the ride status to "accepted" int he database, and lock is released in Redis. If the driver doesnot accept the ride within the TTL window, the lock in Redis expires automatically. This expiration allows the Ride matching service to consider the driver for new ride requests again.

Challenges
The main challenge whith this approcah is the system's reliance on the availabillity and performance of the in-memory data store for locking. This requires robust monitoring and failover strategies to ensure that the system can recover quickly from failure and that locks are not lost or corrupted. Given locks are only held for 10 sec, this is reasonable tradeoff as the emphemorality fo the data makes it easier to recover from failure.



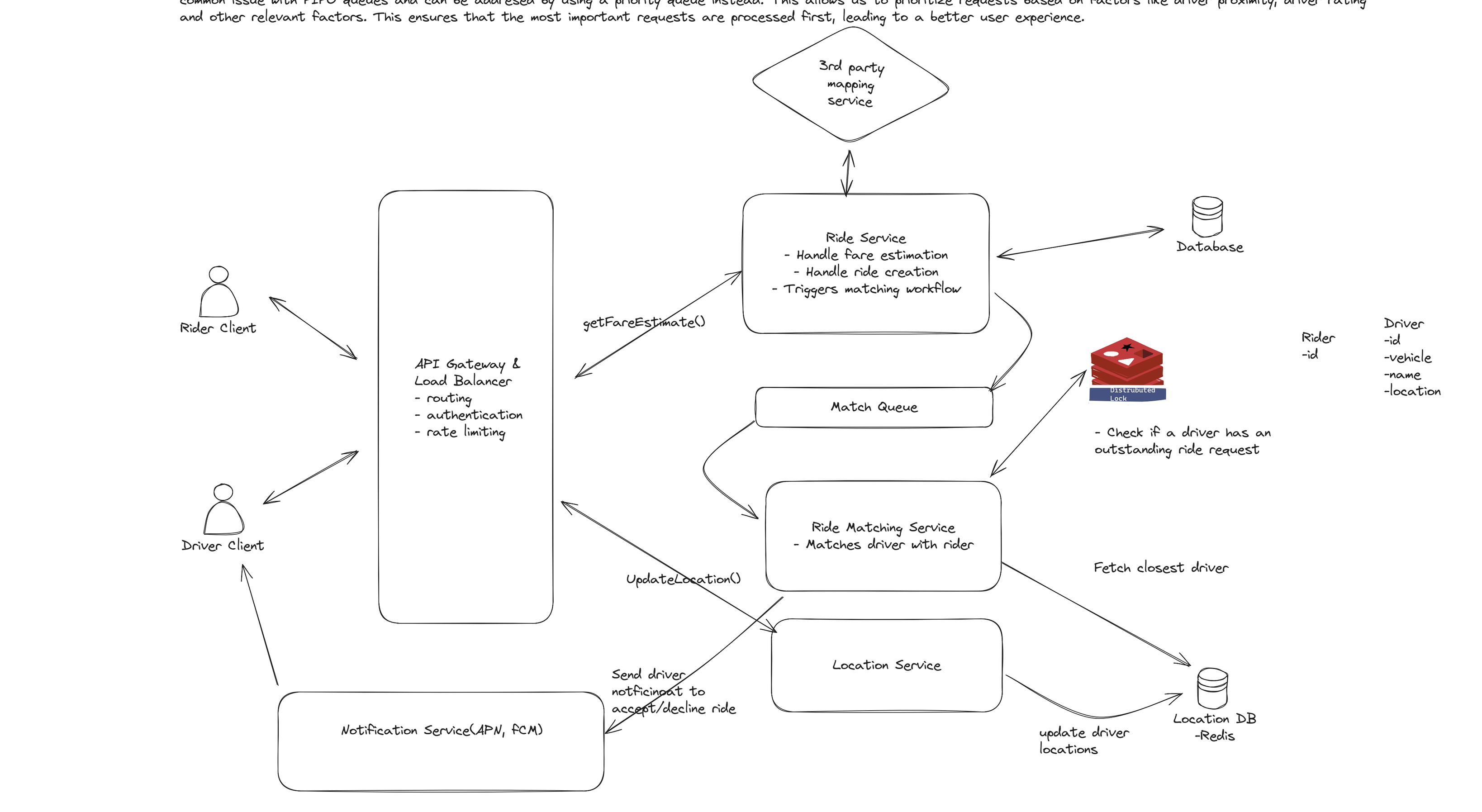
How can we ensure no ride requests are dropped during peak demand periods?

During peak demand periods, the system may receive a high volume ride request which can lead to dropped requests. This is particular problematic during special events or holidays when demand is high and the system is under stress. We also need to protect against the case where an instance of Ride matching service crashes or is restarted, leading to dropped rides.

Bad Solution : First-come, first served with no queue
Problem: It doesnot scale well during high-demand periods.
Additionaally, if a ride matching service goes down, any request that were being processed by that instance would be lost.

Great Solution : Queue with Dynamic Scaling.
To Address this issue we can introduce queuing system with dynamic scaling. When a ride request comes in, it is added to the queue. The Ride matching service then processes requests from the queue in a first-come, first served manner. If the queue grows too large, the system scales horizontally by adding more instances of Ride Sharing service to handle the increased load. This allow us to scale the system dynamically based on demand, ensuring that no requests are dropped. We can also partition the queues based on geographic regions to further improve efficency.
We could use a distributed message queue system like kafka, which allows us to commit the offset of the message in the queue only after we have successfully found a match.
This way, if the Ride matching service goes down, the match request would still be in the queu, and a new instance of the service would pick it up. This approach ensures that no ride request are lost due to service failure and provides fault tolerance to our system.

Challenges
The main challenge with this approach is the complexity of managing a queueing system. We need to ensure that the queue is scalable, fault tolerant and highly available. We can address this by using a managed queuing service like amazon SQS or Kafka, whihc provies these capabilities out of the box. This allows us to focus on the business logic of the system without worrying about the underlying infrastructure.
The other issue with this approach is that since it is a FIFO queue you could have requests that are stuck behind a request that is taking a long time to process. This is common issue with FIFO queues and can be addresed by using a priority queue instead. This allows us to prioritize requests based on factors like driver proximity, driver rating and other relevant factors. This ensures that the most important requests are processed first, leading to a better user experience.



5. How can you further scale the system to reduce latency and improve throughput?

Bad Solution : Vertical Scaling
Great Solution : Geo-Sharding with Read Replicas
A better solution is to scale horizontally by adding more servers. We can do this by sharding our data geographically and using read replicas to improve read throughput. This allows us to scale system to handle more requests while reducing latency and improving throughtput. Importantly, this not only allows us to scale, but it reduces latency by reducing the distance between the client and the server. This applies to everything from our services, message queues, to our databases all of which can be sharded geographically. The only time that we would need to scatter gather is when we are doing a proximity search on a boundary.
Challenges:
The main challenge with this approach is the complexity of sharding and managing multiple servers. We need to ensure that data is distributed evenly across shards and that the system can handle failures and rebalancing. We can address this by using consistent hashing to distribute data across shards and by implementing a replication strategy to ensure that data is replicated acrosss multiple servers. Thsi allows us to scale the sytem horizontally while maintaining fault tolerance and high availability.