Smart Cab Allocation System

Introduction

SmartRide is an innovative platform designed to streamline cab allocation for administrators while elevating the overall user experience. Utilizing state-of-the-art algorithms and real-time data tracking, SmartRide establishes itself as a trailblazing solution. The platform introduces a proximity-based algorithm for administrators, reducing travel distances and providing employees with a seamless interface that offers real-time suggestions for nearby cabs. This cutting-edge design seamlessly incorporates real-time location data into the system, enhancing the efficiency of cab allocation.

Project Overview

The SmartRide Allocation System caters to three primary user roles: Riders, Drivers, and Administrators, each equipped with tailored functionalities to meet their specific requirements.

Riders:

- Cab Booking: Riders can easily browse available cabs, view their locations, and book a cab for their desired destination with ease.
- Trip History: Access a comprehensive log of previous trips, offering detailed insights into past bookings, routes taken, and fare details for better record-keeping.

Drivers:

- Trip Acceptance and Rejection: Drivers have the flexibility to accept or decline trip requests based on their availability and location, allowing for better schedule management.
- Real-Time Navigation: Drivers receive real-time navigation assistance, optimizing routes and estimated arrival times for efficient trip completion.
- Performance Analysis: Access to a dashboard displaying trip history and performance analytics empowers drivers with valuable insights for ongoing improvement.

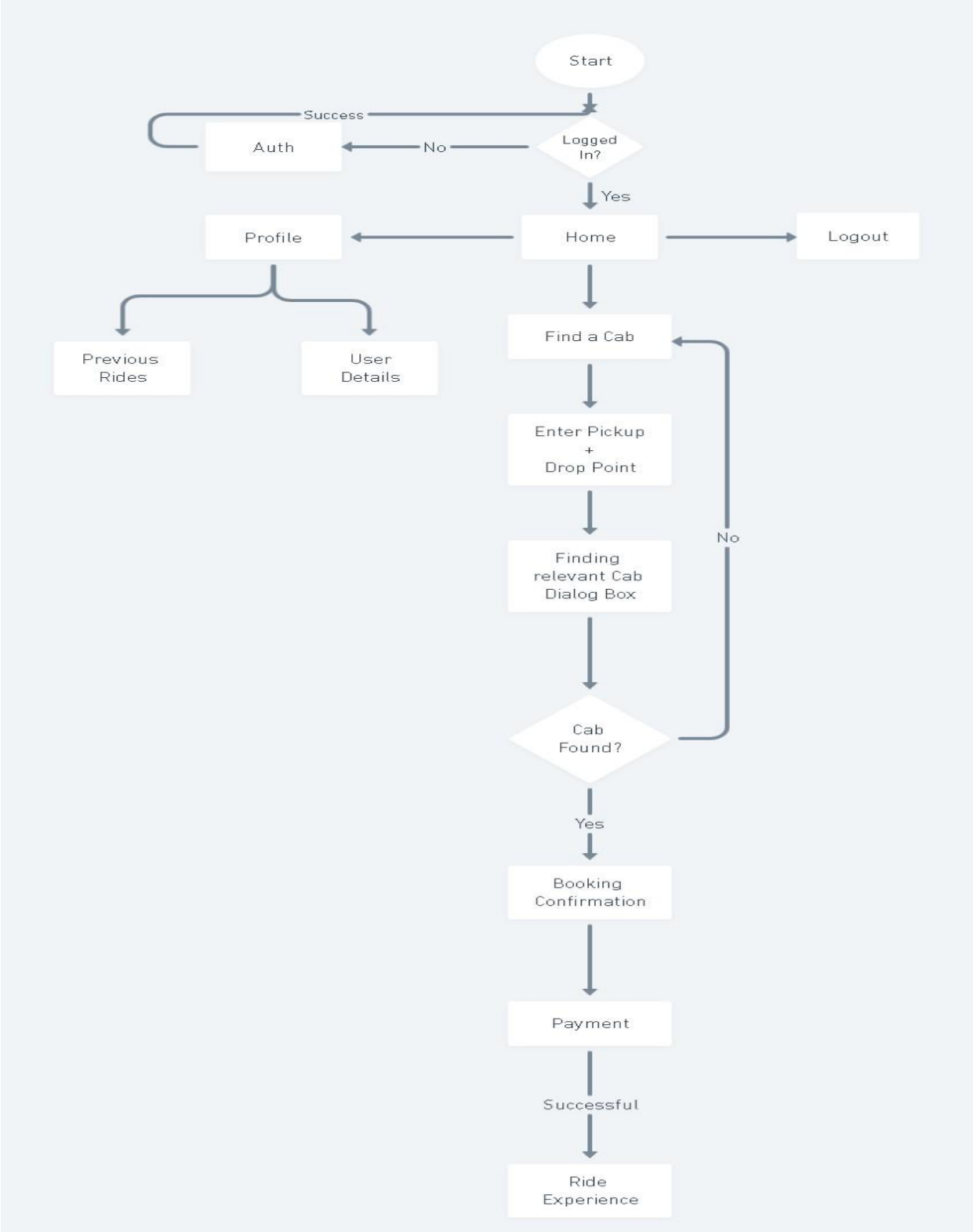
Administrators:

- Cab Allocation Optimization: Administrators can fine-tune the cab allocation algorithm, ensuring strategic allocation to minimize overall travel distances.
- User and Driver Management: Administrators oversee user and driver profiles, handling tasks such as account creation or removal to maintain a secure user base.

- Fault-Tolerant Mechanisms: Equipped with tools to implement fault-tolerant mechanisms, backup and recovery strategies, and error recovery procedures, administrators contribute to system robustness and data integrity.

These features collaboratively guarantee a smooth and efficient experience for users, drivers, and administrators within the SmartRide Allocation System.

Project Flow Chart



Authentication

The libraries and mechanisms outlined in the flowchart encompass:

Bcrypt: Enhanced Password Security through Robust Hashing

- Bcrypt serves as a robust library for secure password hashing, incorporating salting to reinforce the hashing process. Salting ensures the uniqueness of hashed representations, even for identical passwords, significantly bolstering password security.
- Utilizing Bcrypt elevates the authentication process to a higher level of security, mitigating vulnerabilities associated with common password usage and potential brute-force attacks. The salting mechanism adds an extra layer of protection, preventing attackers from exploiting precomputed tables or rainbow tables.

authToken/JWT: Cryptographically Signed User Identity Representation

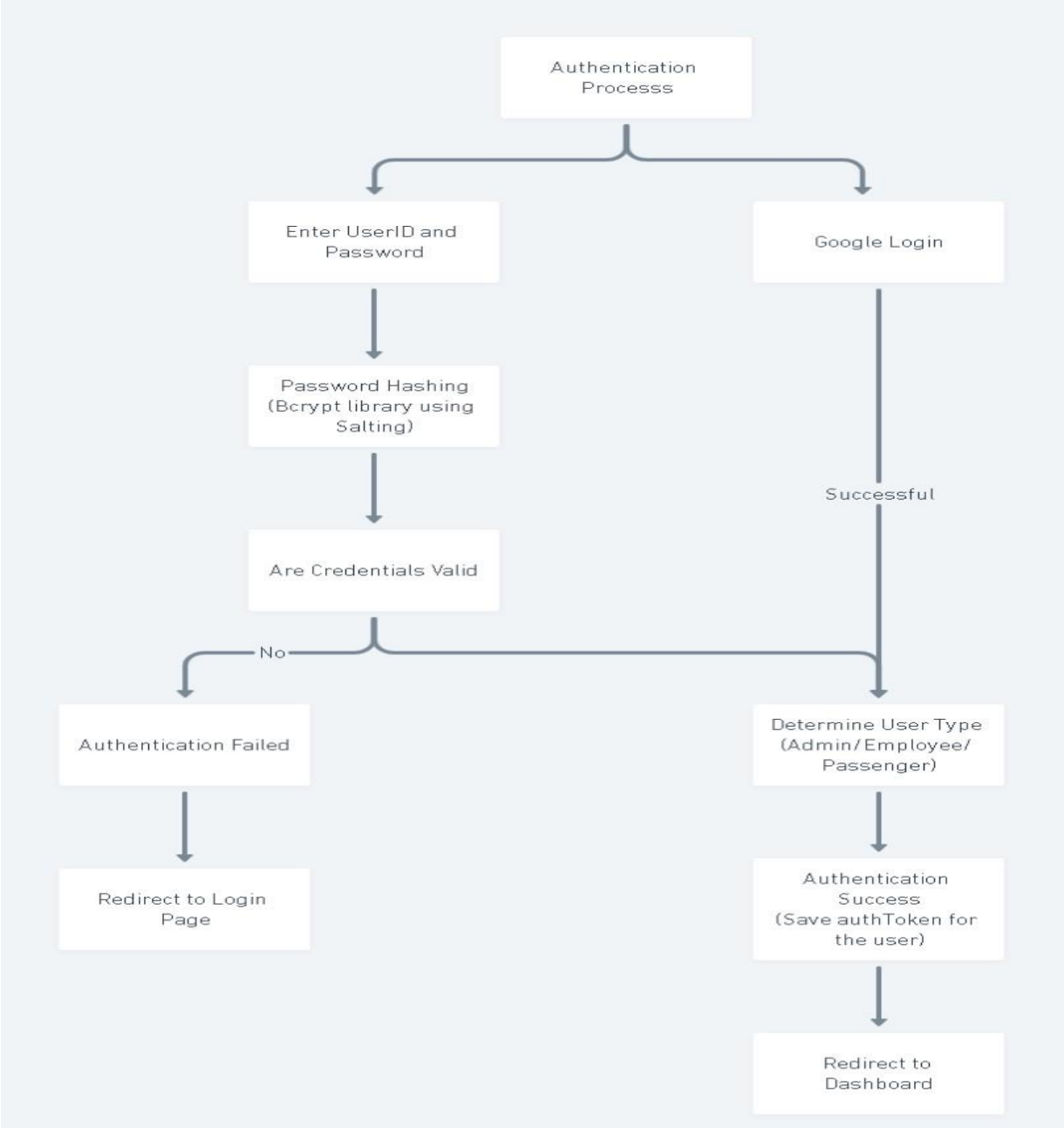
- authToken, implemented as a JSON Web Token (JWT), functions as a secure and cryptographically signed representation of the user's identity. This token includes claims conveying essential user details and permissions, facilitating secure session management and subsequent user authentication for requests.
- JWTs provide a standardized and secure method for representing user identities in the authentication process. By digitally signing the token, the integrity of the user's identity claims is assured, preventing tampering or unauthorized modifications. This mechanism enhances the overall security and reliability of user sessions.

OAuth 2.0 (Google Login): Secure User Authentication via Google Services

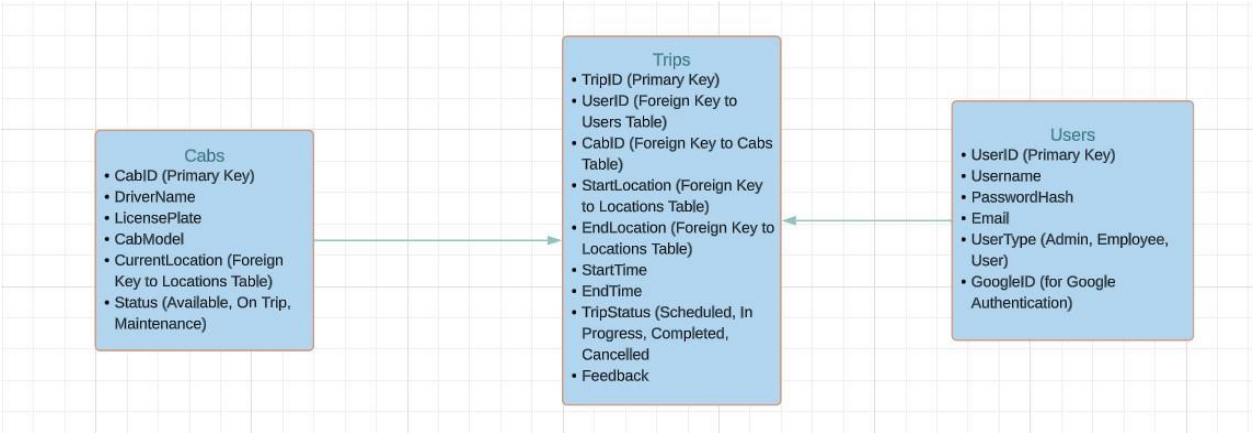
- OAuth 2.0 acts as an authorization framework for secure user authentication through Google's services. This mechanism grants the application limited access to user accounts on its HTTP service by delegating the authentication process to Google's hosting service.
- Leveraging OAuth 2.0 for Google Login ensures a secure and standardized approach to user authentication. By relying on Google's authentication service, the application minimizes the handling of sensitive user credentials, enhancing overall security. This framework also simplifies user access, allowing users to utilize their existing Google credentials for authentication.

In summary, the integration of Bcrypt, authToken/JWT, and OAuth 2.0 plays a crucial role in establishing a secure, efficient, and standardized authentication process. These libraries and mechanisms collectively contribute to the robustness of user identity management, password security, and the overall integrity of the authentication flow within the application.

Authentication Flow Chart



Database



Schema

Utilize REDIS to store rows <CabID, location{x,y}> for efficient handling of high read and write throughput for real-time location data. Each cab is programmed to transmit its location every 10 seconds, updating the database.

REDIS proves advantageous in this scenario due to its Geospatial features and capability to manage data partitioning based on location. To determine the specific partition corresponding to a {lat, long}, employ data structures like quadtrees for streamlined location-based organization.

Code for making tables in MongoDB:

```
const tripSchema = new mongoose.Schema({
  tripId: { type: String, unique: true },
  userId: { type: mongoose.Schema.Types.ObjectId, ref: 'User' },
  cabId: { type: mongoose.Schema.Types.ObjectId, ref: 'Cab' },
  startLocation: { type: mongoose.Schema.Types.ObjectId, ref: 'Loca
  endLocation: { type: mongoose.Schema.Types.ObjectId, ref: 'Locati
  startTime: Date,
  endTime: Date,
  tripStatus: String // Scheduled, In Progress, Completed, Cancelle
});

const User = mongoose.model('User', userSchema);
const Cab = mongoose.model('Cab', cabSchema);
const Trip = mongoose.model('Trip', tripSchema);
```

```
import mongoose from 'mongoose';

const userSchema = new mongoose.Schema({
  userId: { type: String, unique: true },
  username: String,
  passwordHash: String,
  email: String,
  userType: String, // Admin, Employee, User
  googleId: String
});

const cabSchema = new mongoose.Schema({
  cabId: { type: String, unique: true },
  driverName: String,
  licensePlate: String,
  cabModel: String,
  currentLocation: { type: mongoose.Schema.Types.ObjectId, ref: 'Lo
  status: String // Available, On Trip, Maintenance
});
```

Real-time location data integration

Real-time location data integration is a crucial component of any mode

```
mongoose.connect('mongodb://localhost:27017/smartCabDB', {
useNewUrlParser: true, useUnifiedTopology: true });
```

rn ride-sharing platform. It allows the system to track cabs continuously, ensuring that the information used for making decisions about cab allocation is as accurate and timely as possible.

Cab-Side Implementation:

- Each cab is equipped with a GPS device or a smartphone app capable of determining its current location.
- The device/app is configured to push location data to the server at predefined intervals, typically every 10 seconds. This interval can dynamically adjust based on the cab's state, such as becoming more frequent during an ongoing trip.
- The cab's device/app undergoes authentication with the server using secure protocols, establishing a secure session for data subscription.

Server-Side Implementation:

- The server maintains a persistent connection or listens for incoming location updates from each cab.
- Source authentication is employed to prevent spoofing or unauthorized access.
- Upon receiving an update, the server promptly processes it or places it in a queue for batch processing, depending on the system's real-time data handling design.

Data Handling:

- The server utilizes a geospatial index, like a Quad-tree or a geospatial-capable database such as PostGIS, for updating the cab's location efficiently.
- A dynamic list/map ensures a current view of each cab's status and other relevant details, facilitating quick access and updates.

Location Update Algorithm:

- When a location update is received, the algorithm checks for significant differences from the last known location to avoid unnecessary updates.
- If the location has changed beyond a certain threshold, the cab's entry in the geospatial index is updated.
- The dynamic list/map is also updated to reflect the new location and potentially the cab's availability status.

Fault Tolerance and Reliability:

- The server gracefully handles missed updates, setting a cab's status to "unknown" if it fails to send updates within a specified period.
- Redundancy is integrated into the communication system to address primary update channel failures.

Scalability:

- The server's infrastructure horizontally scales to accommodate additional cabs without significant performance degradation.
- Load balancers distribute incoming data across multiple servers.
- Sharding or replication of the geospatial index and dynamic list/map across servers efficiently handles read/write loads.

Security Considerations:

- TLS/SSL encrypts all communication between cabs and the server to safeguard the privacy and integrity of location data.
- Access to the location update API is restricted to authenticated devices/apps, preventing unauthorized data injection.

Optimizations:

- Predictive algorithms may be implemented, leveraging historical data to anticipate a cab's future location, reducing the reliance on constant real-time updates.
- Data aggregation techniques summarize location information when precise details are unnecessary, reducing stored and processed data volume.

System Monitoring and Alerts:

- Monitoring tools track location data flow, triggering alerts if data streams fall below certain thresholds, indicating potential issues with cab-side devices or network connectivity.
- System metrics, including update latencies, processing times, and queue lengths, are monitored to ensure the real-time system remains performant and responsive.

Through this comprehensive approach, the platform ensures up-to-date cab location data, facilitating efficient cab allocation and an optimal user experience for both drivers and passengers.

.

Employee's Cab Search Optimization:

The aim of the Employee's Cab Search Optimization is to design a system that enhances the user experience for employees seeking cabs. This system focuses on providing employees with information on cabs that are currently in use but are near their location and likely to become available shortly. Here is an explanation of how this might work:

- **Real-Time Cab Status Tracking:** The system maintains a real-time status of all cabs, indicating whether they are currently engaged in a trip, available, or expected to be available shortly based on the estimated time of trip completion.
- **Proximity-Based Cab Suggestions:** When an employee initiates a cab search, the system quickly performs a geospatial query to identify cabs on a trip but in the vicinity of the employee's location. This involves calculating the distance between the employee's location and the current locations of cabs from the real-time geospatial index.
- **Estimation of Cab Availability:** The system estimates when a cab will become available by using the current trip's destination and estimated time of arrival (ETA). If the ETA falls within a specified time frame, such as the next 5-10 minutes, the cab is flagged as a potential match for the employee.

Algorithm:

1. Upon receiving a ride request from a user, the client application communicates with the server to initiate the cab allocation process. The server processes the user's location, identifying available cabs within a 100-meter radius. These cabs are then displayed to the user for selection, and simultaneously, the server communicates the ride request to the drivers of the selected cabs.
2. If a driver accepts the request, the cab is allocated to the user, and the ride details are finalized. However, if there is no acceptance from any driver within one minute of the user's request, the algorithm enters a state of incremental search radius expansion. Specifically, the search radius is doubled, and the search for available cabs is retried.
3. The server continues to double the search radius every minute until a driver accepts the request or until the maximum wait threshold of five minutes is reached. If this threshold is crossed without any driver acceptance, the algorithm concludes that no cabs are available within a reasonable distance. Consequently, the user is notified of the unavailability of cabs, ensuring transparent and timely communication.

Basic Code in py

```
Begin Dynamic Cab Alloca initial_radius = 100  # Setting an initial radius value
time_interval = 1
current_radius = initial_radius
timer = 0
max_wait_time = 5

def identify_available_cabs(current_radius):
    pass

def display_cabs_to_user(available_cabs):

    pass

def send_request_to_drivers(available_cabs):
    pass

def driver_accepts_request():

    pass

def allocate_cab_to_user():

    pass

def reset_time_interval():

    pass

def notify_user_cab_unavailability():

    pass

# setting time interval for radius expansion to 1 minute
time_interval = 1
current_radius = initial_radius
timer = 0

while timer < max_wait_time:
    available_cabs = identify_available_cabs(current_radius)
    display_cabs_to_user(available_cabs)
    send_request_to_drivers(available_cabs)

    if driver_accepts_request():
        allocate_cab_to_user()
        break
```

```
    if timer % time_interval == 0:
        current_radius *= 2
        reset_time_interval()

    timer += 1

if timer == max_wait_time:
    notify_user_cab_unavailability()
```

```
tion

    Set initial search radius to 100 meters

    Set maximum wait time to 5 minutes
```

Evaluating the System's Effectiveness:

- **Response Time Measurement:** The system's effectiveness is quantified by assessing the speed at which it delivers suggestions. The duration from the employee's search request to the presentation of nearby cabs is recorded and analyzed.
- **Relevance of Suggestions:** Relevance is determined by evaluating how frequently employees choose the suggested cabs and the feedback provided post-ride. If suggestions are consistently ignored or receive negative feedback, adjustments to the relevance algorithm may be necessary.
- **Feedback Loop Integration:** Employees have the option to provide feedback on suggested cabs, enabling the system to refine its suggestion algorithm. For instance, if employees prefer cabs available sooner, even if slightly farther away, the system can adapt its suggestions based on this feedback.
- **Algorithm Tuning:** Regular reviews of performance data and feedback inform the fine-tuning of proximity thresholds and the timeframe for when cabs are expected to become available.
- **User Experience Metrics:** Metrics such as the number of successful matches, average waiting time after selection, and user satisfaction scores offer insights into the system's overall performance.

By concentrating on these evaluation criteria, the platform strives to minimize waiting times for employees and enhance the utilization rate of cabs. The key to success lies in adopting a dynamic, real-time data-driven approach that continuously adapts to observed patterns in employee usage and preferences.

Employing Caching:

Integrating caching into the SmartRide Allocation System can significantly enhance its performance by reducing database load and improving response times. Here's how caching can be effectively utilized:

- **Caching Real-Time Cab Locations:** Implement an in-memory data store such as Redis to cache the real-time locations of cabs. This approach facilitates quick access to current cab locations without the need for repetitive database queries, a crucial aspect for real-time systems.
- **Caching User Session Data:** To optimize user experience, especially in terms of authentication speed, caching user session tokens or credentials after the initial login can be beneficial. This minimizes the necessity for frequent database hits during subsequent requests for authentication checks.
- **Caching Trip Histories and Analytics:** For swift access to historical data, caching trip histories and related analytics proves advantageous. This is particularly useful for generating reports or analytics without relying on extensive database queries.
- **Cache Invalidation and Update Strategies:** Implement strategies for real-time cache updates when data changes and set appropriate Time-to-Live (TTL) for each cache entry. Ensuring proper cache invalidation is essential to maintain data consistency.
- **Choosing the Right Caching Strategy:** Depending on the nature of the data, employ different caching strategies such as write-through cache (for critical data) or lazy loading (for less critical data).
- **Scalability and High Availability of Cache:** The caching solution should be scalable and capable of handling high loads. A distributed caching system is recommended for high availability, preventing a single point of failure.

By strategically implementing caching in these areas, the SmartRide Allocation System can achieve faster data retrieval, reduce the load on the database, and provide an overall more efficient and responsive service.