RideShare

Seo Pallichirayil, Rahul Gaonkar, Rishi Ranjan, and Anmol Singh

***Abstract*—Currently, all the urban cities face a major problem of transportation and air pollution. In the following paper, we have tried to reduce this problem by creating a new system called RideShare which is a simple carpooling service that both Drivers and Riders can use to Transit from one place to the other.**

I. Introduction

RideShare is a simple online application which has been made available on a website as well as an Android app, where users can log in as a Driver or Rider. For both Driver and Rider, their source location from where they want to start the journey is picked by the GPS and they enter their destination till where they want to go and the RideShare App will find all the currently nearby users.For Drivers, App shows the nearby available Riders and vice versa.Therefore, two people can share the same car and reduce their cost of transportation and it also has environmental advantages.

II. Implementation or Working

The user can get into the system through a Sign in or Sign Up page which provides email authentication. After that User can select what role would they like to choose that day Driver or Rider.

After this, we're using Google Maps to pinpoint the source and destination of both the Rider and Driver on the map. When a user as a Driver or Rider clicks on the Search Ride button on the App, all the possible nearby users (Rider or Driver respectively) are shown on the map, the nearby users are depicted by grey icons on the map. When user as a Driver or Rider clicks on the nearby users, he can see an option to send a request to that user.

Once he has sent a request, that request is logged on the database and there is a database polling service that the app is using continuously after every 10 seconds, to see if there are any requests that are coming in for that particular user and as soon as it sees the request the icon depicting their nearby user changes and the grey icon turns into a yellow icon so that the other user knows that a request has come from that particular user.

After the user received a request from a particular user, he can click on it and he can see two more options that is accept or decline, he can either choose to decline that request or accept the request. If the user declines the request then the icon returns back to grey color and will no longer be yellow.

If the user chooses to accept the request, that request is stored as a transaction in the SQL database, so that at the same time no two users can accept the same request for Driver or Rider in this way we ensure that only one-to-one mapping can be done between a Driver and Rider and there are no two people claiming the same seat in a ride

The App continuously uses a database polling service after every 10 seconds to check if there is any ride entry with approved status (nearby user has accepted the ride request) for that particular user and if it finds one, it redirects both the users sharing the ride to the Start/View ride page. This is the page that the Driver uses to pick up the approved nearby user. On this page we are using the Haversine formula, but instead of using a distance API call from Lambda we are using this formula in the client side, so that we can increase the throughput , availability and scalability of the system.

As soon as the Driver reaches within 0.05 miles of the Rider, the Rider icon disappears on the screen so it indicates that the Driver has picked up the Rider and the journey starts to the destination location of the Rider.

We consider that the journey is completed when the Driver drops off the Rider to his destination location and as soon as the Rider reaches his destination point we pop up a rating page on both the Driver and then the Rider.

III. Architecture

The architecture is pretty simple and straightforward in which we are using a few Lambda functions, API gateways, CloudWatch triggers, MySQL (AWS RDS) and AWS Cognito

The main architecture revolves around how the system pairs Drivers and Riders. We sort out all the Drivers in a 1.5 mile radius of the Riders and vice versa. We have done this using the Haversine formula which is a globally accepted algorithm to calculate distances.

But this formula alone is not enough, we still need to calculate if all the people are going to the same place or if they have different destinations. we can't find exactly two people going to the same destination but instead of that, what we have actually done is we have found out the distance of the original journey and then we have seen if the journey that deflects would take more than 5 km. In this, our system ensures that the Driver doesn't have to go out of his way to drop the Rider.

The basic flow with significance of each Lambda functions:

PostRouteDetails: As soon as the user enters the details of the journey, this lambda function makes an entry into the database with a status of searching.

SearchNearestRideRequestDetails: This piece of code is responsible for finding all the nearest eligible Riders /Drivers near the logged in User.

RideRequest: Any time a user sends a request, accepts it or declines it, This lambda function gets called changing the status of database entry to Pending Approval, Approved, Declined respectively.

Seo Pallichirayil (e-mail: sgp322@nyu.edu)
Rahul Gaonkar (e-mail: rpg283@nyu.edu)
Rishi Ranjan (e-mail: rr3261@nyu.edu)
Anmol Singh (e-mail: as11294@nyu.edu)

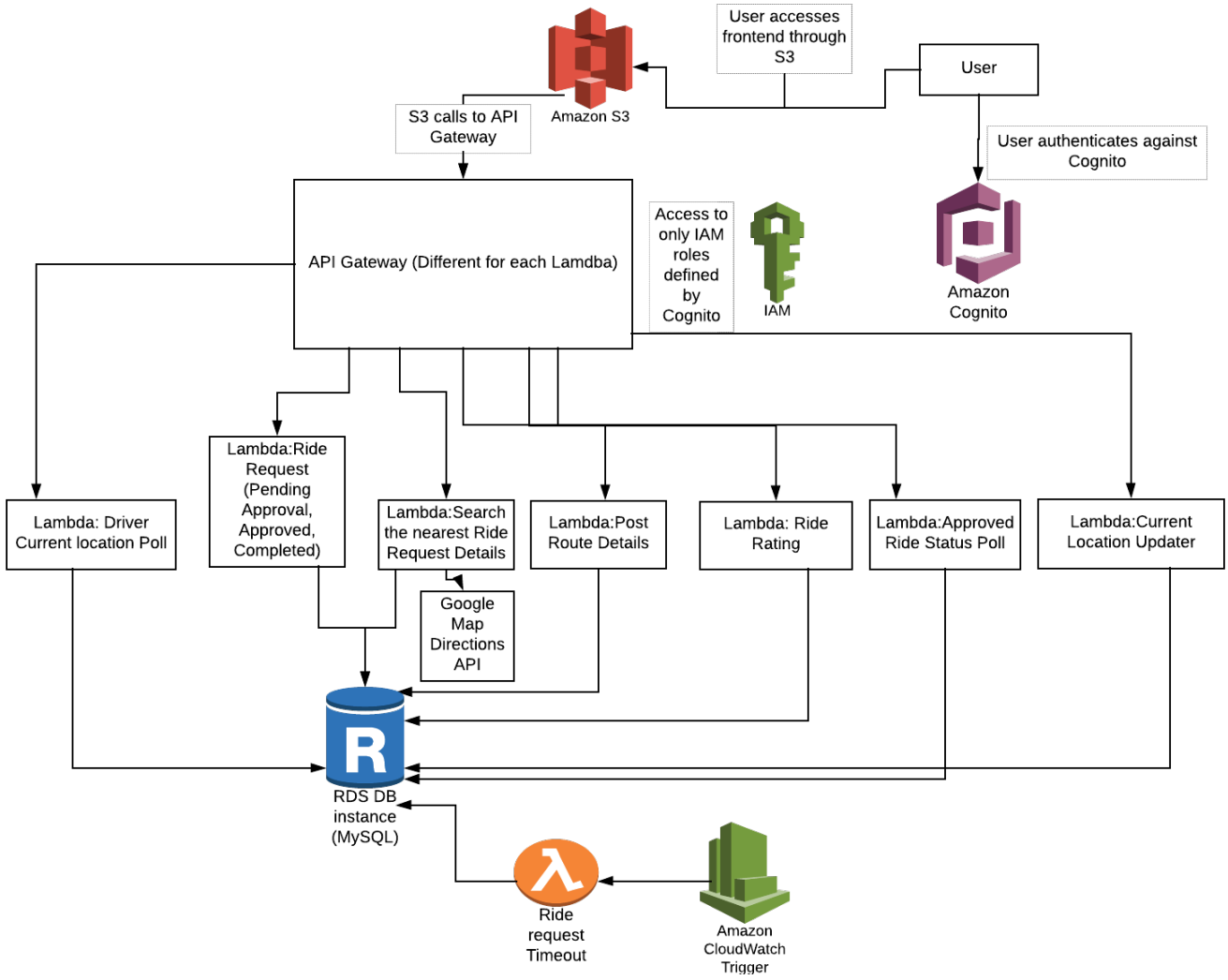

Fig. 1.

ApprovedRideStatusPoll: This is a polling function that helps the client to know if any of the users have matched and approved each other , so that the App can redirect them to the ride page.

RideRequestTimeout: Inorder to make sure that user doesn't just make an entry into the database as searching and then just doesn't act on it, We make periodic deletion of stale entries. This is done by Cloudwatch Trigger, that triggers this lambda function every thirty minutes and deletes the stale entries.

CurrentLocationUpdator: This lambda function is used to update the current location of Driver/Rider when the journey starts.

RideRating: This lambda function is used to log the rating record of Driver/Rider once the journey is completed.

DriverCurrentLocationPoll: This lambda function is used for polling current location of Driver on the Rider's page.

Hosting the App:

The app is hosted on S3 public bucket with enabled web hosting.

The reasons to choose database technology:

For database, we have chosen an instance of AWS RDS MySQL. The reason to choose AWS RDS MySQL is that we only have data with a fixed schema and have to perform some transactions.

IV. Security

AWS Cognito:

Login and Signup are maintained by AWS Cognito. So the user passwords are securely stored. All the API calls require IAM authorization.

Databases:

The root account of the database is not used anywhere in the code, Separate users have been created for the lambda functions to access, the separate users have been granted only the necessary permissions.

Code:

None of the passwords are stored or written in the code, all the passwords are stored in the system or environment variables, so even making the code public won't cause any attacks.

Scalability:

We are using a serverless microservice architecture for RideShare Application with AWS Lambda and API Gateway.

Offline Calculation of Distance:

We are using Haversine formula to calculate distance in SQL procedures thereby preventing too many calls to Google API which will reduce the overall costing of the system.

V. Conclusion

We have build an Android App as well as web application and made it scalable for multiple users to enjoy the service.

VI. Future Work

mplement Kinesis to improve the scalability of the system, so the requests can be accepted as streaming requests to scale the system to millions of users.

Right now we are only focusing on one-to-one mapping that is one Driver can have only one Rider. But in the future, we would like to expand that and make Driver put the number of seats available in this car so that he can take multiple people to different locations on the same ride.

The system right now sorts the nearby users based on the difference of the distance that the Driver needs to go. We would like to further work on a system that would take real-time traffic into consideration as well.