CSE 546 — Quick'Ply

Divya Kshatriya (ASU ID : 1217888165)
Lakshmi Sindhuja Karuparti (ASU ID : 1217211723)
Nipun Mediratta (ASU ID : 1217888464)

Introduction

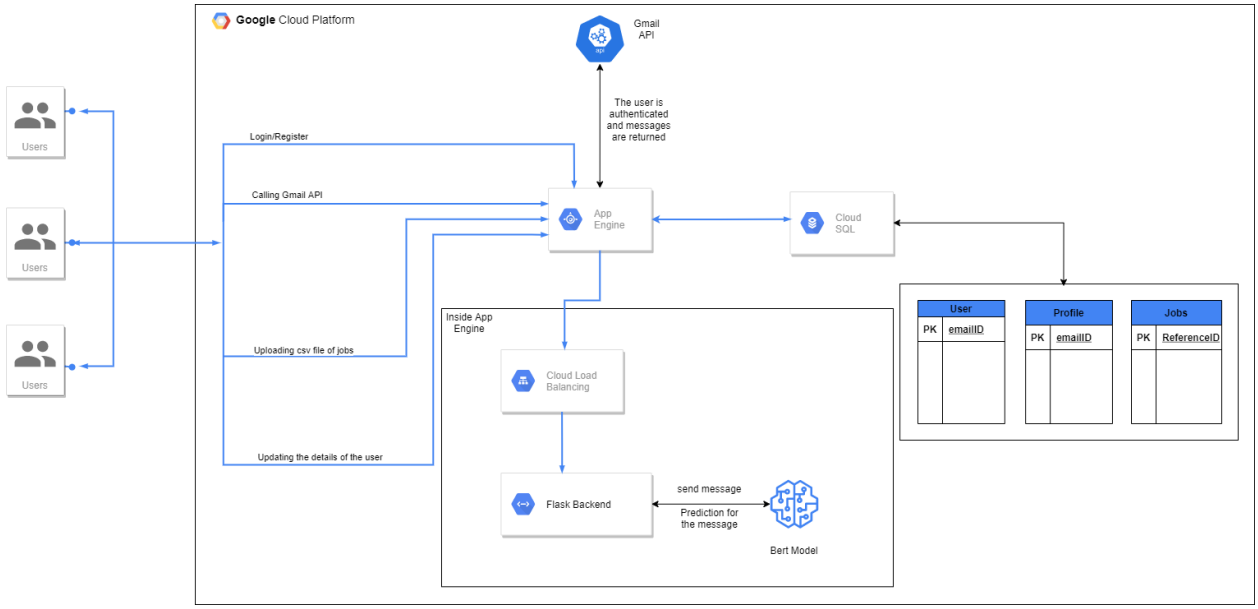
Job seeking process is a very hectic process. Added to this, keeping track of various job applications is also an added responsibility. Job seekers, mainly undergraduate and graduate students, apply to copious jobs and internships. As a result, many students also maintain an excel sheet that keeps track of their job applications along with the statuses of the corresponding applied jobs. In the event of applying to many jobs, sometimes students forget to update the excel sheet as a result of which they tend to apply again to an already applied job and they either get a message stating that they have already applied to that job or it just gets accepted thereby becoming redundant. This is a real problem. However, there is no real solution to this problem. We aim to bridge the gap between this problem and the lack of solution by providing a state of the art and cutting edge solution using Machine Learning and scalable Cloud technologies. This application will be available for every user who registers with us.

Background

This is a common problem faced by the students. However, there are no solutions offered to this problem. This is what motivated us to address this problem by building a scalable cloud application. It is an application that automates the process of students updating and tracking their job applications with just one click of a button.

This is a very important problem that needs to be solved. It has been there since ages. Traditionally, in order to apply to jobs, job seekers apply to many jobs as well as maintain a separate excel sheet where they update each of their job statuses. In addition, whenever they receive any response from the companies they've applied to, they update the statuses for each of the corresponding jobs in the excel sheet. This process is repeated until they find a job of their choice. This becomes very tedious, unnecessary, and counter productive. It is counter productive because job seekers should rather spend their precious time on networking and improving their resumes for every job that they are applying to. There are no existing solutions that cater to the problem described above. This entire process of applying to jobs follows a pattern. Thus, it can be automated. Therefore, we solve the problem by building a scalable cloud application that automatically updates the statuses of the users' job applications with just the click of a button. The solution proposed in this report is first of its kind and we plan to deploy this implementation into production.

Design and Implementation



In this project, we have four different flows as seen in the diagram above. These are explained below.

1. **Login/Register**: Every user intended to use our application needs to authenticate using login/register functionality. Once the user successfully authenticates, his or her Dashboard would be displayed. Users can use this Dashboard to upload jobs, view the status of their jobs or even have their jobs updated using their emails from his or her account with just a button click.
    a. **Login**: A user who has already registered will need to login using this functionality to view his Dashboard. User needs to enter his email and password to login. This will invoke the checkuser function in routes.py. When this is invoked, in the backend, a check happens with the user table (which contains users' credentials) in the Cloud SQL database. If the user gives the wrong password, a flash message saying "Wrong password! Re-enter" will be displayed. If the user does not exist and if he still tried to login, flash message saying "No user exists! Please register!" will be displayed and the user will be redirected to the register page. If the user successfully logs in, his Dashboard will be displayed with a flash message saying "Successfully logged in! Here is your Dashboard"
    b. **Register**: Every new user will be directed to register. The various options available in the register page are Upload resume, First Name, Last Name, Phone Number, Email, Password, LinkedIn Url, Twitter Url, Github Url, Portfolio Url, Other Website Url, Are you legally able to work in US?, Will you now or in the future require sponsorship to be provided?, What is your age range?, What are your racial, ethnic and origin identities?, What gender do you identify as?, Are you a veteran? and Do you have a disability? Though the current functionality provided required only the email and password of the user, all this information is collected as one of the major enhancements for this project

would be to have our application apply jobs for the user when the user gives the job link. In these options, some of the options are marked with a * which means the user cannot click on the submit button unless he gives input in these. If the user gives the email with which he already registered earlier, a flash message saying "You are already registered, please login" will be displayed and the user will be redirected to the login page. If the registers successfully, flash a message saying "You are now successfully registered! You can now login" will be displayed in green and redirects the user to the login page. In the backend, the profile table in SQL Cloud database will be updated with the registered user's details

2. **Upload jobs:** This is an option available in the user's Dashboard viewed as "Uploading csv file of jobs" route in the diagram. When the user clicks on 'Choose File', he can choose csv that contains the details of all the jobs he applied for. In the csv he uploads, he needs to have four columns namely email, company, jobtitle, jobid in the same order. Once he uploads, he should click on the Submit button right next to the 'Choose File' option. With this click, the jobs table in the backend will be updated with the default job status as "Applied". And on the User Interface, a flash message saying "Jobs are submitted" will be displayed.
3. **Update jobs from emails:** This is the key functionality of our application. This is also available in the Dashboard viewed as "Calling Gmail API" route in the diagram. When the user clicks on this button "Update jobs from emails", he will be redirected to gmail login and once the user logs in via gmail which means letting our application access his emails, each company user applied to will be filtered in his gmail account, the text in the email is read and sent to BERT model which classifies the text to "In_Process", "Offered" or "Reject".
4. **Display jobs and statuses:** This functionality is also available in the Dashboard viewed as "Updating the details of the user" route in the diagram. When the user clicks on "Display jobs and statuses", displayjobs function inside routes.py is called and all the jobs from the jobs table in the backend matching the user's email will be displayed in the User Interface.

This project was implemented using Machine Learning and Scalable Cloud Technologies as follows

1. **Google App Engine (GAE)** : to host our application and to handle all requests from the clients. Specifically, we used the GAE flexible environment.
2. **Google Cloud SQL** : to store the information of users using our application and the users' job status information.
3. **APIs** (Application Programming Interfaces) -
    a. **Gmail API :** Gmail API was used so that our application can request access to the emails of users for updating the statuses of their job applications by using our Machine Learning model.
    b. **Flask API** : Flask API is a web framework for Python that is used for building the web application, managing the HTTP requests and for rendering the HTML pages.

c. **SQLAlchemy :** SQLAlchemy is the Python SQL toolkit and Object Relational Mapper that gives application developers the full power and flexibility of SQL.

4. **Machine Learning** : BERT (Bidirectional Encoder Representations From Transformers) model is used to classify users' emails into one of the three categories - In_process, reject, offered.
5. **Web Technologies** : Web Technologies like HTML, CSS, JavaScript were used as part of our FrontEnd.
6. **Languages** : Python was used to build our backend as well as to implement the BERT model.

Some of the notable components in our application are:

1. **Google App Engine (GAE)**: This is 'Platform as a Service' (PaaS) provided by Google Cloud Platform. This is used to host our entire web application. After the application is deployed, an endpoint "https://enhanced-rarity-312105.uc.r.appspot.com" is provided which when opened in any application browser like Google Chrome, Mozilla Firefox etc would display the main page of our application and allows the jobs to use all the functionalities our application provides. This could be opened both on web or mobile.

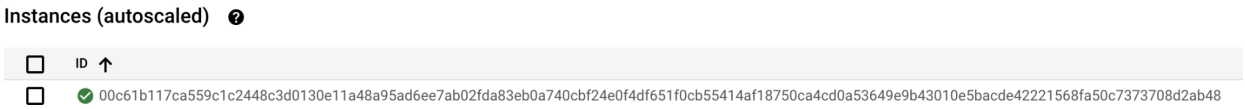
   When the application is hosted, our entire Flask Application is deployed on GAE with the starting point being main.py. A new version of our application is created in GAE everytime we deploy. There is also an option to view the code from any version in GAE. All the print statements provided in the application are displayed in Logs Explorer once deployed and played around in the User Interface. This turned out to be very useful while debugging. The important functionality of GAE, "Automatic Scaling" is explained in the following section "Load Balancer".

2. **Load Balancer**: This provides the automatic scaling for our application. This is one of the important features of the GAE. Since we deployed our application on Google App Engine, auto-scaling will be taken care of by the Cloud provider. Quick'ply is powered by any number of instances at any given point of time. The App Engine scheduler decides whether or not to serve a new incoming request with the existing instance or spawn a new instance. In addition, each instance has its queue for incoming requests and a new instance is created only when the queue becomes too long. Despite the dynamic instance allocation provided by GCP, we specified some parameters relevant to Quick'Ply. For instance, we mentioned the following parameters in app.yaml so that our application can have a maximum of 10 instances

   automatic_scaling:

   Maximum_instances:15

   One instance was used when the data of size 1 is uploaded, calling gmail API and displaying jobs on the UI as shown below.



7 instances were used when the data of size 7 is uploaded, calling gmail API and displaying jobs on the UI as shown below.

Novelty Of Our Solution

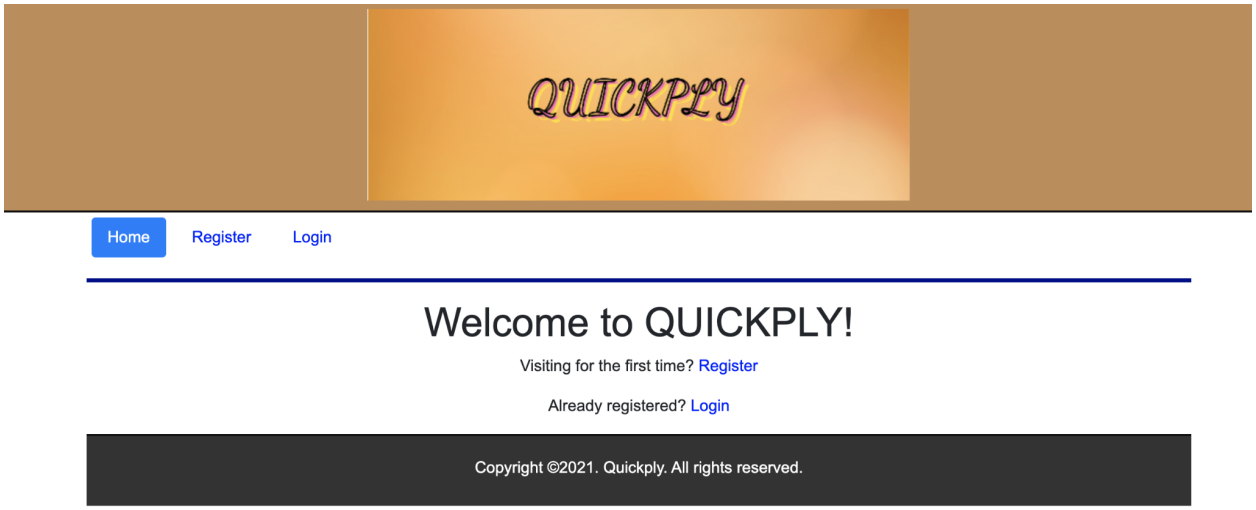
Firstly, we are sure that our solution is unique and that there is no other solution implemented and available in the market. Our application is unique in the fact that it automatically updates the statuses of every job application for every user who is using our application. All these job tracking and job status updates happen automatically for all the jobs with just one click of a button. In terms of the implementation point of view, we implemented the BERT model that automatically classifies the job related emails into one of the three categories - in_process, offered, reject. The job seekers can utilize their precious time by networking, with the hiring managers and other people, and also improving their resumes relevant to the job postings.

Step by Step explanation of how our solution solves the problem

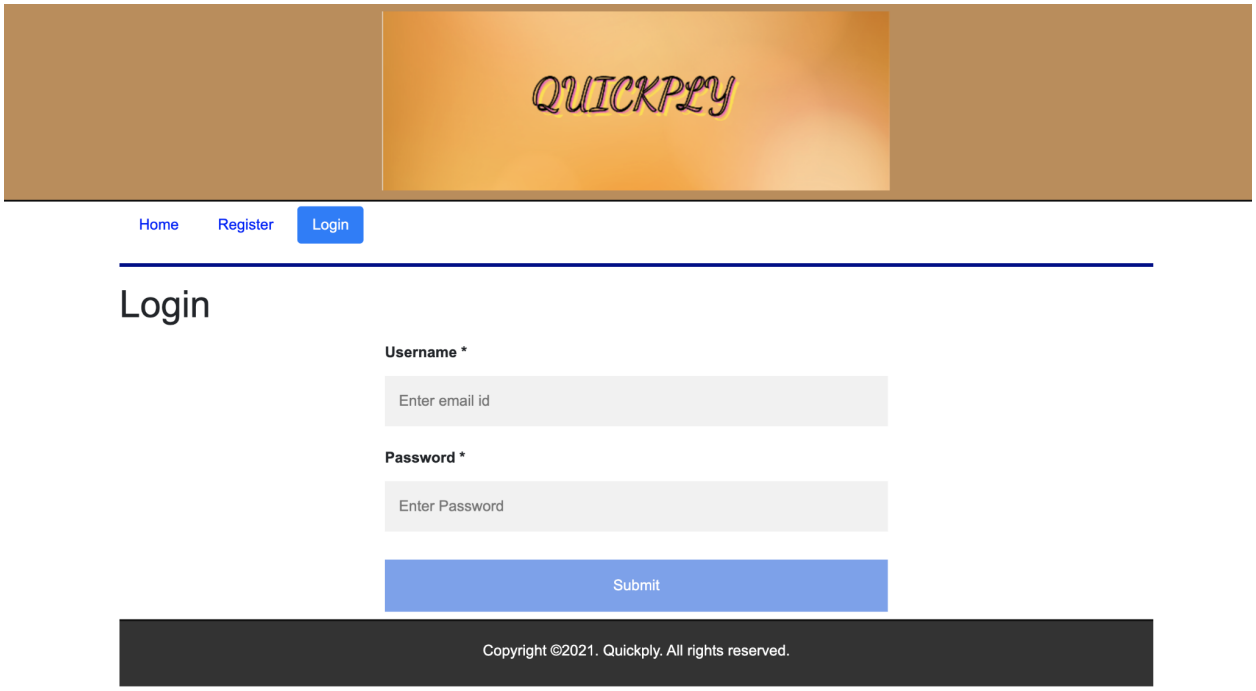
- User registration where all the details of the users are saved in our database in a confidential and safe manner
- Users can login with their login details.
- Users can upload a csv of their job statuses that they have been doing manually in order to sync with our platform
- Users can update their job table with just one click of a button. However, users should first authenticate and give us the permission to access their emails for tracking their job applications.
- Users can view their job statuses on the front end with just one click of a button
- Users can edit their personal information and update their information in the database.

Snapshots of the User Interface:

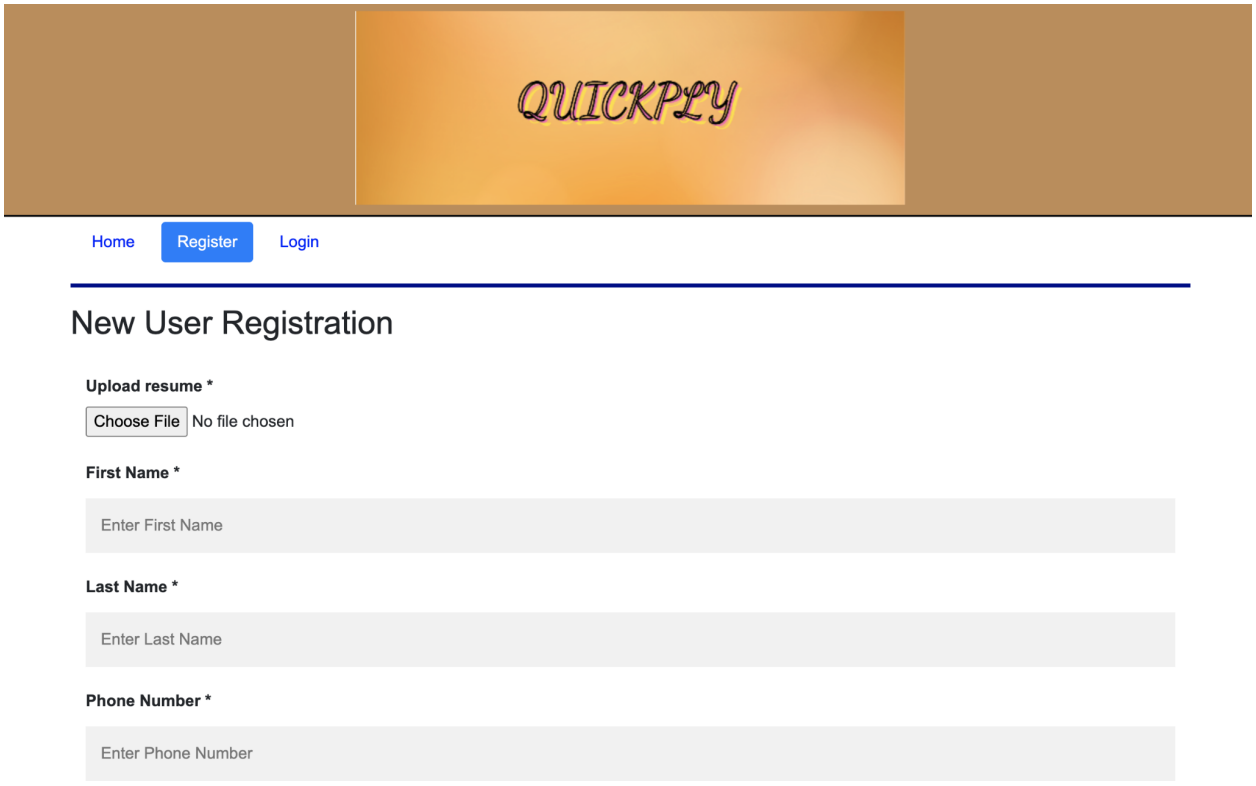
Home page:



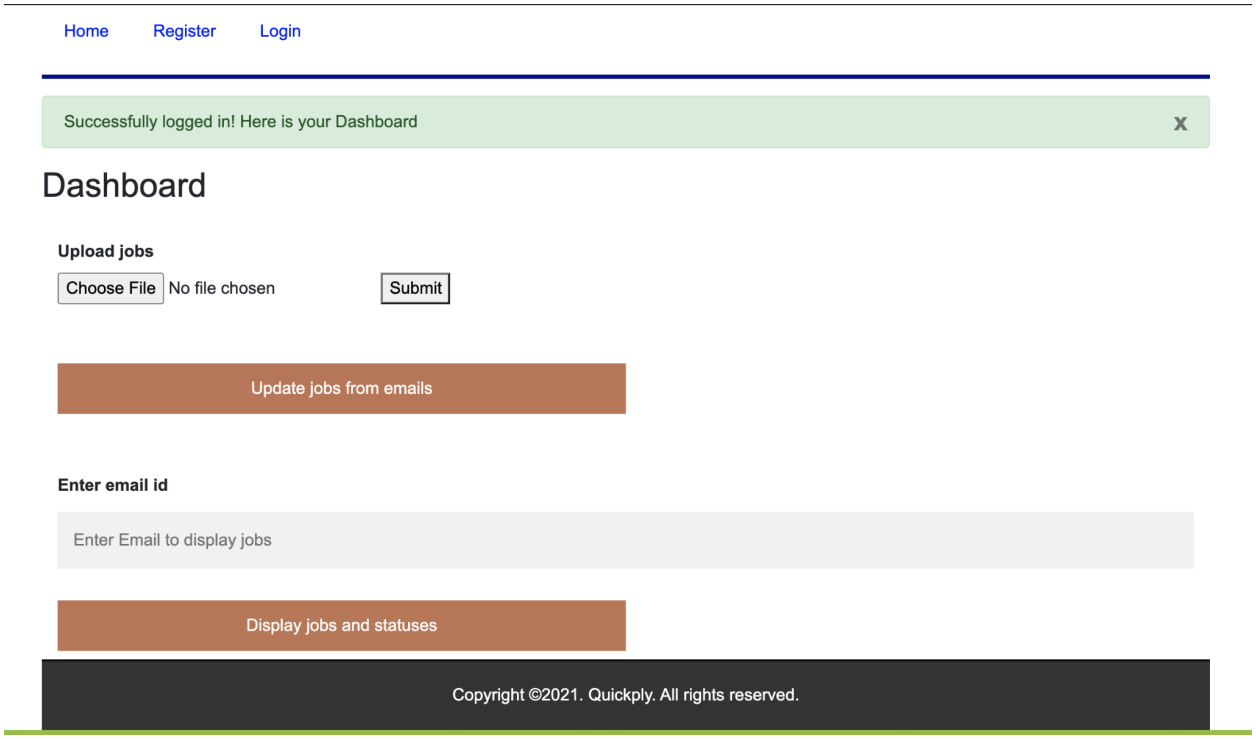
Login page:



Register page:



Dashboard page:



User table:

```
mysql> desc user;
+----------+--------------+------+-----+---------+-------+
| Field    | Type         | Null | Key | Default | Extra |
+----------+--------------+------+-----+---------+-------+
| email    | varchar(255) | NO   | MUL | NULL    |       |
| password | varchar(255) | NO   |     | NULL    |       |
+----------+--------------+------+-----+---------+-------+
2 rows in set (0.04 sec)
```

Register table:

```
mysql> desc profile;
+---------------------------------+-------------+------+-----+---------+-------+
| Field                           | Type        | Null | Key | Default | Extra |
+---------------------------------+-------------+------+-----+---------+-------+
| firstName                       | varchar(255) | NO   |     | NULL    |       |
| lastName                        | varchar(255) | NO   |     | NULL    |       |
| phone                           | varchar(31)  | NO   |     | NULL    |       |
| emailId                         | varchar(255) | NO   | PRI | NULL    |       |
| password                        | varchar(255) | NO   |     | NULL    |       |
| linkedInUrl                     | varchar(255) | YES  |     | NULL    |       |
| twitterUrl                      | varchar(255) | YES  |     | NULL    |       |
| githubUrl                       | varchar(255) | YES  |     | NULL    |       |
| portfolioUrl                    | varchar(255) | YES  |     | NULL    |       |
| otherUrl                        | varchar(255) | YES  |     | NULL    |       |
| canLegallyWorkWithoutSponsorship | tinyint(1)   | NO   |     | NULL    |       |
| needSponsorshipInFuture         | tinyint(1)   | NO   |     | NULL    |       |
| ageRange                        | varchar(13)  | YES  |     | NULL    |       |
| ethnicity                       | varchar(255) | YES  |     | NULL    |       |
| gender                          | varchar(32)  | YES  |     | NULL    |       |
| veteranStatus                   | varchar(1)   | YES  |     | NULL    |       |
| disabilityStatus                | varchar(1)   | YES  |     | NULL    |       |
| resumeUrl                       | varchar(255) | NO   |     | NULL    |       |
+---------------------------------+-------------+------+-----+---------+-------+
18 rows in set (0.04 sec)
```

Jobs table:

```
mysql> desc jobs;
+--------------+--------------+------+-----+---------+----------------+
| Field        | Type         | Null | Key | Default | Extra          |
+--------------+--------------+------+-----+---------+----------------+
| referenceId  | int(11)      | NO   | PRI | NULL    | auto_increment |
| emailId      | varchar(255) | NO   | MUL | NULL    |                |
| companyName  | varchar(255) | NO   |     | NULL    |                |
| jobId        | int(11)      | YES  |     | NULL    |                |
| jobtitle     | varchar(255) | NO   |     | NULL    |                |
| status       | varchar(12)  | NO   |     | NULL    |                |
| dateInserted | varchar(10)  | NO   |     | NULL    |                |
+--------------+--------------+------+-----+---------+----------------+
7 rows in set (0.04 sec)
```

Testing and evaluation

We tested the application thoroughly to ensure there are no failures. To make the system more durable to a large number of users, we tested the application by sending a large amount of concurrent requests.

There were different parts of our project that were tested thoroughly.

1. We registered using malicious details and performed sql injection attacks to check whether the form is taking malicious details are not. We successfully implemented all the form validations for the registration form.
2. Testing the model - our ML model was rigorously tested for better accuracy of the prediction of the intent of the email. We created a testing database with different types of emails.
3. We specified our application to be in the us-central zone and we also tested using vpn of other locations and network latency is in ideal range.
4. Though we used Google App Engine which handles instances dynamically, we specified some parameters such as max_instances, min_instances and CPU utilization.

No of requests	**Time taken (mm:ss:milliseconds)**	**No, of instances created**
1	0:36:16	1
10	1:00:86	1
100	6:04:20	7
500	14:30:50	15

As seen from the table above, the time taken to serve the requests increases as the requests increase. It is however not exponential because once the maximum number of instances are spawned (in our case, 15), the application uses the same instances to serve various requests.

Code

The structure of the frontend code in our Flask application is as follows:

.flaskenv - This sets the environment variables. FLASK_APP gives the starting point for our application which is the main.py file.

main.py - This imports the Flask application defined in __init__.py and the application is run. This is the entry point for our Flask application.

requirements.txt - This contains all the packages that the Flask application needs to download in order for the application to run.

static/images/CCproject.png - This is the images displayed at the top in the User Interface.

templates -

1. layout.html - This is the main html file that has the entire html code available or called. The nav.html and footer.html are called in this. The block content inside this layout.html will change based on the various functionalities as described in [2-5] below. This also has the code to display the flash messages.
2. index.html - This is the code for the home page.
3. login.html - This page opens up when the user clicks on login in the navigation bar or from the home page.
4. register.html: This page opens up when the user clicks on Register in the navigation bar or from the home page.
5. display.html: This page opens up when the user clicks on 'Display jobs and statuses' from the Dashboard page and it displays all the jobs of the user from the jobs table.
6. Includes:
    a. nav.html: This page has the code for displaying the Navigation bar which has Home, Login and Register as the options.
    b. footer.html: This page has the code for displaying the footer of the User Interface which mentions about the Copyrights of our application.

css/main.css - This has the styling for our entire application.

The structure of our backend code is in MVC architecture is as follows:

routes.py - This file is one of the most important files in our application. This file contains the API routes.

1. /login - to let the user login into our application.
2. /dashboard - to let the user visit the dashboard. The dashboard consists of all the details of the user.
3. /gmail, /callauth, /authorize, /oauth2callback - to authenticate the user to help with the authentication
4. /registersubmit - to register the user.
5. /fileupload - If the user wants to update the list of the jobs that they have applied to, they can upload a csv file with the jobs in it. This would update the table with the list of jobs in the file.

controller.py - This file contains the logic for filtering the emails that we get after the authentication using Gmail API. Based on the email Id that we received from Gmail authentication, we make a call to our SQL instance to get the list of jobs applied by this email Id. The list of companies that the user has applied for is created using the list of jobs returned by the SQL instance. The emails are filtered using a list of companies. The filtered emails are parsed and decoded. The body of these emails are sent to our

BERT model. Once the model has given a prediction of what the intent of the email is, we update it in our Jobs table in our database present in our SQL instance.

1. ListMessagesMatchingQuery - get the emails from the service once authenticated. It iterates over the messages, decodes it and parses through to get the body of the message.
2. filterEmails - Once the Gmail authentication is completed, the messages are read from the gmail. getCompanyNames() is called to get the list of companies the user has applied for. The list of companies is used to filter the emails and is sent to the Bert model.
3. getCompanyNames - returns the list of company names from the list of jobs passed as the parameter of the method.
4. updateJobs - updates the status for each job object provided by the Bert model. After this, these job objects are used to update the rows in the Jobs table.
5. applyBert - Once the emails are filtered, this method is called. It calls the bert model with the messages and returns the predictions of the messages.

db.py - All the interactions with the database are handled by this python file. SQLAlchemy package is used for the database connection. Added to that, we created ORM for our database interactions. Object-relational-mapping is the idea of being able to write queries like the one above, as well as much more complicated ones, using the object-oriented paradigm of your preferred programming language. This helped us simplify our complex search and update queries. There are three models created; one for each table: User, Profile, and Jobs.

1. checkUser - Checks if the user is present in the User table in the database.
2. Register - Add a new user in the User and Profile table in the database.
3. jobApplied - Adds a new job in the Jobs table based on the email Id provided.
4. updateJobStatus - updates the status of a job for the email Id provided.
5. getJobsByEmail - gets the list of jobs present in the Jobs table after filtering it by email Id.
6. getAllJobsByEmailAndStatusForGmail - gets the list of jobs based on the email Id and status.
7. updateProfile - updates the row in the Profile table based on the email Id provided.
8. getProfileByEmail - gets the profile details from the Profile table based on the email Id.

app.yaml - The configurations for our app engine are present here. The autoscaling on our app engine is enabled here.

bert.py - We implemented bert.py which is a machine learning model in order to classify the emails into one of the three categories - in_process, reject, offered. We used a BERT-based model that has 110 million parameters. For training purposes, we froze all the layers of the model before fine-tuning it. For optimization purposes, we used AdamW optimizer which is an improved version of the Adam optimizer. There is a class imbalance in our dataset as we have more rejects than offered emails. So, we will first compute the class weights for the labels in the train set and then pass these weights to the loss function so that it takes care of the class imbalance. Following this, we trained our dataset on the model after fine tuning it by implementing the functions for it. On running bertemailclassification.py, we get saved_weights.pt which is a pickled version of this trained model. We then load saved_weights.pt file in our bert.py to classify test emails.

Program installations:

For running our application, just have us start SQL instance (It is stopped keeping free tier limit in mind) and open https://enhanced-rarity-312105.uc.r.appspot.com/

To deploy our application manually, do the following:

1. Unzip the Quickply folder. This has all the code for our application.
2. Run bertemailclassification.py to generate the trained model (pickled), bert_model.pt and have it inside the quickply/application/static/bert.
3. Install Cloud SDK from https://cloud.google.com/sdk/docs/quickstart
4. 'cd quickply'
5. Give 'gcloud auth login' and authenticate the gmail where you would like to deploy your code.
6. Change to your specific project in GCP using 'gcloud config set project PROJECT_NAME'
7. You can create your own SQL instance in the google cloud account and have the tables created. To connect to this, change line 10 of db.py to your specific Cloud SQL instance.
8. Finally, give 'gcloud app deploy'

This application does not run in a local environment as we need Cloud SQL Auth Proxy for SQL Alchemy (used to connect to Cloud SQL) to work. Our focus was to host the application, have a proper endpoint and have it run in all environments, so we checked and tested directly after deploying.

Conclusion

There were a lot of accomplishments throughout this project. Following are our accomplishments :-

1. Login and register functionalities were successfully implemented.
2. Popular Natural Language Processing BERT model was successfully implemented and tested.
3. Integration of Flask and various components in Google Cloud Platform like Cloud SQL and Google App Engine was done successfully.
4. We could successfully deploy our application on Google App Engine and have it run on various environments.
5. Gmail authentication to have our application access user's emails to give text of the filtered emails by company names to BERT model, so we could have job statuses of the user updated as well implemented.
6. Finally, the project was successfully completed as mentioned in our proposal with thorough testing.

To start with the learnings, this was the first encounter with Google Cloud Platform for all three of us, so it was a great learning experience to implement all the above mentioned functionalities. To highlight a few,

1. Learnt Flask web framework and implemented it.
2. Learnt to integrate various front end technologies like html, css and javascript using Flask web framework.
3. In the backend, we learnt how to have the various GCP applications communicate with each other.
4. We also learnt how to integrate Flask with GCP.
5. We learnt ways to debug when we faced errors and also how to effectively utilize the Google Cloud documentations and websites like stackoverflow everytime we got stuck on something.
6. Efficient utilization of Logs Explorer in GCP was learnt.
7. Finally, how to co-ordinate and communicate with team members to successfully complete the project in an Agile environment, tracking the frequent progress using trello.

Currently, our application helps job seekers track and update their job applications with just one click of a button. It currently serves its features via a Web Interface. In the future, we want to provide the following functionalities

1. Implement a feature where the users can automatically apply to jobs by just providing our application with the URLs of the jobs that they are interested in applying to.
2. Mobile app version of the application that we developed so that the users can update and track their job statuses via their cell phones.
3. Implement a chrome extension that lets users to update their job statuses by clicking on the chrome extension.
4. Implement a recommendation system to recommend jobs to students by analyzing their job related emails. Upon analyzing their job related emails, we understand the kind of jobs that they are applying for and we recommend a similar set of jobs.

References

[1]https://towardsdatascience.com/i-built-a-reject-not-reject-email-classifier-for-my-job-applications-844a3b6cd67e

[2] https://docs.sqlalchemy.org/en/13/orm/

[3]https://github.com/GoogleCloudPlatform/python-docs-samples

[4]https://developers.google.com/gmail/api/guides

[5]https://towardsdatascience.com/bert-for-dummies-step-by-step-tutorial-fb90890ffe03

[6]https://stackoverflow.com/questions/48426239/working-with-gmail-api-from-google-appengine