CSE 546

Sun Devils Roommate Recommender

Rahul Advani (1217160594)
Shantanu Purandare (1217160516)
Shreya Darak (1215202846)

1. Introduction

For almost all non-resident incoming students, one of the most time consuming and more importantly, stressful tasks before starting their program is, finding good-natured and like minded flatmates. There are a multitude of factors which students consider while choosing their flatmate such as the amount of rent they are willing to pay, the size of the apartment or even an individual's meal preference. Finding potential flatmates is a manual and long process, which usually involves posting advertisements across different social media platforms and waiting for days to get a genuine response. Through our web application we aim to streamline this process and provide an automated and quick way to find potential roommates based on the similarity between their professional and personal interests.

2. Background

We propose a scalable web application which will allow incoming students to list their preferences such as their personal information and choices, as well as specific ASU related questions through which we intend to extract and record parameters that will be useful in determining compatibility among the students. Once a student has completed the form, his information will be added to the user pool and using the parameters he/she entered we will propose a list of compatible flatmates. Thus, the technologies most relevant to our project would be-

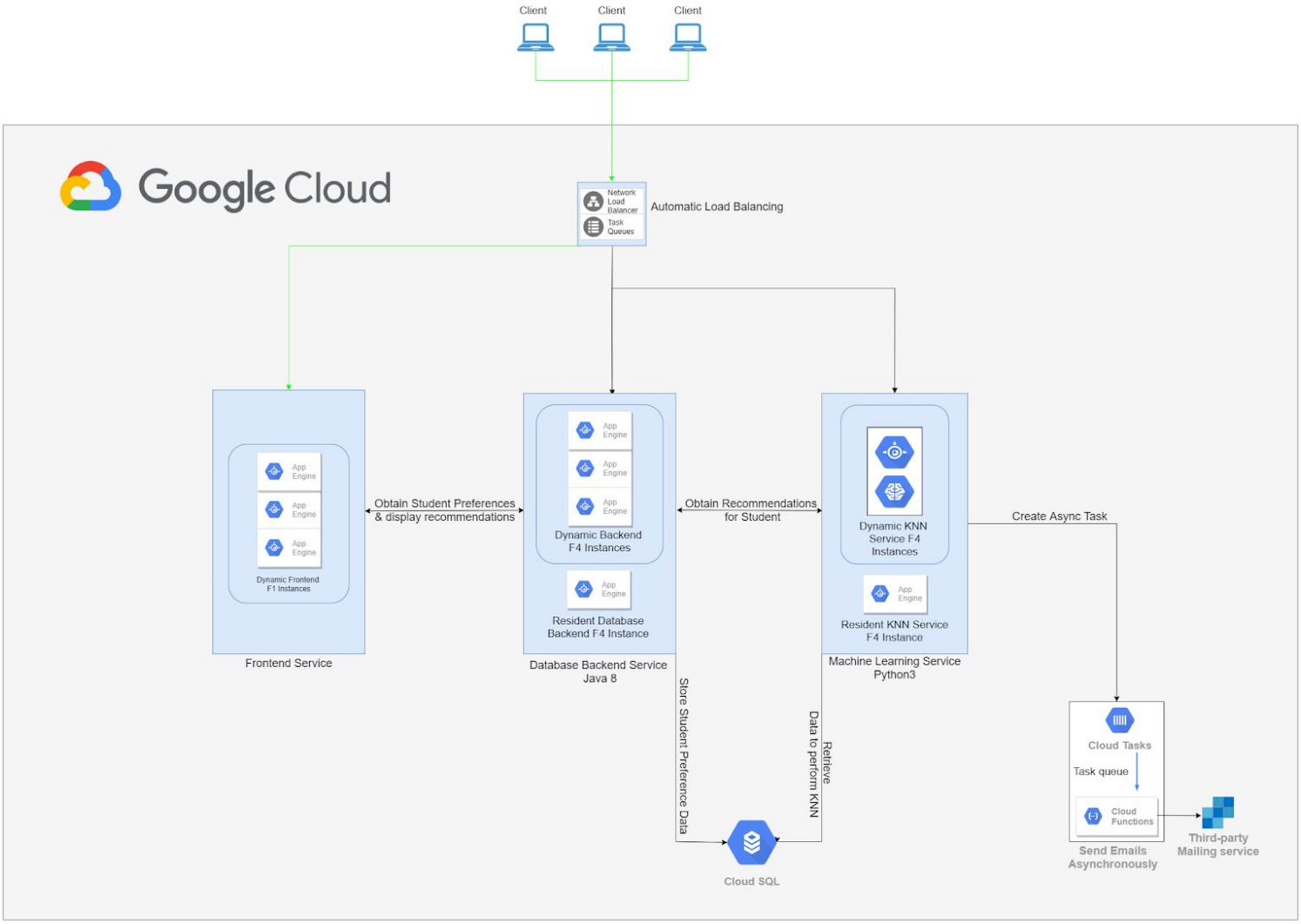
1. A front end service that the users could interact with.
2. A backend service that stores the user preferences and interacts with the ML module.
3. An ML module that uses machine learning to find similar users to group them together.
4. Auto Scaling, which is used to make the application responsive and reduce end-user latency which is by far the most important factor from a user's perspective.

Incoming students find it extremely stressful to find good roommates. More often than not it is quite a manual task. For example, an incoming student might have to post advertisements on multiple social media platforms and after spending quite a lot of time in this process may or may not finally find some people that may be interested, all this before you even get to know the person.

Universities in the US do not usually help out students with finding roommates outside the on campus housings. So, there are no solutions like this available for university students. However, if you google ‘roommate finder’ you get a lot of options, the top 2 are websites roomster.com & rommates.com. The fundamental difference between our app and such existing solutions is that they are based on an advertisement model i.e. as a user you need to own/rent an apartment to place an advertisement for the requirement of roommates,people can view your ad then and express their interest. So, the existing solutions are not sufficient for our case because a large portion of incoming students (especially international ones) do not own/rent an apartment months before they start their program. However, in our app we do not need any such prerequisite. Our app is targeted at grouping people having similar personalities together to connect and collaboratively find an apartment.

3. Design and Implementation

Architecture



Components:

- **Frontend Service:** Frontend service is deployed on google app engine using Python web server flask that will take care of server side processing. Whenever a user sends a HTTP request to the frontend application, this service generates UI pages visible to the user. These web pages are developed using HTML, CSS and Javascript. Bootstrap is also used for beautification of the web pages. User registration and recommendation requests will be served by a backend application. Response from backend service will be shown to the user in the table format.

- **Database Backend Service:** The Database Backend service is a spring boot application running on GAE and connects all the services together. By exposing an API the information entered by the user is captured and placed into a Cloud SQL instance. After placing this entry into the database, the backend service calls an API exposed by the machine learning service and sends the primary key of the row it just inserted as a path parameter. The ML service applies the KNN algorithm and returns a list of student recommendations. The list returned by the ML service is then forwarded to the front end service to display it to the user.
- **Machine Learning Service:** The ML service is deployed on Google App Engine using Python3 as the environment and uses flask for its web functionalities. This service is used to recommend similar uses in response to a rest api call generated by the Database backend service targeting an endpoint in this service. This service also creates a task to execute a 'cloud functions' method using the cloud tasks client and adds it to the push task queue.
- **Cloud SQL:** Given the type of data, using a relational database was an obvious choice. For the purpose of this project we were storing all of the user data in a cloud SQL database. The backend service and the ML service interact with the database to insert and retrieve the user data. The schema of the database was designed on the basis of the json object which is forwarded between services.
- **Mailing Service:** The mailing service consists of several cloud services. These services work together to send emails to the users, containing a list of possible matches based on their own profile. Since sending emails is a time consuming task, the cloud services used to satisfy this functionality work asynchronously to the rest of the application. This allows the rest of the application to be fast and responsive while the mailing service works slower without affecting the efficiency of other services.

Cloud Services:

- **Google App Engine(GAE)** - We use GAE to deploy our 3 main services, namely: frontend, backend and the ML service. The Database backend service runs on Java8, while the ML service runs on a python3 environment. The front end service consists of HTML and Javascript files which are rendered by Python Flask library.

- **Cloud SQL**- We were dealing with tabular data which made using a relational database a good choice. We use the Cloud SQL service provided by GCP as the database backend to store the user data that is entered by the user. The other services running on GAE like the backend service and the ML service interact with the cloud SQL to insert, maintain and retrieve the data.

- **Cloud Tasks**- We use cloud tasks client in our ML service to create new tasks and place these tasks onto a task queue to schedule them for execution. The task is to run a cloud functions method asynchronously from the rest of the application. We use a push queue for

our purposes. A push queue explicitly triggers an http request for each task targeting the method to be executed by the task.

- **Cloud Functions**- Cloud Functions uses a Function-as-a-service paradigm. Here you can select the environment you want your function to run on. We used python3 for our mailing service. The cloud functions method ran the actual code for sending mail to a user containing the list of recommendations for him/her.

- **Sendgrid-** Sendgrid is the third-party mailing service that was used to send mails to the users in bulk. Google cloud platform supports the use of this service and we were able to incorporate it in our pipeline by calling this service from our python code inside the cloud functions method.

Role of App Engine and AutoScaling:

We used App Engine in our application to run the 3 main services that make up most of our application's functionalities- the frontend, the database backend and the ML service. As GAE follows the Platform As A Service (PAAS) paradigm, the instances on which these services run are created/scaled-up based on the number of requests they are handling, and as developers we do not need to worry about scaling up/down of these instances. The frontend service provides users with an interface to interact with our application. The database backend service interacts with the frontend, the actual database and the ML service to get data the user has entered, store it in the database and make requests to generate recommendations. The ML service is used to make actual recommendations based on the user input and similar users present in the user pool. It also triggers another service that is used to send mails asynchronously to the users specifying their matches.

App Engine provides automatic auto scaling functionality so the services we deployed scaled in and out automatically. We had to configure the auto scaling features to suit our design and the traffic the application handles.[1] The following configurations were set for our applications-

- Max_concurrent_requests
- Min_pending_latency & Max_pending_latency
- Target_cpu_utilization & Target_throughput_utilization

Max_concurrent_requests : It is the number of concurrent requests an instance can handle before scaling up. We set this value to 7.

Min_Pending_Latency & Max_Pending_Latency : These values specify the amount of time a request can wait before it gets served. If it waits for time greater than the min value then that

[1] "App Engine, Scheduler settings, and instance count. - Medium." 27 Apr. 2017, https://medium.com/google-cloud/app-engine-scheduler-settings-and-instance-count-4d1e669f33d5.

instance is eligible for scale up, if its wait time is greater than the max limit then it has to scale up. We set the min and max pending latency values to 7s and 10s respectively.

Target_cpu_utilization & Target_throughput_utilization : These values specify the threshold values for CPU and throughput utilization, if an instance's values cross this threshold then it scales up. We set both thes values to 0.7 for our application.

We decided to set these particular values for the above mentioned scaling parameters by testing our application for increased responsiveness.

Apart from this we also had 1 resident instance that was always running for the database backend and the ML service since these two services did most of the heavy lifting. As a result of which both services had F4 instances, on the other hand the front end service was running on an F1 class. Having a resident instance in the application guarantees that requests are constantly getting served without any long interruptions in between. This was especially important for us, as when the instances were scaled up, the latency spikes were massively reduced.[2]

Advantages of existing solutions:

To put it bluntly, ASU and other universities do not help students find roommates. We believe that finding good people to live with is very important when moving to a new place. And our project would provide incoming students with a platform to find new roommates based on similar profiles and needs. This would make a long, stressful and manual process of finding compatible roommates much more easier. As we were unable to test it with any incoming students, we tested our solution with students who currently have roommates. Our application gave students almost similar recommendations to the roommates they currently have. Thus we are leaning towards the fact that our application is quite effective.

4. Testing and Evaluation

As our application is split into microservices, one of the main factors which we stressed on testing was the communication between services and the data being passed betweens services via REST APIs. As a result of which our main focus was the end-end communication between services, i.e. the JSON object being sent to the backend and later ML service and the list of JSON object responses which were propagated from the ML service to the backend service and finally to the front end service. We incrementally tested each service of our application by exposing a REST API on each of the services, and firing them with test-data. We would then evaluate the response of the request or look at the contents of the database to validate the service's functionality.

---

[2] "App Engine Resident instances and the startup time problem." 4 May. 2017, https://medium.com/google-cloud/app-engine-resident-instances-and-the-startup-time-problem-8c6587040a80.

Another important factor which we set out to test was the end-end latency of our application and the automatic scaling handled by GAE. For which, we implemented a script that stress-tested our application by concurrently firing multiple REST API calls. Through this script, we also captured the latency recorded for each request to be completed. As per our tests, we realised that multiple concurrent requests cannot be handled by the backed and ML services which initially had F1 instances. Following which, we shifted to a F4 instance type for these two services. Similarly using this test script, we realised that when instances scale up, the latency spiked for a few seconds before reducing significantly, this observation made us deploy 1 resident instance for the backend and ML services. This significantly reduced the latency for the first request and subsequent requests during scale-up.

Empirically, we recorded the following latency while firing the '/getPreferences' API which saved a record in the DB and returned a list of recommendations.

Latency Recorded	First Request Latency (Upper Bound)	Mean Latency During Scale Up (Upper Bound)
No resident instance	11 Seconds	35 Seconds
One resident instance	2 Seconds	13 Seconds

With our current autoscale configuration, the mean latency of 10 concurrent requests is 3.2 seconds.

5. Code

Functionality:

Frontend Service:

We have designed visually simplistic User Interface pages to focus on the functionality and features of the application. There are mainly two web pages in the frontend service developed using HTML, Bootstrap and javascript.

1. Registration Page: Whenever a user comes to our website in intent of finding roommates, he/she will have to register with personal details as well as ASU program details. Also users can mention food and rent preference. After submitting details, frontend services will call backend REST endpoint with user form data. Some of these fields are mandatory to provide accurate recommendations for roommates. After successful registration of the user, user data will be saved in the cloud sql store.

2. Recommendation page: This page shows a listing of recommended roommates according to details provided by the user.This list is in table format and the same list will be mailed to the user. It will have details of recommended roommates so that the user can contact them as well. These web pages are deployed in the GAE environment using Python flask server and are configured for auto scaling as per the incoming http requests.
3. main.py : This page will take care of server side processing of the web app.Here we have imported the flask module and it will figure out requests this app is getting and routing of the requests to particular page/functionality in the code. It will also take care of the response to send back to the user.

Database Backend Service:

The Database Backend service is developed using Spring Boot Framework in Java8 to handle the requests sent by Front end service. This backend service will accept user form data sent using HTTP POST request. The Frontend sends the user input in the form of a JSON object and sends it to the backend service by using the '/student/getPreferences' API. This data collected by the backend service will be put into a database running on a Cloud SQL instance. Once the data is stored in the Cloud SQL instance, a client for the ML service will get created. Using this client, an API exposed on the ML service is called, which returns a list of recommended roommates for this particular student. This response was then propagated to the front end service. As we are using the Spring Boot framework, we do not have to worry about dependency injection and bean creation which is necessary while creating a web application.
Brief overview of each package in the backend service-

1. com.asu.recommender.constants : Constants which are used through the project are set in the classes in this package. For example: The URL of the ML service is statically set in this package.
2. com.asu.recommender.controller : The classes in this package are responsible for exposing REST APIs which enable this service to be accessible to other services. In the controller classes in the package we are calling the ML service and sending its response to the front end.
3. com.asu.recommender.model : The Plain Old Java Objects (POJOs) classes are defined in this package. These POJOs define the JSONs which are accepted via the REST APIs and the table schema in Cloud SQL is based on these POJOs.
4. com.asu.recommender.repo : The database connection is established in the classes in this package.
5. com.asu.recommender.service : The Logic for filtering out recommendations is coded in the classes for this package.
6. src/webapp/WEB-INF : The appengine-web.xml file is present in this folder, which is responsible for setting params for the backend service when it is deployed in GAE. Parameters such as auto scaling are set in this file, in the form of XML Tags.

ML Service:

The ML service is in main.py file and provides the following functions:

1. Retrieve user pool data from Cloud SQL database.
2. Run KNN algorithm on the data to find similar users
    a. Calculate euclidean distance for numeric fields like Age, Minrent, Maxrent
    b. Convert categorical fields like Food preference, Major, program, gender into a one hot vector encoding to calculate the distances.
    c. Use a sparse matrix of TF-IDF features followed by a dimensionality reduction algorithm called SVD to convert the description field consisting of unstructured data into features.
    d. Using these features, the KNN algorithm calculates similarities between users and returns a list of K matching users specified back to the backend service.
3. Create tasks using cloud tasks client and add it to the push queue for asynchronous execution. These tasks contain the request and payload to cloud functions method that will send email to the user that made the request. Payload consists of the email id of the user and information of all the similar users.

The app.yaml file has configurations for deploying the application on app engine. The following attributes were used-

- Runtime(Python3)- specify environment
- service(testservice)- name of service
- Instance_class(F4)- higher performance instance class since ML module does lot of computations
- Auto scaling- Set features like max concurrent requests on the app engine instance, max and min latency values for starting a new instance and having some instances always running.
- Request handlers and redirectors- For handling incoming requests to our application.
- Environment variables- For storing app specific variables.

Requirements.txt file contains names and versions of the python modules necessary for running the application.

Cloud Functions:

Cloud functions method ‘send_email’ is triggered by the task created in the ML service. This task interacts with the third-party mailing service called SendGrid to send emails in bulk. These emails contain information of all the similar users including their emails so that the user can contact them.

Stress Test Script:

For testing the autoscaling of the application, we developed a python script which concurrently fires APIs on the application to generate the recommendations. Through this script

we captured the time elapsed in completing a request. This was crucial for us as it helped us tweak the auto scaling parameters which inturn reduced the applications latency.

Installing and setting up the project

We recommend running the application on the GCP environment, as that will require fewer installations and is relatively straightforward.

Running the application on GCP: To run the project, you need access to the 'RoommateRecommender' project which is a part of our Google Cloud Environment. The project needs to be executed in the above mentioned environment, as we are making use of SQL instances, task queues and cloud functions which reside only in our project. If you need access, please send an email to 'radvani1@asu.edu' and you will be given the necessary privileges for the project. Once you have access you need to ensure that you have the google cloud sdk installed on your local machine. After which, you need to change the project your local SDK is pointing towards, to 'RoommateRecommender'.
The detailed steps and commands to run each component of the project on GAE are placed in the readme.md file.

Running the application locally: To run the application locally, you need to manually run every service locally. The instructions for running each service locally is explained in detail in the later section of the readme.md file.

6. Conclusion

Learning and Accomplishment:

- Got accustomed with Google cloud platform and the various services provided.
- Interact with GCP using the UI and console provided by Google as well as using the cloud SDK provided for developers.
- Created a PaaS web application by deploying multiple services on GAE.
- Learned how GCP manages autoscaling and how the parameters for auto scaling can be fine tuned for achieving better performance metrics.
- Learnt about and implemented loosely coupled design patterns such as Microservices.
- Cloud Tasks, learned how to use cloud tasks to decouple the architecture and support time consuming tasks by running them asynchronously without creating bottlenecks in the execution.
- Cloud Functions, used FaaS paradigm to achieve the decoupled architecture by triggering cloud functions using cloud tasks.
- Learned how to integrate third party mailing services with a cloud application.

Ideas for Improvements:

- As the user data increases, KNN will start becoming slower since it is a lazy learner and spends all of its time in the inference phase and basically has no training phase. To avoid this issue, we can make several changes-
    - Save the feature matrices of the users so that there's no need to create them again for every request.
    - Filter users based on the preferences of the user being queried to reduce the amount of data that has to be worked on.
    - Use different recommendation services like K means clustering. The main benefit of using this type of algorithm would be the ability to save a model and use the same model for subsequent requests thus avoiding the need to calculate the similarities for each request.
- To make the applications more efficient we can make the task of inserting data into the Cloud SQL relational database asynchronous using the cloud tasks and queues.
- Integrate Firebase authentication module to make our application more secure.

7. Individual Contributions

Rahul Kamalkumar Advani (1217160594)

Design

Completely designed the backend service application running on the Google App Engine to store the user information into Cloud SQL. As the backend service was central and connected to all the components in our design, I designed the interface between the backend service with the other components. For which, I developed and designed clients and REST APIs to communicate with other services. According to the payload captured from the front end service, I created a database with the corresponding schema in Cloud SQL. After testing the autoscale and latency recorded for concurrent requests, I designed each service's autoscale policy.

Implementation

Backend Service: I used the Spring Boot framework in Java to build the backend service. I specifically worked on writing code for accepting the student information and placing it into a Cloud SQL instance. As this service was accepting information from the Frontend service, I exposed a REST API which was called by the frontend service. Once the data was stored in Cloud SQL, I created a client for the ML service. Using which, an API exposed on the ML service was triggered, which returned a list of recommended roommates. This response was then propagated to the front end service.

Frontend Service: I partially worked on the Javascript code to package the student information entered in the form and send it in the body of the REST call to the backend service and designed the CSS of the User Interface.

Autoscale Test Script: To ensure the autoscale configuration set by us was working well in our application, I implemented a Python script which concurrently called an API exposed by the backend service. This was achieved by using threads, specifically, a thread for every request.

Testing

As a part of testing the autoscaling of the three services, I wrote a python script that concurrently fired 10 APIs on the backend service to store a student's information in the DB and call the ML service to generate the recommendations. As part of the script I even captured the time elapsed in completing a request, as per our observation I noticed that on scaling the latency spiked for a few requests before drastically reducing, to fix this problem, I came up with the design to deploy resident instances for the backend and ML service. This in turn significantly reduced the latency. Also, as our design was split into microservices, along with my group, I regularly tested the communication between each service and its end-end functionality.

Shantanu Purandare (1217160516)

Design

Designed the ML service to generate recommendations for a particular user using the KNN algorithm to find similar users in the user pool. The main design features included- 1. An interface to interact with the relational DB to fetch all the user data. 2. Preprocessing this retrieved data into the required format so that the KNN algorithm could use it for generating recommendations. Performing KNN and sending the results back to the Backend service in a json format. 3. Creating tasks using the Cloud Tasks client.

Designed the complete mailing service which involved the interaction between the following components- ML service, cloud tasks and queues, cloud functions and a third-party mailing service called SendGrid.

Also helped design the overall architecture of the web application to make it service oriented and decouple the components so that they would be able to function and scale in or out independent of each other.

Implementation

ML service: I used python and its flask module to create the ML service. In this service, I implemented the functionality to retrieve data from the cloud sql database, preprocess this data to convert it to a format acceptable by the KNN algorithm. This was done using the sklearn package's excellent features like Pipelining and Transformers. These transformations included one hot vector encoding the categorical data and generating TF-IDF features out of the unstructured fields and applying dimensionality reduction algorithms on these fields to maintain the efficiency of the service.

Mailing service: Created the push task queue. Used cloud tasks client in the ML service to create tasks and place those in the push queue. Each task was assigned a target url and a payload that contained the email of the receiver and the data of similar users suggested by ML service. The target url in a task pointed to a cloud functions method called 'send_emails' which was used to actually send the emails containing the user data with the help of the third party mailing service called SendGrid.

Testing

Performed unit testing each of the following individual components- ML service, Cloud tasks and queues, Cloud functions. Worked on the integration testing for the components within the mailing service as well as the integration between the backend app-ML service , ML service-mailing service. With the help of my teammates, I performed end to end testing and stress tests on the application to capture responsiveness.

8. References

The following articles were essential in our understanding of the autoscaling in GAE. The following articles also helped us solve problems regarding the high latency and crashing of instances.

1. https://medium.com/google-cloud/app-engine-scheduler-settings-and-instance-count-4d1e669f33d5
2. https://medium.com/google-cloud/app-engine-resident-instances-and-the-startup-time-problem-8c6587040a80

The following two papers helped us establish validity to our approach followed in our Machine Learning Service

1. Yilmaz M., Al-Taei A., O'Connor R.V. (2015) A Machine-Based Personality Oriented Team Recommender for Software Development Organizations. In: O'Connor R., Umay Akkaya M., Kemaneci K., Yilmaz M., Poth A., Messnarz R. (eds) Systems, Software and Services Process Improvement. EuroSPI 2015. Communications in Computer and Information Science, vol 543. Springer, Cham

2. https://arxiv.org/abs/2004.06970

This article was helpful in understanding the flask application and its role in server side processing.

1. https://www.freecodecamp.org/news/how-to-build-a-web-application-using-flask-and-deploy-it-to-the-cloud-3551c985e492/