# PedalHire

Manikanta Vankayala (1218816326)
Rahul Santhosh Kumar Varma (1218475349)
Vivek Kumar Maskara (1218403147)

## 1. Introduction

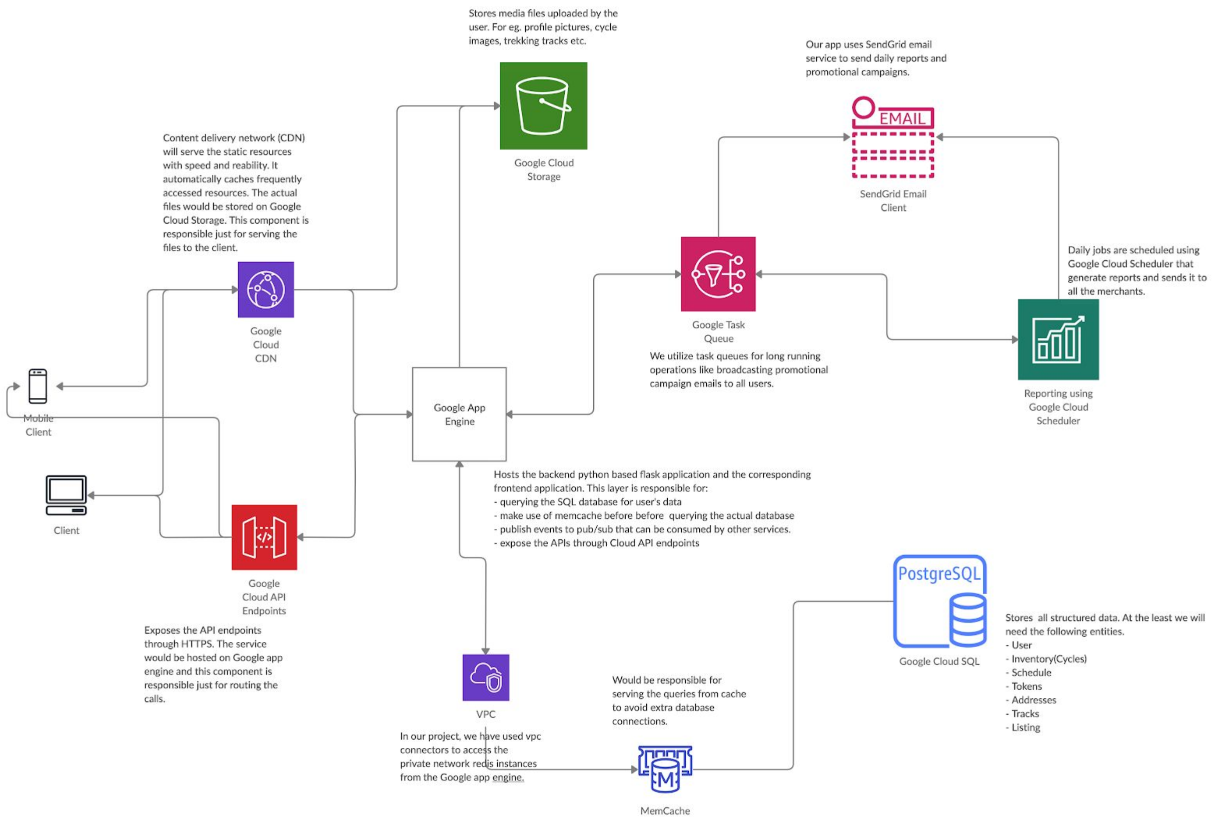
PedalHire is a peer-to-peer bike rental application. It allows us to rent and hire a bike in real-time using a site hosted on the Google Cloud platform. This application supports two types of users, Merchants, the person who wants to rent the bike and Users, the person who wants to hire a bike. A merchant can list his bike(s) for rent and provide a schedule during which it is available for rent. A user can search for available bikes using the location and date parameters when he requires it and it will show all the listings after which the user can proceed to book it for the required slot.

## 2. Background

A lot of students buy bikes for their personal use but only a few of them use it regularly as noticed at ASU. This gave us the idea of creating an app which connects people who can put their bikes for rent when they are not using it and other customers who just want to use it for a short time. This helps in saving expenses for customers who don't have to spend a huge amount and time to purchase a bike and can rent one for a short time. This is very useful for students who can earn some money on the side. With the rise of biking as a recreational and adventurous activity, people who visit from other places can quickly rent bikes and use it without any hassles. This application was developed using Python/Flask with the UI implemented using Jinja templates. This was hosted on Google Cloud (https://cse546-group17.uc.r.appspot.com/) and utilized a lot of features of the cloud which helped in optimizing the application such as Memcache, Task Queues, etc.
At present there are only bike rental applications by companies and there are no peer-to-peer rental agencies. Once this application is successfully launched, we would be the first in the market to connect different individuals to facilitate a P2P model of bike rentals.

# 3. Design and Implementation

## 3.1. Architecture


Stores media files uploaded by the user. For eg. profile pictures, cycle images, trekking tracks etc.
Google Cloud Storage
Our app uses SendGrid email service to send daily reports and promotional campaigns.
EMAIL
SendGrid Email Client
Content delivery network (CDN) will serve the static resources with speed and reability. It automatically caches frequently accessed resources. The actual files would be stored on Google Cloud Storage. This component is responsible just for serving the files to the client.
Google Cloud CDN
Google Task Queue
We utilize task queues for long running operations like broadcasting promotional campaign emails to all users.
Daily jobs are scheduled using Google Cloud Scheduler that generate reports and sends it to all the merchants.
Reporting using Google Cloud Scheduler
Mobile Client
Google App Engine
Client
Hosts the backend python based flask application and the corresponding frontend application. This layer is responsible for:
- querying the SQL database for user's data
- make use of memcache before before querying the actual database
- publish events to pub/sub that can be consumed by other services.
- expose the APIs through Cloud API endpoints
Google Cloud API Endpoints
Exposes the API endpoints through HTTPS. The service would be hosted on Google app engine and this component is responsible just for routing the calls.
PostgreSQL
Google Cloud SQL
Stores all structured data. At the least we will need the following entities.
- User
- Inventory(Cycles)
- Schedule
- Tokens
- Addresses
- Tracks
- Listing
VPC
In our project, we have used vpc connectors to access the private network redis instances from the Google app engine.
Would be responsible for serving the queries from cache to avoid extra database connections.
MemCache

### 3.1.1 Google Cloud Services Used:

1) **Google App Engine:** GAE hosts the complete Flask application. This layer is responsible for performing the following tasks.
   - Querying the SQL database for the users data and product data.
   - Accessing the media files from Google Cloud Storage.
   - Authenticating the user while accessing restricted APIs and pages.
   - Using Memcache to decrease dependence on the database and enable faster retrieval of constantly accessed data

- Publish events to task queues to enable asynchronous execution of tasks.
- Exposing the Flask applications API through Google endpoints
- Updating the memory store data when the corresponding data in cloud SQL gets updated

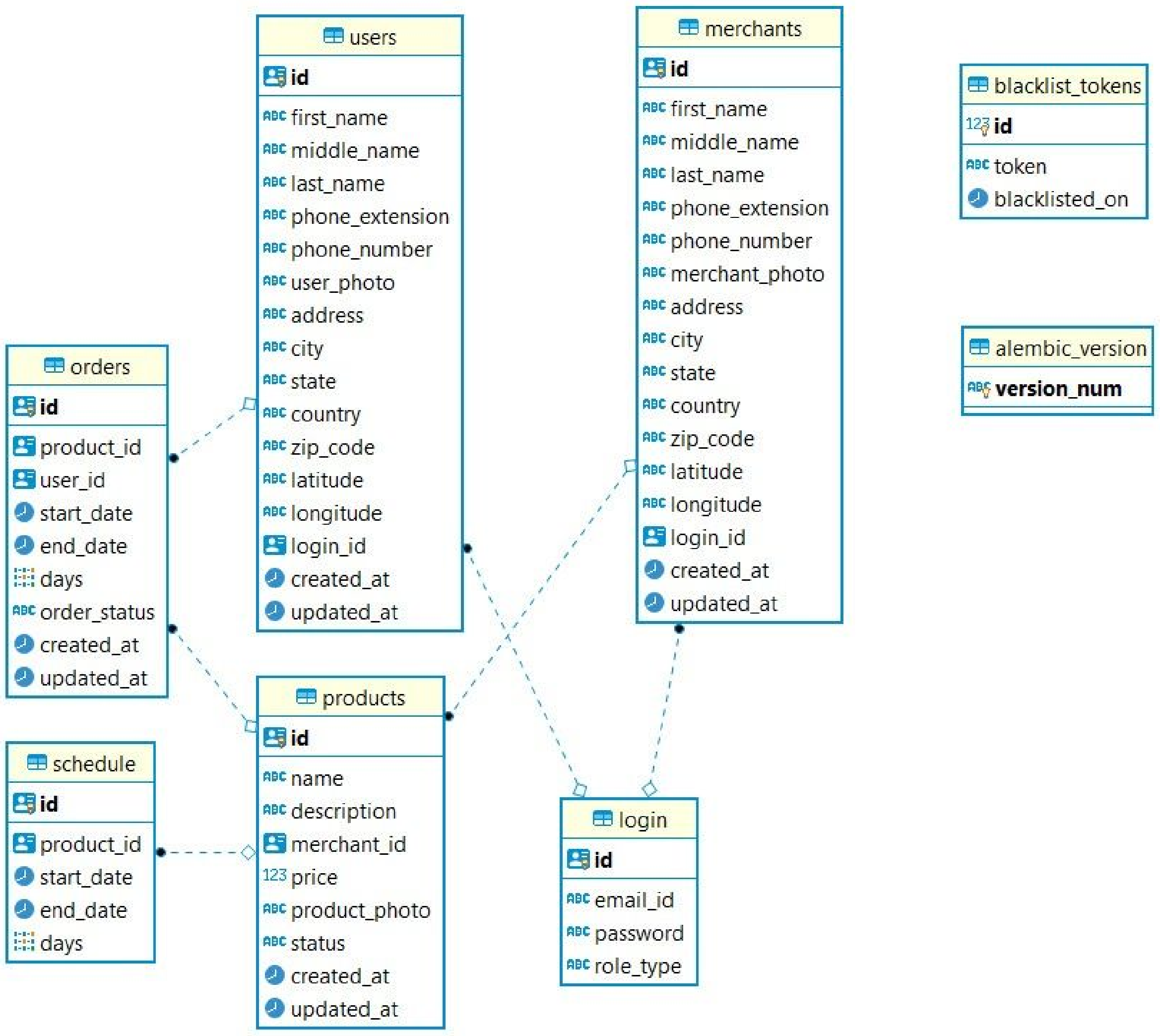
2) **Google CDN:** Google Content Delivery Network (CDN) will serve the static resources with speed and reliability. It automatically caches frequently accessed resources. The actual files would be stored on Google Cloud Storage. This component is responsible just for serving the files to the client. It has high availability, low network latency, and improves the user experience. It also helps in segmenting the audience.

3) **Google Endpoint:** Google Endpoint exposes the API endpoints through HTTPS. The service is located in the Google App Engine and this component is responsible for routing the calls.

4) **Google Cloud Storage:** Google Cloud Storage is used to store and access the data from the Google Cloud Platform (GCP). Google Cloud storage is more reliable and scalable with advanced sharing and security capabilities. In this project, we used Cloud Storage to store the media files uploaded such as the bike images.

5) **Google Cloud SQL**: Google Cloud SQL  is the relational database to store the structured data on the cloud. In this project, we decided to use a relational form to store the data and created tables such as Users, Merchants, Products, Login, and Orders. Google Cloud SQL provides high availability, automated backups, and better performance by integrating with Google App.  The below diagram represents the way we stored the data in the database.

    - Users: This table stores the users information. A new record will be added to this table if any new user registers on the site.
    - Merchants: This table stores the merchants information. A new record will be added to this table if any new merchant registers on the site.
    - Login: This table stores login credentials of the user and merchant i.e. the email and password. A new record will be added to this table if any new user or merchant registers on the site.
    - Products: This table stores the product details. A new record will be added when the merchant logs in and adds a product on the site.
    - Schedule: This table stores the product's schedules. A new record will be added when the merchant logs in and adds a product on the site.
    - Orders: This table stores the order details. A new record will be added when the user logs in and orders a product.



6) **Google Cloud MemoryStore:** Google Cloud Memory Store is responsible for serving the queries from the cache to avoid additional database calls to retrieve the data. It improves the performance of the application and reduces the load on the cloud SQL. It stores the data in a key-value format. Here, the key is the primary key value of the row and the value is the entire row of the particular table.

7) **Google Cloud Task Queues:** Task queues allow the applications to perform asynchronous work. They can also act as a messaging queue. In our project, we used it for performing the asynchronous task of broadcasting promotional campaign emails to all the users.

8) **VPC Connectors:** A Google VPC connector acts as a bridge between different networks. In our project, we have used it to access the private network Redis instances from the Google App Engine.

9) **Auto Scaling**: As we are using the Google App Engine (GAE), there is no need for explicitly implementing the auto-scaling as GAE takes care of auto-scaling. We just have to select the type of scaling that has to be used. In this project we used the F1 type autoscaling. In F1 type autoscaling, one instance is always running in the cloud while additional instances are provided as per the demand.

### 3.1.2. The flow of Execution:

Our proposed solution solves the problem of renting and hiring a bike in the real time in a very simple way. The below explanation helps to understand the way we solved the problem.
Application mainly works for two types of users -

1) Merchant - User who lists the bikes for rent
2) User - User who want to rent the bikes for rent

**1) The flow of execution for Merchant**

- If the merchant is already registered then he can directly sign in, If not, he has to register with his details first after which he will be logged in. Internally, the password that he provides during registration will be hashed and stored in the login table in Google Cloud SQL and the remaining details will get stored in the user table in cloud SQL and parallelly it gets stored in the cache.
- After sign in, he will get the option to view his profile. Merchant can recheck his details. Internally, these details will first be checked for their existence in the cache. If present, they will be retrieved from the Google Cloud Memory Store. If not, they will be retrieved from the user table in Google Cloud SQL.
- Once he confirms with his details, he can add the bike availability time slot details and the photo of the bike. This information will get stored in the product table and parallel we will also update it in the memory store.

**2) The flow of execution for User**

- If the user was already registered then he can directly sign in, If not, he has to register with his details first after which he will be logged in. Internally, the password that he provides during registration will be hashed and stored in the login table in Google Cloud SQL and the remaining details will get stored in the user table in Cloud SQL and parallelly it gets stored in the cache.
- After sign in, he will get the option to view his profile. The user can recheck his details. Internally, these details will first be checked for their existence in the cache. If present,

they will be retrieved from the Google Cloud Memory Store. If not, they will be retrieved from the user table in cloud Google Cloud SQL.

- The user can then search for the products in the *Product Search* page based on the date and location where he wants to rent the vehicle. Once these details are entered, a new page opens up which shows all the products satisfying these criteria.
- The user can book the bike he likes by clicking on the *Buy* button.
- The user can check the details of the order on the *Order Details* page.

Our proposed solution is better than the state-of-art in the following ways:

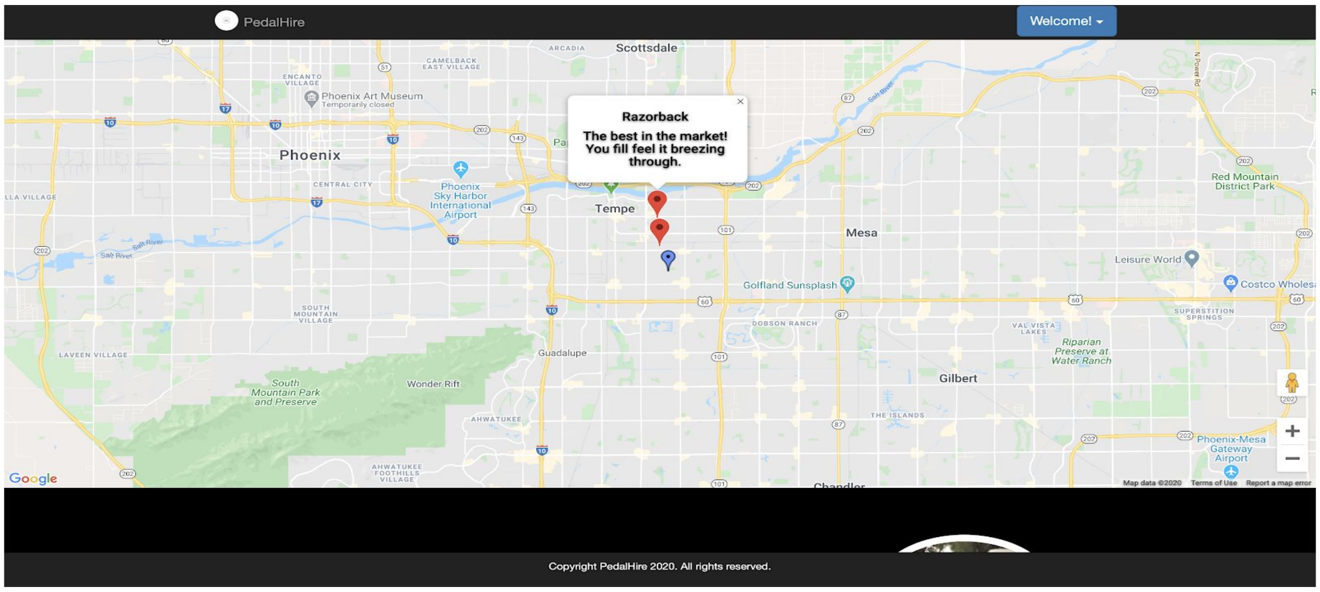
1) The services which we implemented in code are all loosely coupled which means no service is dependent on any other service.
2) Optimal utilization of the resources in the google cloud.
3) Implemented the more robust, reliable and scalable application.
4) Performance was improved using the google cloud memory store and google content delivery network.

## 4. Testing and Evaluation

### 4.1 Testing

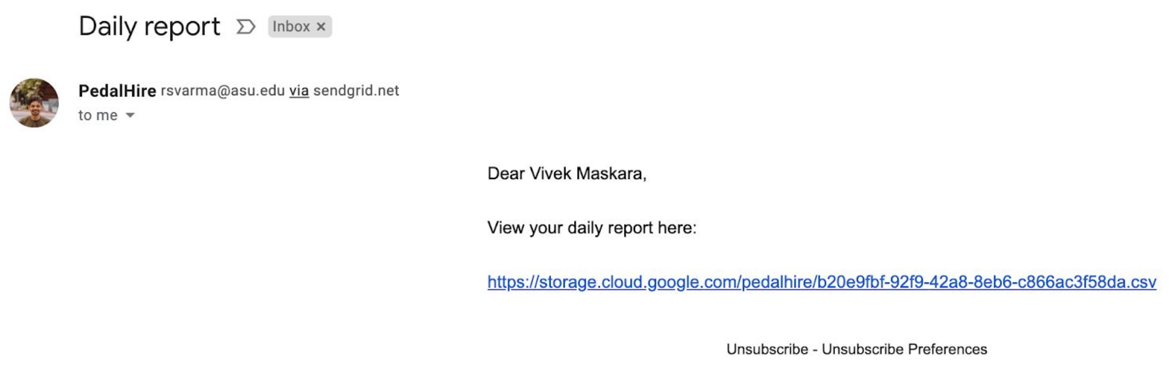
The testing phase can be broken down into 3 broad categories.
App URL: https://cse546-group17.uc.r.appspot.com/

**Core functionality**


PedalHire
Welcome!
Razorback
The best in the market! You fill feel it breezing through.
Scottsdale
Phoenix
Tempe
Mesa
Gilbert
Chandler
Guadalupe
Copyright PedalHire 2020. All rights reserved.

- Using the frontend application, we signed up a merchant in the Tempe area using the *merchantRegistration* page. We then listed a cycle by this merchant by adding the product details in the *addProduct* page.
- The merchant login page uses Google Places API to autofill the address, city and country.
- We repeated the first step for a few merchants with different locations and different products.
- Next, we registered a new user using the *userRegistration* page.
- All the APIs are exposed on the HTTPS protocol and the passwords are hashed using the argon2 algorithm.
- Whenever a user or merchant does a sign up or login, they are issued a JWT token that is valid for a period of 7 days. We tested this functionality by trying to call an authenticated API without the auth token.
- The JWT token is hashed using a SHA256 algorithm and the secret is stored securely in the app engine. All the API calls require proper authentication and some of the APIs use appropriate authorization.
- Once the user is logged in to the system with a valid auth token, he can navigate to the *productSearch* page and enter the start date, end date and location to search for cycles in a particular area.
- At this point the backend API is hit which returns a list of cycles in a 5 kilometer radius that are available for the days specified by the user. If a cycle is already booked by some other user, it is not returned by the API. We tested this functionality by registering multiple users in the system.
- The product search results are displayed on an embedded Google maps component with red marker pins showing the available cycles.
- Like every other API, the product search uses a redis cache(memcache) to avoid redundant calls to the DB. This helps us scale the application by optimizing the number of database connections required.
- The user can click and confirm to book any of these cycles. Once he confirms, the order is placed and it starts showing in the *viewOrders* page.

**Reporting:**



Daily report Inbox ×

**PedalHire** rsvarma@asu.edu via sendgrid.net
to me

Dear Vivek Maskara,

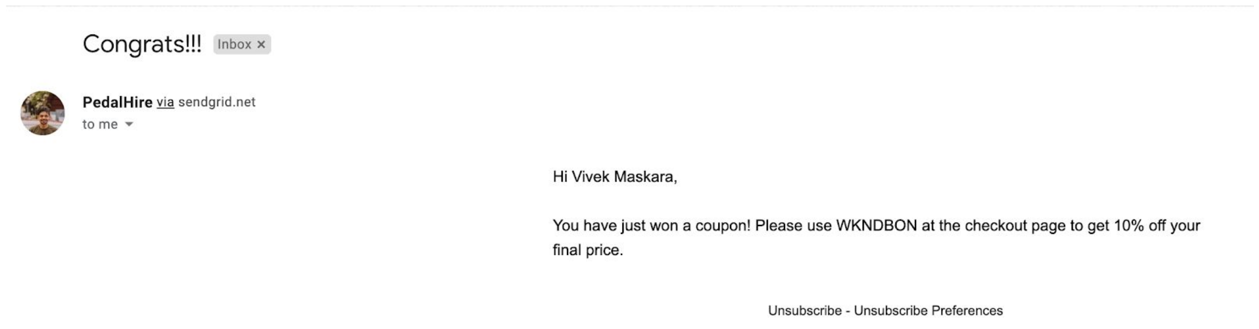
View your daily report here:

https://storage.cloud.google.com/pedalhire/b20e9fbf-92f9-42a8-8eb6-c866ac3f58da.csv

Unsubscribe - Unsubscribe Preferences

- Every morning an automated report is sent to all the merchants with details about the orders placed.
- We are using Cloud Scheduler to schedule these daily reports. At the specified time *generateMerchantReports* API is triggered by the scheduler.
- The API queries our PostgreSQL DB to fetch the data, generates the CSV file and uploads the file to Google Cloud Storage.
- A mail is triggered to the merchants with a link to the report. We are using SendGrid client APIs to trigger the emails.
- We were able to test the end to end functionality by triggering instant jobs using the Google Cloud Scheduler dashboard. Also, we verified that the same report was being generated at the specified schedule.

**Campaigns:**


Congrats!!! Inbox ×

PedalHire via sendgrid.net
to me

Hi Vivek Maskara,

You have just won a coupon! Please use WKNDBON at the checkout page to get 10% off your final price.

Unsubscribe - Unsubscribe Preferences

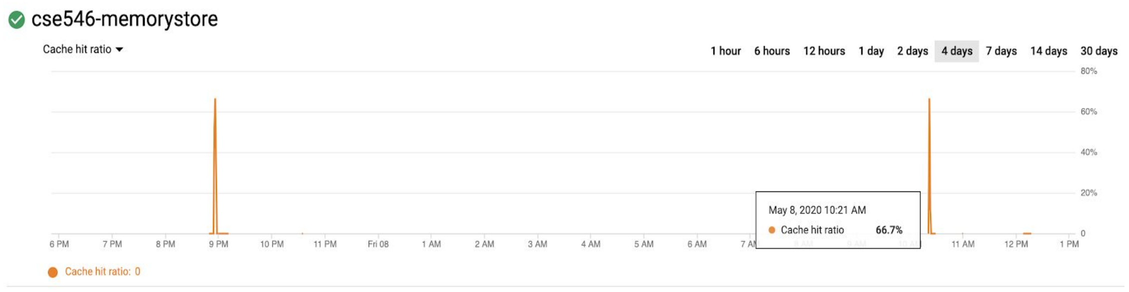
- The admin can broadcast emails to all users, informing them about any ongoing campaigns.
- Given the nature of this task we decided to use Google Task Queues so that the operation can execute asynchronously.
- We created a queue named *pedalhire-broadcast-queue* where we can add tasks using Python. The task calls the *broadcastEmails* API that is hosted on Google App engine.
- This task queries all the users from the PostgreSQL DB and sends them a mail using the SendGrid client.

## 4. 1 Evaluation

Our main criteria for evaluation was to ensure that all the core functionalities work smoothly, the application works with a very low latency, is reliable, scales well for a large number of users and uses the resources optimally.

While developing a PaaS based cloud application we were able to focus on the core functionalities of the application as the cloud provider takes care of the basic setup and provides numerous configurations to suit the application's needs. We were able to take advantage of these configurations and were also able to use quite a few different cloud components to build different parts of our application.

**Speed:**

- Our focus was to build an application with low latency and we wanted to make sure that the speed is maintained even when the application scales to a larger number of users.
- Our application uses a redis server for in-memory caching of the data that is stored in PostgreSQL. With caching in place, we were able to achieve a good hit rate, sometimes as high as 60-70%. We made sure that the cache never went stale so that the consistency of the application is not compromised even if the response is returned from cache.
- 

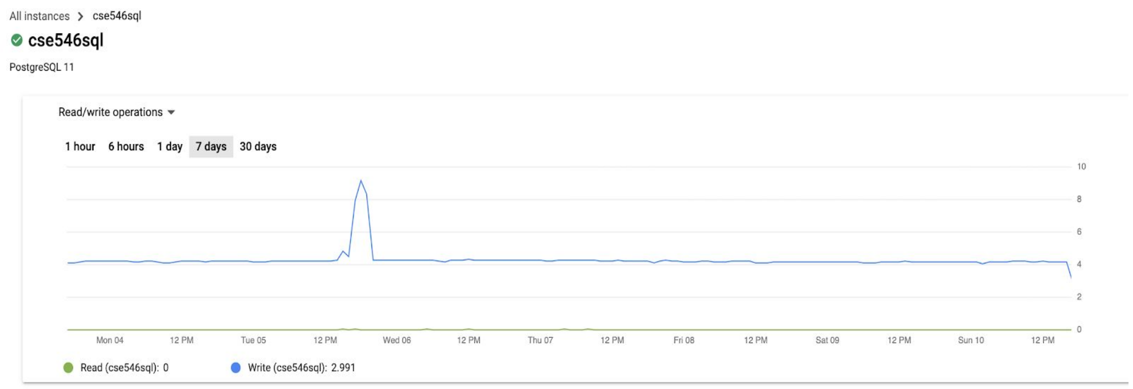
- We also utilized client side caching to avoid redundant calls to fetch things like auth token or user profile data.

**Optimal Utilization and Scalability:**

- We are using PaaS that takes care of auto scaling but it can be configured to suit the application's needs.
- We used the *instance_class* parameter to control the type of auto scaling we want. Also, we configured other automatic_scaling parameters.
    - The *instance_class* was set to F1
    - We set *target_cpu_utilization* to 0.65 so that a new instance is spawned when the CPU utilization goes above this threshold
    - We also set the *max_concurrent_requests* to 50 to specify that 1 instance can handle at most 50 concurrent requests.
- With our configurations the application should be able to scale up and down based on the load and would make the optimal use of resources.

**Reliability and Stability:**

- We take care of reliability by making the application modular and by introducing retrials for important tasks.
- We have imposed *retry_parameters* in the task queue to ensure that if the task to broadcast emails fails due to some reason it is retried again and again.
- Similarly, even though the app relies on redis cache, the application logic handles the unavailability of redis server gracefully. It falls back to fetching the data from the PostgreSQL database if the connection to redis fails.



- Most importantly, the Google App Engine(GAE) is stateless and the system doesn't rely on any in memory data that might get lost if the instance is shut down and started again. This happens quite frequently in case of PaaS. Our app caches the auth token in the browser's local storage so that it need not rely on the session variables set in the backend.

## 5. Code

### 5.1 Application Components

**Services:**

- **Memcache Service (memcache_service.py):** This service performs mainly five operations.
  1) Insert key value pair in google memory store : when we try to access or update any Row in the google cloud SQL and that row did not present in the memory store then By using this insert operation, we put the row in the memory store for the future use and the expiration time will be set to 10 mins.
  2) Get the value for a particular key: When we try to access the row in the cloud SQL. And if the corresponding value is present in the memory store. We get that row from Memory store using this get operation.
  3) Delete the value for the key: The row which got deleted in the google cloud SQL and the corresponding row that present in the google memorystore will be deleted using this update operation.
  4) Insert the list of values for a key: While we performing the product search we need to include many values for a particular key. Using insert list, we have included all the Values for that key.
  5) Get the list for a key: while performing the product search to get the list of products. We used this get list operation. We retrieve all the products from the memory store.

- **Login service (login_service.py)** - This service provides three main functionalities. It provides a registration functionality where it adds a user/merchant to the login table in case there is no corresponding entry with the same email present in the database. After adding it to the database, an authorization token is also generated which helps in making the API request on behalf of the user and prevents unauthorized page accesses or API calls. This authorization token is saved in the local storage of the system.
  It also provides the login functionality which validates the user/merchant credentials against the

username and the hashed password in the system and generates a corresponding auth token for the user/merchant.
It also provides a logout functionality where the authorization token is blacklisted i.e. deemed invalid such that the user /merchant cannot use the authorization token to access the website again and have to login again.

- **Merchant service (merchant_service.py) -** This service provides the functionality to add the merchant details to the merchant table. The details are retrieved from the merchant registration page on submitting the form.
  The merchant login functionality uses the login service and login in the merchant to add products for rent.
  Other functionalities such as getting all the merchants in the database, getting a merchant based on the ID and to retrieve merchant details for the email provided is implemented.
- **Product Service (product_service.py) -** This service provides the functionality to add the product details which retrieves the details from the *Add Product* page and saves it to the database. The product photo is saved to the Google Cloud storage and only the link to the image is saved to the database.
  It also provides the functionality to search for products based on the location provided and the dates given. These parameters (Start Date Time, End Date Time and Location) are retrieved from the *Product Search* page and the products are filtered based on this.
  Other functionalities such as getting all the products in the database and getting a product based on the ID is implemented.
- **User Service (user_service.py) -** This service provides the functionality to add the user details to the merchant table. The details are retrieved from the user registration page on submitting the form.
  The merchant login functionality uses the login service and login in the user to search for products.
  Other functionalities such as getting all the merchants in the database, getting a merchant based on the ID and to retrieve merchant details for the email provided is implemented.
- **Order Service (order_service.py) -** This service provides the functionality to add the order details when the user books the product in the *Product Results* page.
  It also provides the functionality to get an order based on the ID.
- **Upload service (upload_service.py) -** This service provides the functionality to upload a file to the Google Cloud bucket.
- **Report service (report_service.py) -** This service provides the functionality to generate a report and send a mail to all the merchants regarding the products listed by them and their status.
  It also provides the functionality to send mail to all the users at once so as to support promotional campaigns. It sends a mail using the *Email Service*.
- **Email Service (email_service.py) -** This service provides the functionality to send a mail using the SendGrid API where predefined email templates are defined.

**Templates:**

- **base.html -** The basic HTML page which is extended in all the other pages. This contains the header and footer of the page

- **index.html -** The main landing page of the website
- **merchantLogin.html -** The merchant login page
- **merchantRegistration.html -** The merchant registration page
- **merchantDetails.html -** The page which shows the details of the logged in merchant
- **addProduct.html -** The page accessible to the merchant to list a new product
- **userLogin.html -** The user login page
- **userRegistration.html -** The user registration page
- **userDetails.html -** The page which shows the details of the logged in user
- **productSearch.html -** The page accessible to the user where the user can provide the search criteria for renting a bike such as Start Date Time, End Date Time and Location
- **productResults.html -** The page accessible to the user which shows all the products that fit the criteria (Start Date Time, End Date Time and Location) provided in the *Product Search* page.
- **orderDetails.html -** The page which shows the orders of the logged in user

**Controllers:**

- **Root View controller (root_view_controller.py)**
  - */* - GET/POST method which redirects to the main landing page
  - *api/v1/retrieveRole* - POST method which checks and returns if the logged in user is a MERCHANT or USER
- **Authentication controller (authenticate.py) -** This controller is used for allowing only authenticated users of the site to access restricted pages.
- **Merchant API controller (merchant_api_controller.py)**
  - */api/v1/merchant* - POST method which receives the details from the merchant registration page and calls the *merchant service* to add the merchant to the database
  - */api/v1/merchantRegistration* - POST method which receives the details from the merchant login page and calls the *merchant service* to check if the merchant exists in the database
  - */api/v1/merchantRegistration* - POST method which removes the authentication token from the local storage and logs out the merchant
  - */api/v1/addProduct* - POST method which receives the details from the add product page and calls the *product service* to add it to the database. It requires authentication and redirects only if the logged in user is a MERCHANT
- **Merchant View controller (merchant_view_controller.py)**
  - */merchantLogin* - GET method which redirects to the merchant login page
  - */merchantRegistration* - GET method which redirects to the merchant registration page
  - */addProduct* - GET method which redirects to the page to add a product. It requires authentication and redirects only if the logged in user is a MERCHANT
  - */merchantProfile* - GET method which redirects to the merchant profile page. It requires authentication and redirects only if the logged in user is a MERCHANT
- **User API controller (user_api_controller.py)**
  - */api/v1/merchant* - POST method which receives the details from the user registration page and calls the *merchant service* to add the merchant to the database

    - */api/v1/merchantRegistration* - POST method which receives the details from the user login page and calls the *merchant service* to check if the merchant exists in the database
    - */api/v1/merchantRegistration* - POST method which removes the authentication token from the local storage and logs out the user
    - */api/v1/purchaseProduct* - POST method which adds the order details using the order service when the user books a product
- **User View controller (user_view_controller.py)**
    - */userLogin* - GET method which redirects to the user login page
    - */userRegistration* - GET method which redirects to the user registration page
    - */userProfile* - GET method which redirects to the user profile page. It requires authentication and redirects only if the logged in user is a USER
    - */viewOrders* - GET method which redirects to the page view all the orders of the user. It requires authentication and redirects only if the logged in user is a USER
- **Product View controller (root_view_controller.py)**
    - */productResults* - GET method which receives the details from the product search page and calls the *product service* to search for the products which satisfy the criteria provided in the details (Start Date Time, End Date Time and Location)
- **Report API controller (report_api_controller.py)**
    - */api/v1/generateMerchantReports* - GET/POST method which sends each merchant a status report about the products listed and its status
    - */api/v1/broadcastEmails* - GET/POST method which broadcasts emails to all the users

**Queue**

This is used to create a task for a given queue. It creates a task and pushes it into the task queue.

## 5.2 Installation

- Install the gcloud command line utils
- Install the dependencies using the command *pip3 install -r requirements.txt*
- Run *gcloud app deploy* to deploy the app onto the Google Cloud
- Create a database called pedalhire in Google Cloud SQL
- Run the following commands to set up the database -
    - *export $(cat .env | xargs)*
    - *python -m pedalhire.models.manage db init*
    - *python -m pedalhire.models.manage db migrate*
    - *python -m pedalhire.models.manage db upgrade*
- Run the following SQL statements in the Cloud Database
    - CREATE TYPE role AS ENUM ('USER', 'MERCHANT')
    - CREATE OR REPLACE FUNCTION calculate_distance(lat1 float, lon1 float, lat2 float, lon2 float, units varchar)

```
RETURNS float AS $dist$
  DECLARE
    dist float = 0;
    radlat1 float;
```

```
        radlat2 float;
        theta float;
        radtheta float;
    BEGIN
        IF lat1 = lat2 OR lon1 = lon2
            THEN RETURN dist;
        ELSE
            radlat1 = pi() * lat1 / 180;
            radlat2 = pi() * lat2 / 180;
            theta = lon1 - lon2;
            radtheta = pi() * theta / 180;
            dist = sin(radlat1) * sin(radlat2) + cos(radlat1) * cos(radlat2) * cos(radtheta);

            IF dist > 1 THEN dist = 1; END IF;

            dist = acos(dist);
            dist = dist * 180 / pi();
            dist = dist * 60 * 1.1515;

            IF units = 'K' THEN dist = dist * 1.609344; END IF;
            IF units = 'N' THEN dist = dist * 0.8684; END IF;

            RETURN dist;
        END IF;
    END;
$dist$ LANGUAGE plpgsql;
```

- Create a queue using the command *gcloud app deploy queue.yaml*
- Create a task using the command *python pedalhire/queues/create_task.py*

## 6. Conclusion

We were successful in finishing the first phase of our project where we built a end-to-end web application which caters to two types of users, MERCHANTS and USERS. A Merchant is provided with the functionality of listing his products on the site while the User is provided with the functionality of searching and booking the product. We deployed this app;ication on the Google Cloud platform and integrated a lot of cloud components in our application such as Task Queues for performing the asynchronous task of sending emails and Memcache to cache database results and reduce the database calls.

In the future, we aim to include more features such as a feature to form communities among users to allow various discussions, a functionality to show the most famous bike trails nearby, allow the user to pick up and drop at different locations, etc. These additional features would make our application very enticing for the users and provide a rich experience for everyone using the application.

# 7. Individual Contributions

**Vivek Kumar Maskara (ASU ID: 1218403147)**

In this project, we were required to build a PaaS based cloud application that optimally uses various Google cloud services. We chose to build a peer to peer cycle renting app(PedalHire) as nothing of this sort exists for cycles. The main idea was to let people, especially students, earn some extra bucks by renting out their unused cycles for a few hours/days. Our application uses Google cloud as its backend and in particular it utilizes Google App Engine, Cloud SQL, Cloud Storage, Cloud Scheduler, Task Queues and Memstorage services. Here's my contribution to each of the phases of the project.

**Design**

The design phase involved coming up with the architecture for the whole system and deciding how each component interacts with each other. I came up with the initial architecture and collaborated with other team members to iterate and improve it. I decided that we should use Google App Engine for hosting our application and use the Python based Flask framework for our backend. I also proposed that we use Google task queues for our long running asynchronous operations ie. broadcasting emails to the users. I picked SendGrid email client to efficiently send out bulk emails whenever required. Also, I helped in defining the broad structure and responsibilities of our backend application.

**Implementation**

My major contribution was in setting up the Google App Engine backend. I set up the initial application structure using Flask and SQLAlchemy and took care of setting up the deployment process for the app. I contributed in implementing most of the backend APIs for the app's core functionalities. Apart from that I also contributed in setting up a Google Cloud Task Queue, creating tasks and creating a task handler for broadcasting emails to the users. In this process, I also configured the SendGrid email client to manage email templates and integrated it with our Python backend. Also, I integrated Google Cloud Storage to upload CSV reports when the reports are generated. I also helped in setting up Google Cloud CDN caching for our application to cache static resources.

**Testing**

Testing in a frontend based application becomes very important and we made sure to test all the functionalities after every round of deployments. The application has several components and I helped in multiple rounds of testing to make sure everything was working perfectly. I helped in testing the core functionalities and monitored the logs and latencies in the Google App Engine dashboard to ensure that the application was performing as expected. It was important to monitor the average API latencies and to ensure that the memstore was giving us a high hit rate. Also, I helped in testing and monitoring task queue logs to validate that the tasks were getting executed as expected.

**Rahul Santhosh Kumar Varma (ASU ID: 1218475349)**

The goal of the project was to build a cloud-based peer-to-peer bike rental platform. We created a Flask application with the UI developed using Jinja templates. This application was hosted on the Google Cloud platform and utilizes the Google App Engine, Cloud SQL, Cloud Storage, Cloud Scheduler, Task Queues and Memstorage services. The goal was to optimally use these resources such that the cost could be kept at a minimum and at the same time have the ability to reliably scale up the application based on load. The application was built by keeping in mind its scalability and cost factor. I have listed my contribution during each separate phase of the project.

**Design**

During the design phase, I was involved in designing the entire database and its relations. The design of the database was done by keeping transparency and privacy in mind. I ensured that the database was normalized to avoid data anomalies. I also collaborated with the other members to decide on a few components that would ensure that the application we built is scalable and can cater to a large number of clients. I proposed that we use Google Scheduler to send the updates to the merchants about their products based on the orders placed. Apart from that, I was responsible for designing the UI of the entire application.

**Implementation**

I created and implemented the entire database in Google SQL and ensured that it was fault tolerant. I was also responsible for designing the UI of the application using Jinja templates. I integrated the Google Maps API into the website to facilitate a map view of the product locations. I was also responsible for implementing the Google Scheduler which sent an update to each merchant regarding the products he has listed and the status of the products. To send mails, I designed a SendGrid template which was used to send out the mails. I also facilitated the method for storing the product images in Google Cloud Storage.

**Testing**

The testing phase was one of the phases where we spent a lot of time to ensure that the entire end-to-end functionality worked as expected. I ensured that there were no UI bugs and worked to ensure that the routing protocols i.e. the Google endpoints worked as expected. I also tested the Google Scheduler to check the email delivery system to all the merchants. Me and my team ensured that the application was fault tolerant and could handle multiple concurrent requests with it scaling up as the demand increased.

**Manikanta Vankayala (1218816326):**

Our main focus in this cloud based pedal-hire project is to build the application which is more robust, reliable and scalable. For this we have used the following cloud resources Google App Engine, Google memory store, Google scheduler, Task Queues, Google cloud sql, Google data storage,Google VPC connectors, Task Queues, Google CDN, and Google app endpoint.

**Design:**

In the design phase, Myself with my team mates come up with a best architecture which can handle all the parameters like performance, scalability, reliability and many more. I have contributed to improve the performance, I came up with an idea to utilize the google memory store to improve the performance, contributed while designing the SQL schemas, contributed to addition of google maps to identify the location of the place which the user mentioned.

**Implementation:**

In the implementation phase, I have contributed to develop the front end, implemented the cache to improve the performance of the application using google cloud memory store and further write some SQL query to get and update the data from the google cloud SQL. Further, I have contributed to add VPC connectors to connect the private network redis instances, and along with my teammates worked to improve the code efficiency.

**Testing:**

In testing phase, I have tested the memory store performance which the resulted cache hit ratio of 66% and further, I have done integration testing and stress testing to check the code breakage and further, Myself along with my teammates tested the end to end functionality of the application which lead the application to become more robust.

## 8. References

[1] *GMT20200331-190059_Ming-Zhao-_1280x720*, Ming Zhao, 31 March, 2020, Zoom, https://asu.zoom.us/rec/play/7pd_Jez7q2g3SNLB4wSDUaVwW461fPishygX86JezBq2AiFVYVvyNeQSNuVpAG_QDPrTRfjt9pEbUpjF?continueMode=true&_x_zm_rtaid=uRyiMyM3SmSsnbEgUkDPvA.1589153762358.6f60bcd86c78d6fa31796391afc6d6aa&_x_zm_rhtaid=964.
[2] Google Cloud. 2020. *App Engine | Google Cloud*. [online] Available at: <https://cloud.google.com/appengine> [Accessed 10 May 2020].
[3] Google Cloud. 2020. *Task Queue Overview | App Engine Standard Environment For Python 2*. [online] Available at: <https://cloud.google.com/appengine/docs/standard/python/taskqueue> [Accessed 10 May 2020].
[4] Haber, I., 2020. *Why Redis Beats Memcached For Caching*. [online] InfoWorld. Available at: <https://www.infoworld.com/article/3063161/why-redis-beats-memcached-for-caching.html> [Accessed 10 May 2020].
[5] Google Cloud. 2020. *Cloud Scheduler | Google Cloud*. [online] Available at: <https://cloud.google.com/scheduler> [Accessed 10 May 2020].