Cloud Application Development midterm

Student: Islam Aip
Faculty: School of Information Technology and Engineering
Specialty: Computer Systems and Software

1.Executive Summary

In this project we use all theoretical knowledge that we obtained during the lecture and use them to build and deploy an application. Our application for this is Flask to-do app that adds and displays tasks. We also containerized our application, integrate Google Cloud Functions, configure APIs, set up testing and monitoring. We used services provided by GCP such as, Google Cloud Endpoints, Google Kubernetes Engine, Google Cloud Functions and also we used Docker to containerize our app.

2.Introduction

Google Cloud Platform – modern cloud computing platform provided by Google that uses the same infrastructure that Google uses for its services. Google Cloud Platform allows developers to build, deploy and scale applications efficiently. Products of Google Cloud Platform such as, App Engine, Kubernetes Engine, Cloud Functions and others provide flexibility and cover many use-cases.

Choosing cloud infrastructure like GCP is motivated by several factors. Firstly, developers do not have to focus on managing physical servers. Secondly, pay-as-you-go model allows companies or other organizations achieve cost-efficiency. Thirdly, Google Cloud Platform offers a vast range of integrated tools and services.

3.Project Objectives

Main objectives:

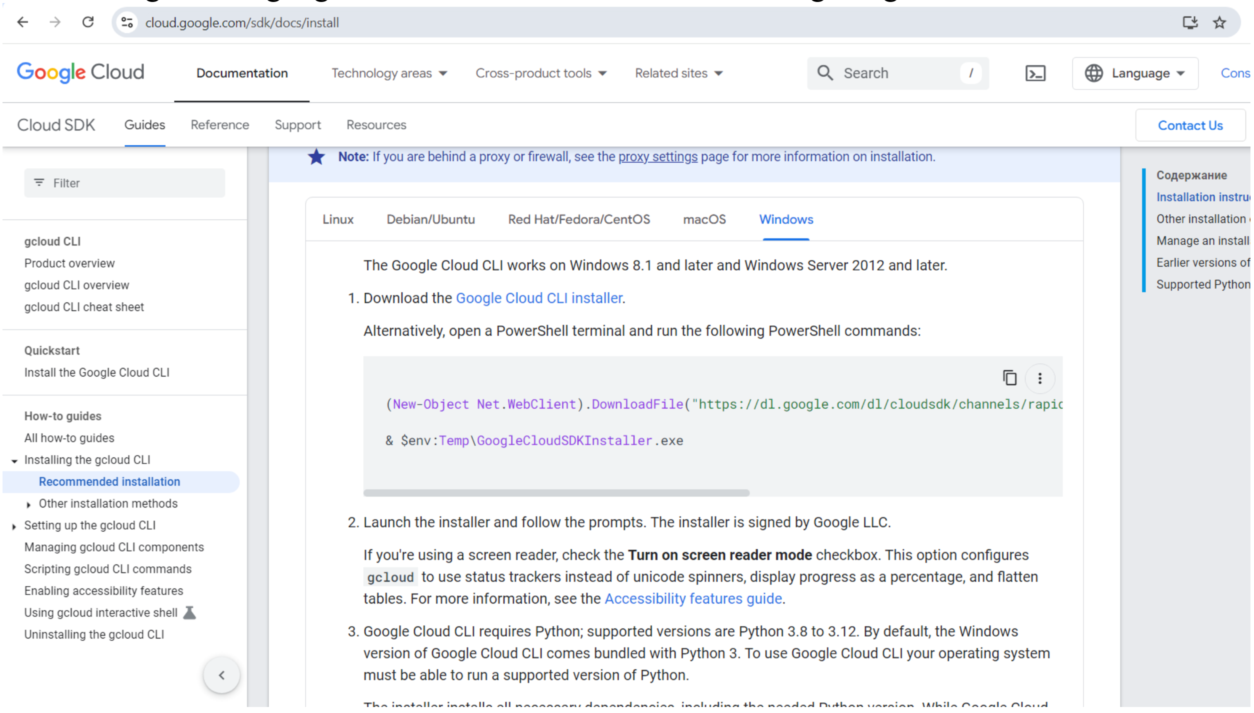
- Developing and deploying application on the cloud
- Integrating Google Cloud Functions into our application
- Containerizing this application
- Managing APIs of our application
- Setting up testing and monitoring for our application

4.Google Cloud Platform Overview

GCP is fast platform offering many useful services. GCP's architecture is built on a global network of data centers.

5.Google Cloud SDK and Cloud Shell

We should go to the google cloud website and find the Installing the gcloud CLI section.



Note: If you are behind a proxy or firewall, see the proxy settings page for more information on installation.

The Google Cloud CLI works on Windows 8.1 and later and Windows Server 2012 and later.

1. Download the Google Cloud CLI installer.

   Alternatively, open a PowerShell terminal and run the following PowerShell commands:

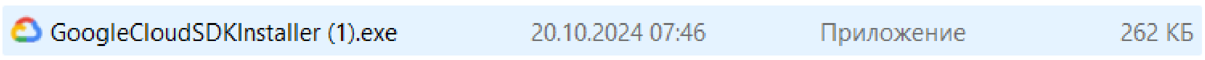
   ```
   (New-Object Net.WebClient).DownloadFile("https://dl.google.com/dl/cloudsdk/channels/rapic

   & $env:Temp\GoogleCloudSDKInstaller.exe
   ```

2. Launch the installer and follow the prompts. The installer is signed by Google LLC.

   If you're using a screen reader, check the **Turn on screen reader mode** checkbox. This option configures `gcloud` to use status trackers instead of unicode spinners, display progress as a percentage, and flatten tables. For more information, see the Accessibility features guide.

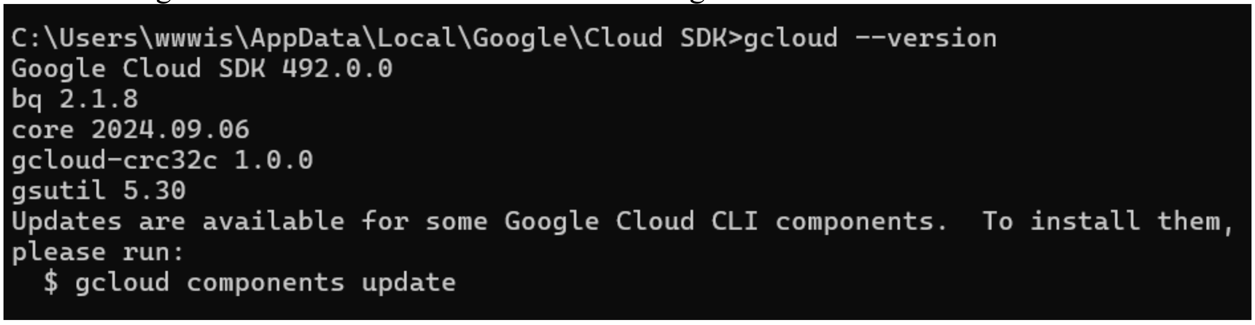
3. Google Cloud CLI requires Python; supported versions are Python 3.8 to 3.12. By default, the Windows version of Google Cloud CLI comes bundled with Python 3. To use Google Cloud CLI your operating system must be able to run a supported version of Python.

We should choose our OS and download the Google Cloud CLI installer. After, we should open the downloaded file:


GoogleCloudSDKInstaller (1).exe 20.10.2024 07:46 Приложение 262 КБ

After opening we can just follow instructions or configure what components we want to install. Google SDK should appear in a Desktop:

We can run gcloud –version in console to ensure that gcloud SDK shell is installed:

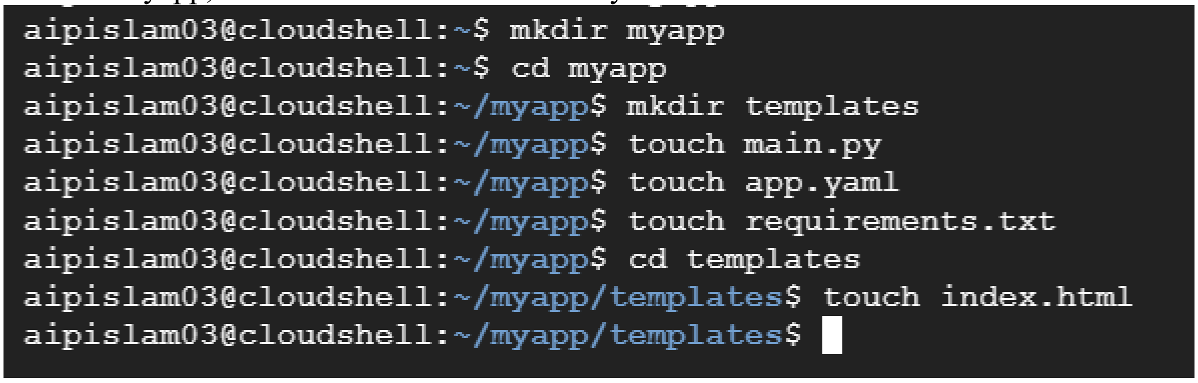


```
C:\Users\wwwis\AppData\Local\Google\Cloud SDK>gcloud --version
Google Cloud SDK 492.0.0
bq 2.1.8
core 2024.09.06
gcloud-crc32c 1.0.0
gsutil 5.30
Updates are available for some Google Cloud CLI components.  To install them,
please run:
  $ gcloud components update
```

We can also use shell to authorize, create and delete projects, deploy apps and so on.

6.Google App Engine

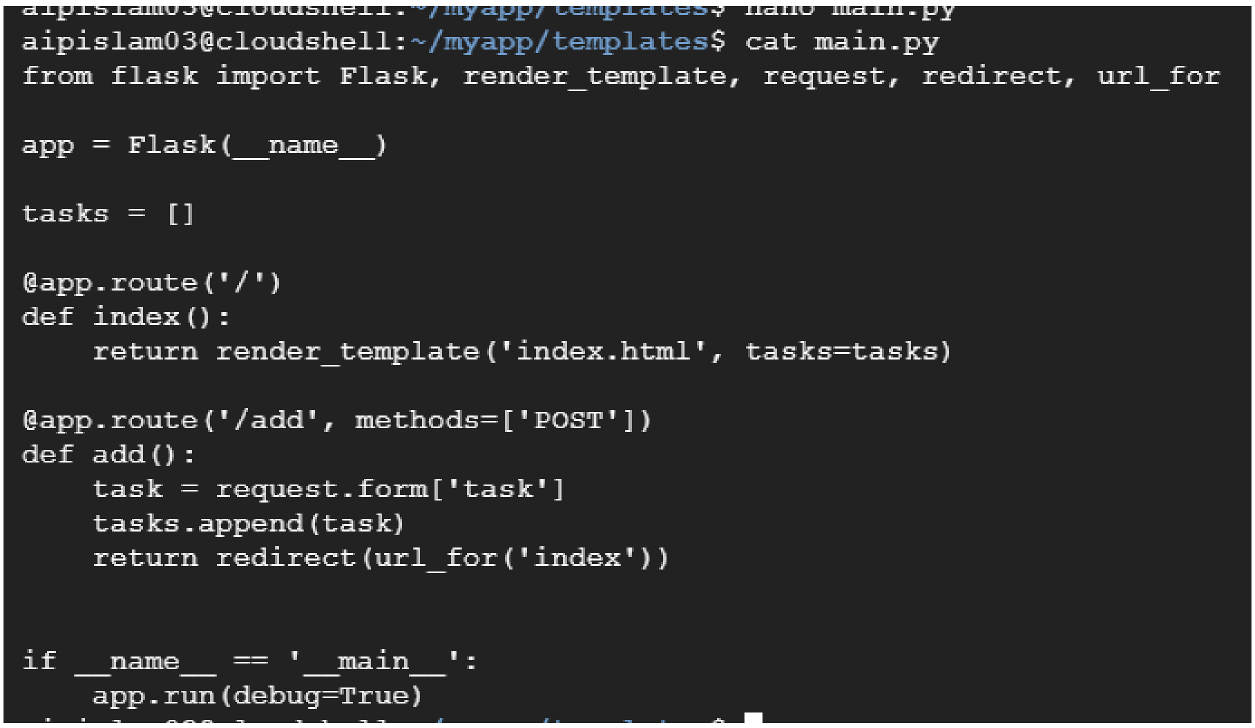
Now, let's develop a simple app and deploy it on the cloud. For this module we can develop a simple To-do list app. Flask will be our framework, and application will just add tasks and list them on the main page. Let's set up our simple app, firstly we create folder for our app let's call it myapp, then we create all the necessary files and folders.



```
aipislam03@cloudshell:~$ mkdir myapp
aipislam03@cloudshell:~$ cd myapp
aipislam03@cloudshell:~/myapp$ mkdir templates
aipislam03@cloudshell:~/myapp$ touch main.py
aipislam03@cloudshell:~/myapp$ touch app.yaml
aipislam03@cloudshell:~/myapp$ touch requirements.txt
aipislam03@cloudshell:~/myapp$ cd templates
aipislam03@cloudshell:~/myapp/templates$ touch index.html
aipislam03@cloudshell:~/myapp/templates$
```

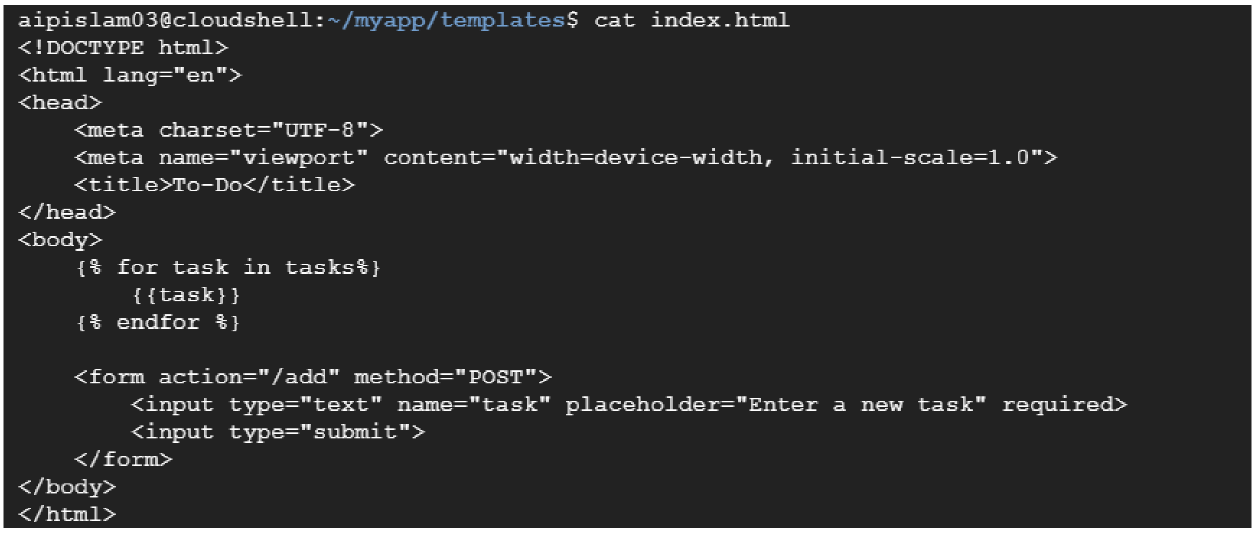
- main.py – Flask app code
- requirements.txt – the file that lists dependencies of our app
- app.yaml – configuration file for our app
- templates - folder where we store our applications templates(html files in our case)

After setting up our app's directory structure, we start by writing main logic of our app.



```
aipislam03@cloudshell:~/myapp/templates$ nano main.py
aipislam03@cloudshell:~/myapp/templates$ cat main.py
from flask import Flask, render_template, request, redirect, url_for

app = Flask(__name__)

tasks = []

@app.route('/')
def index():
    return render_template('index.html', tasks=tasks)

@app.route('/add', methods=['POST'])
def add():
    task = request.form['task']
    tasks.append(task)
    return redirect(url_for('index'))


if __name__ == '__main__':
    app.run(debug=True)
```

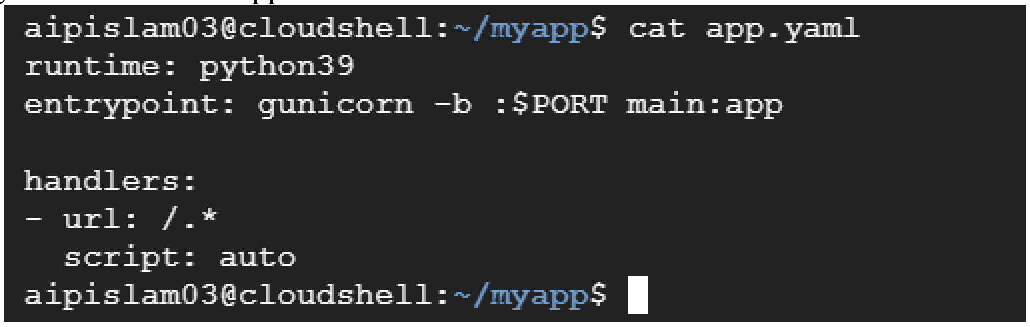
Contens of main.py file. Simple Flask app to add and list tasks



```
aipislam03@cloudshell:~/myapp/templates$ cat index.html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>To-Do</title>
</head>
<body>
    {% for task in tasks%}
        {{task}}
    {% endfor %}

    <form action="/add" method="POST">
        <input type="text" name="task" placeholder="Enter a new task" required>
        <input type="submit">
    </form>
</body>
</html>
```

Contents of index.html file. It will display our tasks and add new ones.

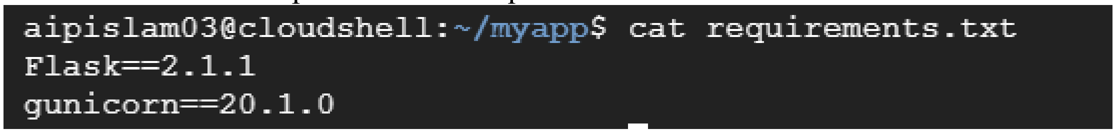
After writing our apps main logic, we should write configuration file. app.yaml will define how App Engine will serve our app.



```
aipislam03@cloudshell:~/myapp$ cat app.yaml
runtime: python39
entrypoint: gunicorn -b :$PORT main:app

handlers:
- url: /.*
  script: auto
aipislam03@cloudshell:~/myapp$
```

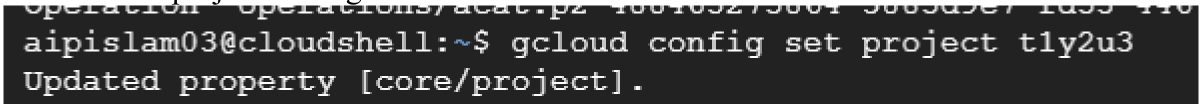
- runtime – runtime environment, Python 3.9 in our case
- entrypoint – specifies how to start our app, in our case gunicorn. $PORT will be dynamically allocated by App Engine and we don’t need to manually define it.
- Handlers – control which URLs are used. Here it uses all URLs in our app

Now let’s write our dependencies in requirements.txt:



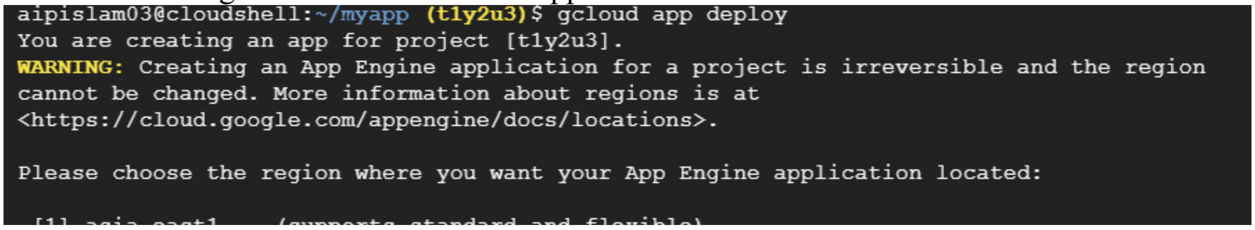
```
aipislam03@cloudshell:~/myapp$ cat requirements.txt
Flask==2.1.1
gunicorn==20.1.0
```

Let’s set our projects configuration:



```
aipislam03@cloudshell:~$ gcloud config set project t1y2u3
Updated property [core/project].
```

Now we can run gcloud app deploy command in our console. After running this command we should choose region where we want our app to be located:



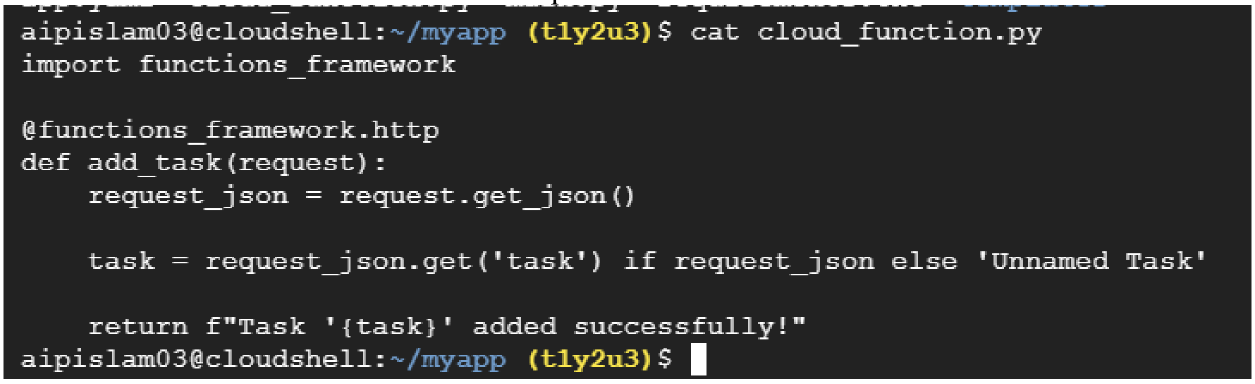
```
aipislam03@cloudshell:~/myapp (t1y2u3)$ gcloud app deploy
You are creating an app for project [t1y2u3].
WARNING: Creating an App Engine application for a project is irreversible and the region
cannot be changed. More information about regions is at
<https://cloud.google.com/appengine/docs/locations>.

Please choose the region where you want your App Engine application located:
```

For this task to be completed, billing account should be enabled.

After choosing our region, our app will be deployed and we will be able to access it through given URL.

7.Building with Google Cloud Functions

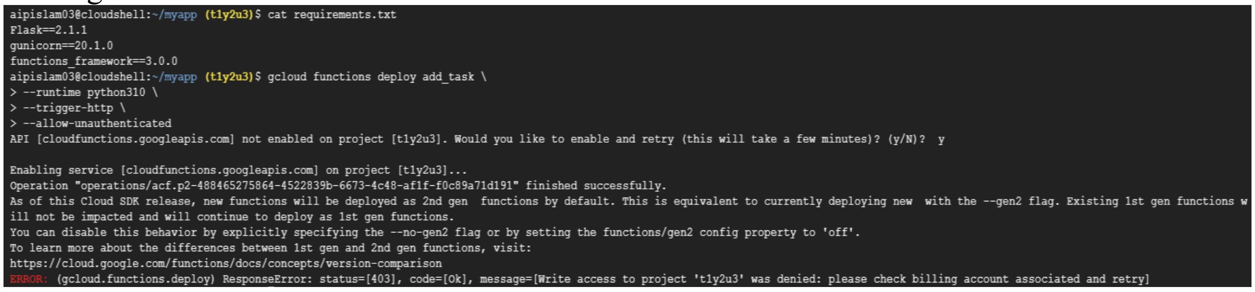
Google Cloud Functions are serverless serverless execution environments, for building functions that will be triggered when event is fired, or by HTTP request.
Let’s write our function that will return response when task is created.



```
aipislam03@cloudshell:~/myapp (t1y2u3)$ cat cloud_function.py
import functions_framework

@functions_framework.http
def add_task(request):
    request_json = request.get_json()

    task = request_json.get('task') if request_json else 'Unnamed Task'

    return f"Task '{task}' added successfully!"
aipislam03@cloudshell:~/myapp (t1y2u3)$ 
```

This function is triggered via HTTP request. It extracts task information from request and returns response with tasks name.

Let’s add functions_framework==3.0.0 to our requirements.txt file and deploy our function by running this command:

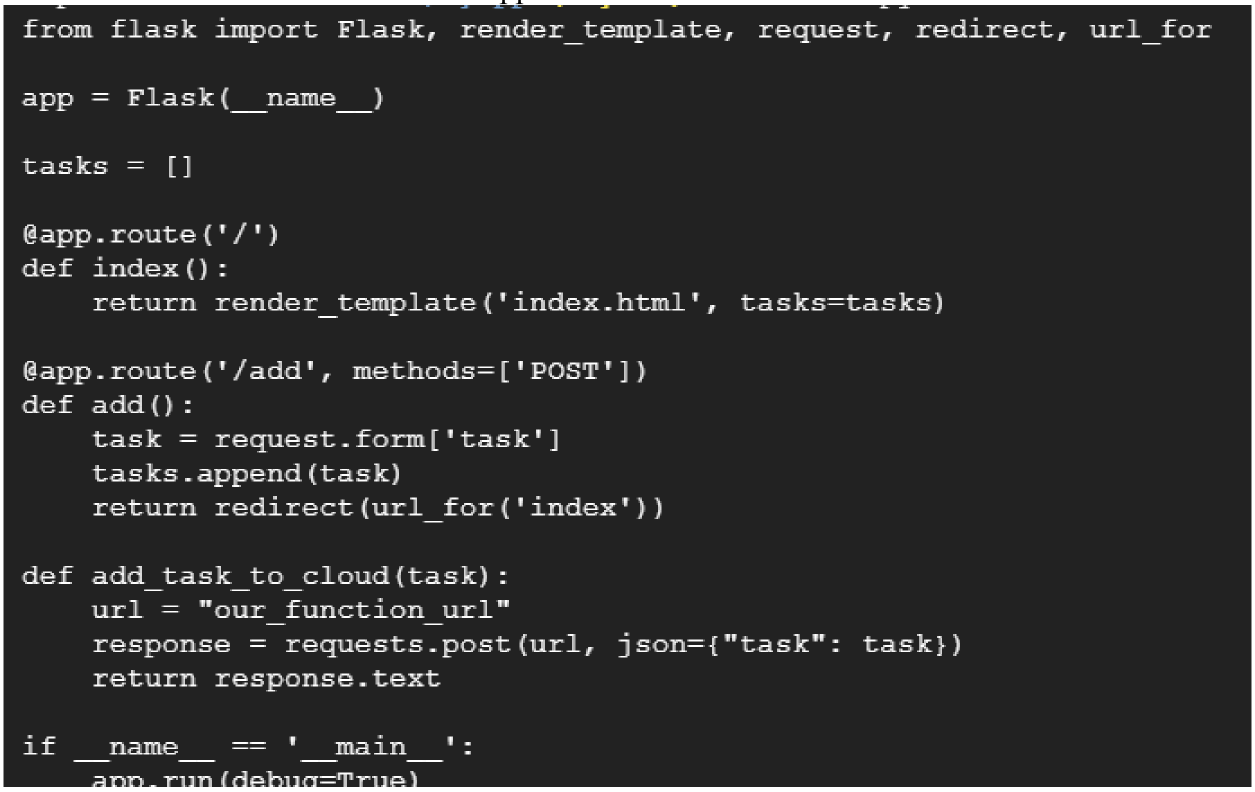


```
aipislam03@cloudshell:~/myapp (t1y2u3)$ cat requirements.txt
Flask==2.1.1
gunicorn==20.1.0
functions_framework==3.0.0
aipislam03@cloudshell:~/myapp (t1y2u3)$ gcloud functions deploy add_task \
> --runtime python310 \
> --trigger-http \
> --allow-unauthenticated
API [cloudfunctions.googleapis.com] not enabled on project [t1y2u3]. Would you like to enable and retry (this will take a few minutes)? (y/N)?  y

Enabling service [cloudfunctions.googleapis.com] on project [t1y2u3]...
Operation "operations/acf.p2-488465275864-4522839b-6673-4c48-af1f-f0c89a71d191" finished successfully.
As of this Cloud SDK release, new functions will be deployed as 2nd gen  functions by default. This is equivalent to currently deploying new  with the --gen2 flag. Existing 1st gen functions w
ill not be impacted and will continue to deploy as 1st gen functions.
You can disable this behavior by explicitly specifying the --no-gen2 flag or by setting the functions/gen2 config property to 'off'.
To learn more about the differences between 1st gen and 2nd gen functions, visit:
https://cloud.google.com/functions/docs/concepts/version-comparison
ERROR: (gcloud.functions.deploy) ResponseError: status=[403], code=[Ok], message=[Write access to project 't1y2u3' was denied: please check billing account associated and retry]
```

- add_task – name of our function
- runtime – specifies the python version
- --trigger-http – specifies that the function is triggered by HTTP request

- --allow-unauthenticated – allows anyone to access to the function without authentication

Deploying function also require billing account. After running this command, we will be given URL where the function is hosted.
We can use this function in our Flask app:

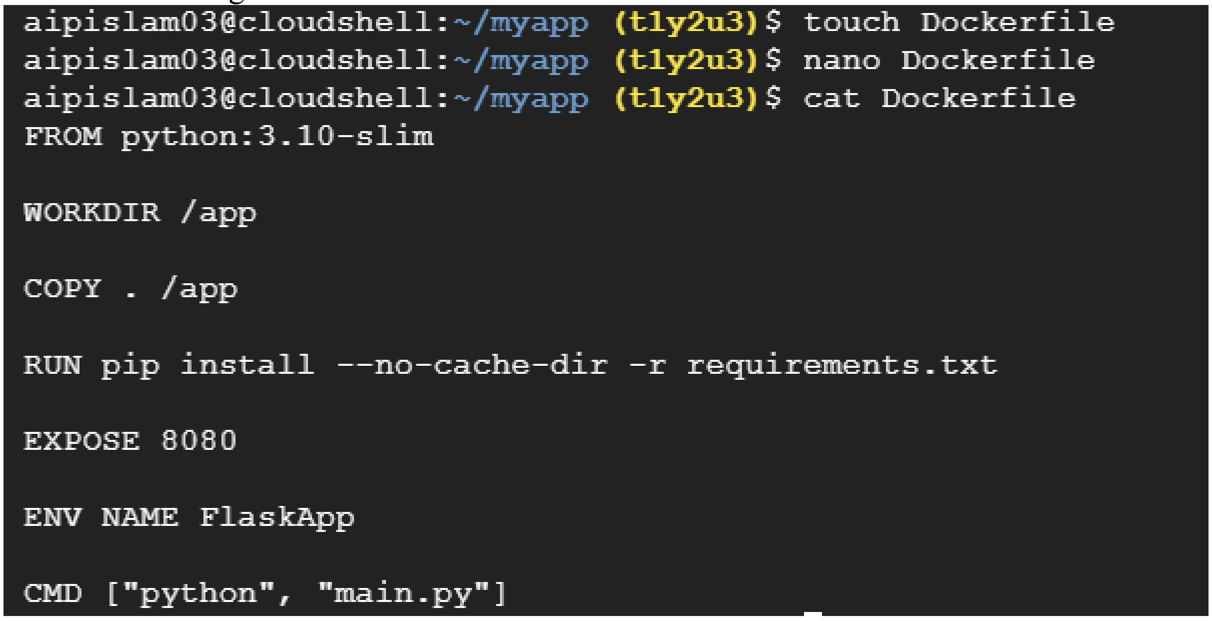


```
from flask import Flask, render_template, request, redirect, url_for

app = Flask(__name__)

tasks = []

@app.route('/')
def index():
    return render_template('index.html', tasks=tasks)

@app.route('/add', methods=['POST'])
def add():
    task = request.form['task']
    tasks.append(task)
    return redirect(url_for('index'))

def add_task_to_cloud(task):
    url = "our_function_url"
    response = requests.post(url, json={"task": task})
    return response.text

if __name__ == '__main__':
    app.run(debug=True)
```

Here we trigger our function by sending HTTP request

8.Containerizing Applications

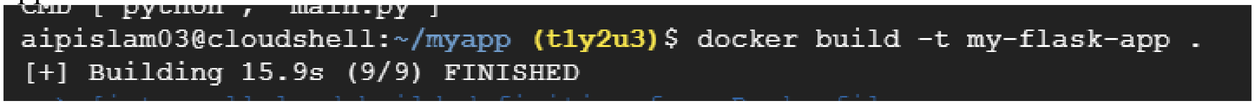
By containerizing our application and its dependencies into containers, we ensure that our app will run consistently in different environments. Containerization also speeds up deployment.
The first step in containerizing our application is creating a Dockefile. Dockerfile defines how our container image should be build.



```
aipislam03@cloudshell:~/myapp (t1y2u3)$ touch Dockerfile
aipislam03@cloudshell:~/myapp (t1y2u3)$ nano Dockerfile
aipislam03@cloudshell:~/myapp (t1y2u3)$ cat Dockerfile
FROM python:3.10-slim

WORKDIR /app

COPY . /app

RUN pip install --no-cache-dir -r requirements.txt

EXPOSE 8080

ENV NAME FlaskApp

CMD ["python", "main.py"]
```

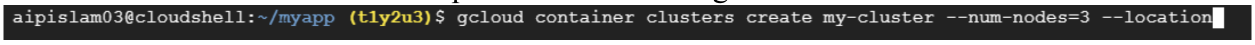
We use python 3.10 as our base image. Then we set the working directory and copy our application code into the container. After that we install dependencies that we specified in requirements.txt and run our application.

After writing Dockerfile for our app we can build the image by running docker build -t my-flask-app . in our console:



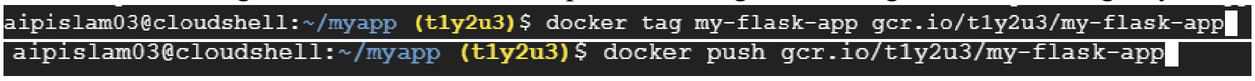
```
aipislam03@cloudshell:~/myapp (t1y2u3)$ docker build -t my-flask-app .
[+] Building 15.9s (9/9) FINISHED
```

Now we need to deploy our containerized application. We can use Google Kubernetes Engine for that.
We should create GKE cluster and push our docker image:



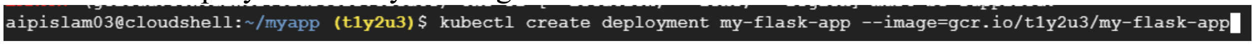
```
aipislam03@cloudshell:~/myapp (t1y2u3)$ gcloud container clusters create my-cluster --num-nodes=3 --location
```

Billing account is required to create GKE cluster and we should specify location, zone, or region.
After creating GKE cluster we should push our image into Google Container Registry.

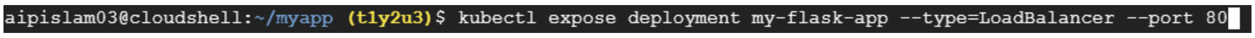


```
aipislam03@cloudshell:~/myapp (t1y2u3)$ docker tag my-flask-app gcr.io/t1y2u3/my-flask-app
aipislam03@cloudshell:~/myapp (t1y2u3)$ docker push gcr.io/t1y2u3/my-flask-app
```

Now we can deploy to GKE by running this commands:



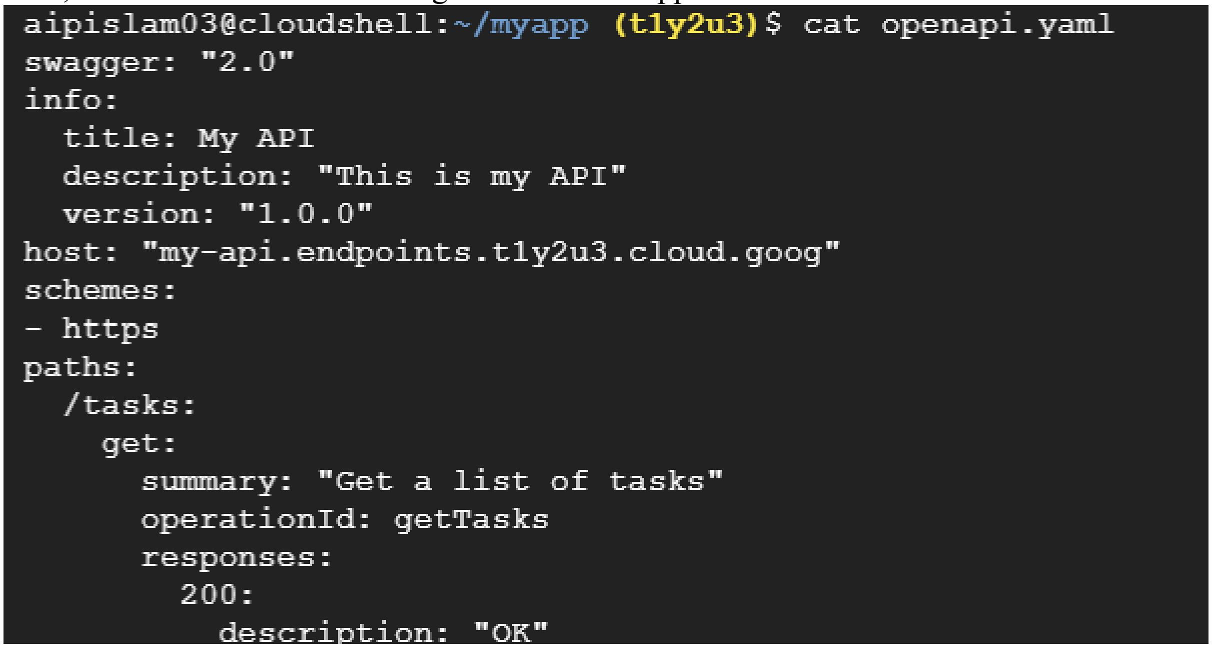
```
aipislam03@cloudshell:~/myapp (t1y2u3)$ kubectl create deployment my-flask-app --image=gcr.io/t1y2u3/my-flask-app
```



```
aipislam03@cloudshell:~/myapp (t1y2u3)$ kubectl expose deployment my-flask-app --type=LoadBalancer --port 80
```

9.Managing APIs with Google Cloud Endpoints

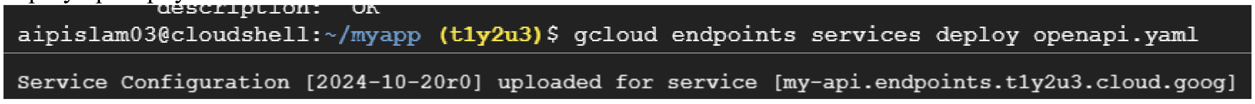
Google Cloud Endpoints allows us to create, deploy and manage APIs of our application. We can integrate Google Cloud Endpoints into our application by creating and deploying our API configuration.
First, we need to create API configuration for our app.



```
aipislam03@cloudshell:~/myapp (t1y2u3)$ cat openapi.yaml
swagger: "2.0"
info:
  title: My API
  description: "This is my API"
  version: "1.0.0"
host: "my-api.endpoints.t1y2u3.cloud.goog"
schemes:
- https
paths:
  /tasks:
    get:
      summary: "Get a list of tasks"
      operationId: getTasks
      responses:
        200:
          description: "OK"
```

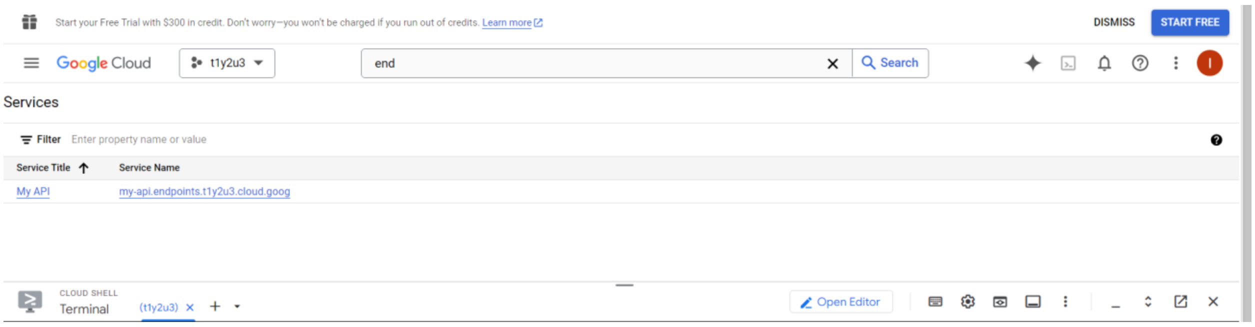
- swagger – tells that the document is written is swagger format
- info – provides general information about API(title, description, version in our case)
- host – specifies URL where API is hosted
- schemes – protocol used by API
- paths – API endpoints
- methods and responses – specifies that in path "/tasks" we have get request that should return "OK" when this request is successfully made.

After we setting up our API configuration we can deploy it by running gcloud endpoints services deploy openapi.yaml in our console.



```
aipislam03@cloudshell:~/myapp (t1y2u3)$ gcloud endpoints services deploy openapi.yaml

Service Configuration [2024-10-20r0] uploaded for service [my-api.endpoints.t1y2u3.cloud.goog]
```

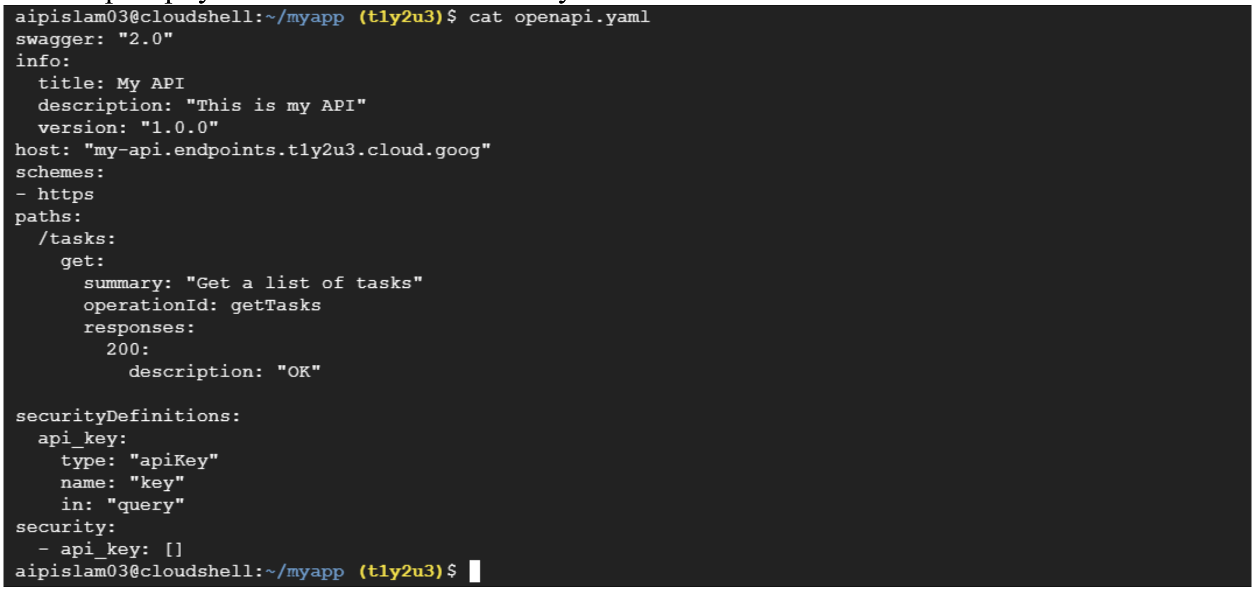
We can manage our api in endpoints section.


Start your Free Trial with $300 in credit. Don't worry—you won't be charged if you run out of credits. Learn more

Services

Service Title	Service Name
My API	my-api.endpoints.t1y2u3.cloud.goog

Implementation of authentication:
In our openapi.yaml file we can add securityDefinitions section to describe authentication.


```
aipislam03@cloudshell:~/myapp (t1y2u3)$ cat openapi.yaml
swagger: "2.0"
info:
  title: My API
  description: "This is my API"
  version: "1.0.0"
host: "my-api.endpoints.t1y2u3.cloud.goog"
schemes:
- https
paths:
  /tasks:
    get:
      summary: "Get a list of tasks"
      operationId: getTasks
      responses:
        200:
          description: "OK"

securityDefinitions:
  api_key:
    type: "apiKey"
    name: "key"
    in: "query"
security:
  - api_key: []
aipislam03@cloudshell:~/myapp (t1y2u3)$
```

Instead of [] there should be real api key.

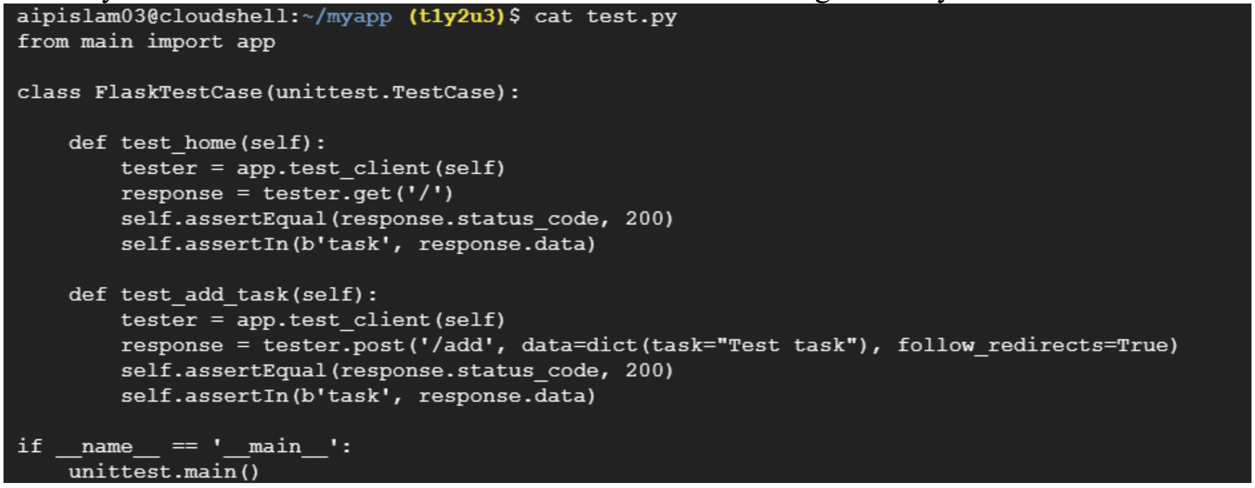
For monitoring our API traffic we can use Endpoints section in Google Cloud. It provides detailed information about Api traffic, errors and more.

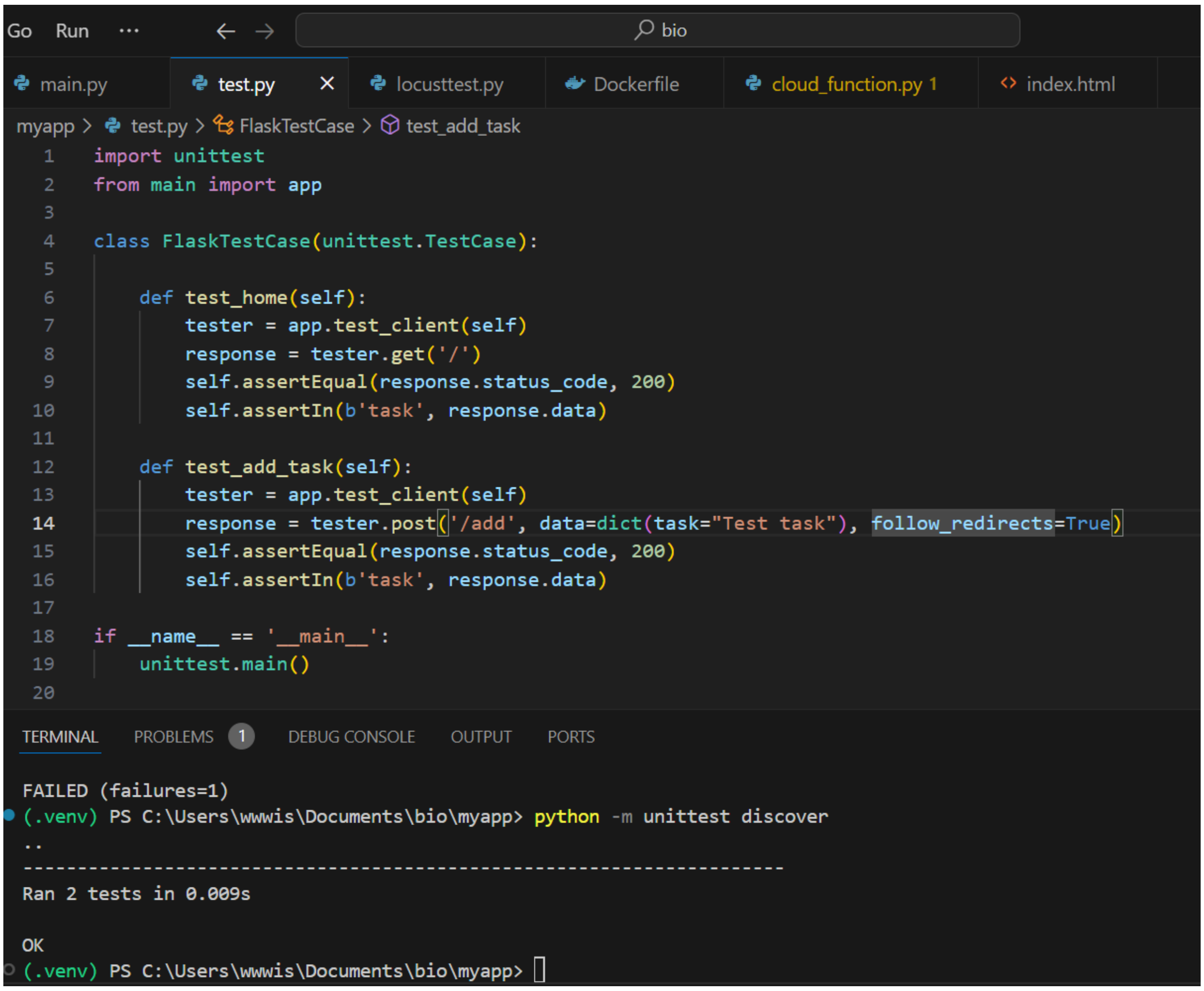
10.Testing and Quality Assurance

Testing is an important part of application development. It ensures that every part of application is working correctly.

Unit testing

Unit testing is often used to test functions and just to ensure that applications functions behave correctly. We can test if our index and add functions are working correctly:


```
aipislam03@cloudshell:~/myapp (t1y2u3)$ cat test.py
from main import app

class FlaskTestCase(unittest.TestCase):

    def test_home(self):
        tester = app.test_client(self)
        response = tester.get('/')
        self.assertEqual(response.status_code, 200)
        self.assertIn(b'task', response.data)

    def test_add_task(self):
        tester = app.test_client(self)
        response = tester.post('/add', data=dict(task="Test task"), follow_redirects=True)
        self.assertEqual(response.status_code, 200)
        self.assertIn(b'task', response.data)

if __name__ == '__main__':
    unittest.main()
```



```python
import unittest
from main import app

class FlaskTestCase(unittest.TestCase):

    def test_home(self):
        tester = app.test_client(self)
        response = tester.get('/')
        self.assertEqual(response.status_code, 200)
        self.assertIn(b'task', response.data)

    def test_add_task(self):
        tester = app.test_client(self)
        response = tester.post('/add', data=dict(task="Test task"), follow_redirects=True)
        self.assertEqual(response.status_code, 200)
        self.assertIn(b'task', response.data)

if __name__ == '__main__':
    unittest.main()

```

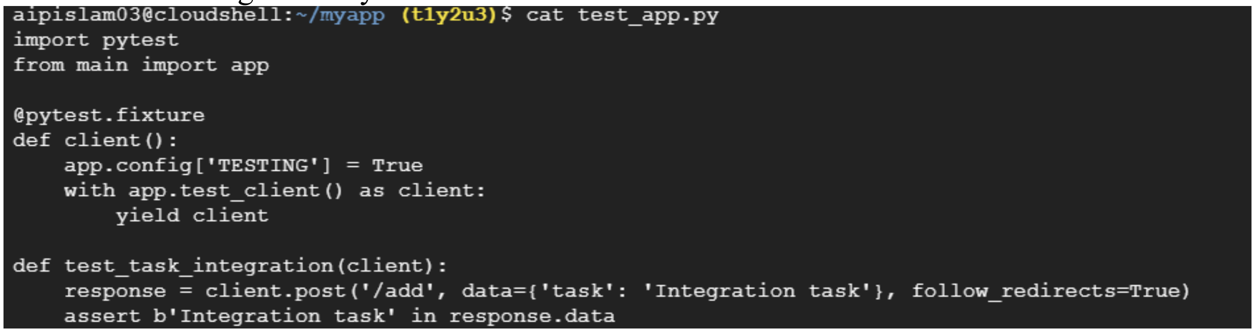
```
FAILED (failures=1)
(.venv) PS C:\Users\wwwis\Documents\bio\myapp> python -m unittest discover
..
----------------------------------------------------------------------
Ran 2 tests in 0.009s

OK
(.venv) PS C:\Users\wwwis\Documents\bio\myapp> 
```

Tested in local environment

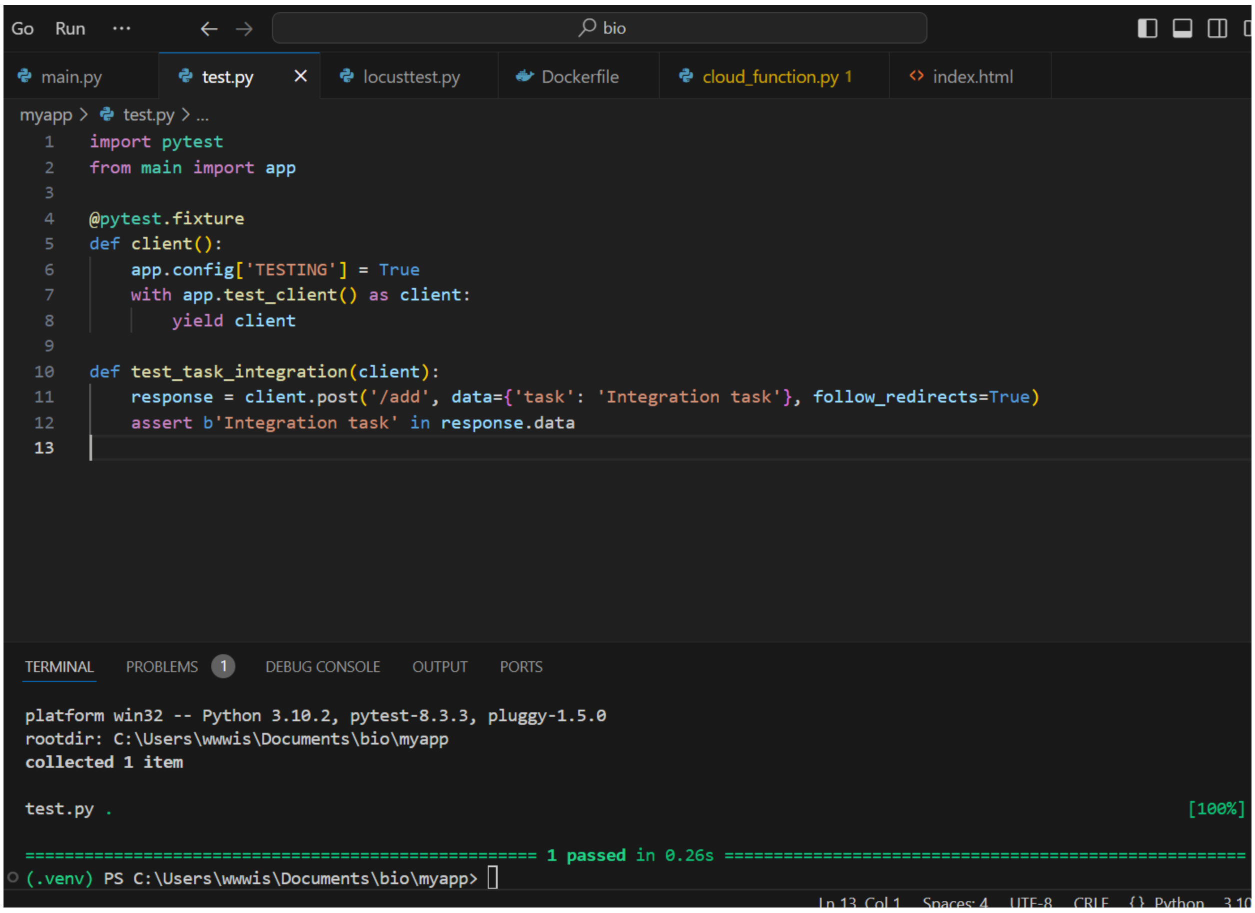
- test_client() – creates test client for the app and used to simulate requests
- assertEqual(response.status_code, 200) – checks if HTTP status code of response is 200(successful)
- assertIn(b'task', response.data) – checks if string 'task' is present in response.data

Integration testing

Integration testing is used to test different parts of application combined. Let's test id our add() function is working correctly.



```
aipislam03@cloudshell:~/myapp (t1y2u3)$ cat test_app.py
import pytest
from main import app

@pytest.fixture
def client():
    app.config['TESTING'] = True
    with app.test_client() as client:
        yield client

def test_task_integration(client):
    response = client.post('/add', data={'task': 'Integration task'}, follow_redirects=True)
    assert b'Integration task' in response.data
```



```python
import pytest
from main import app

@pytest.fixture
def client():
    app.config['TESTING'] = True
    with app.test_client() as client:
        yield client

def test_task_integration(client):
    response = client.post('/add', data={'task': 'Integration task'}, follow_redirects=True)
    assert b'Integration task' in response.data

```

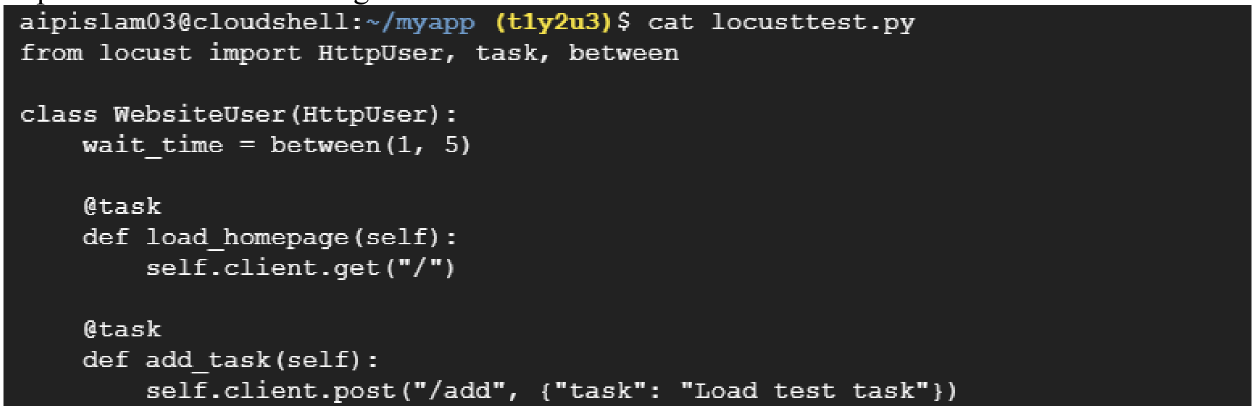
```
platform win32 -- Python 3.10.2, pytest-8.3.3, pluggy-1.5.0
rootdir: C:\Users\wwwis\Documents\bio\myapp
collected 1 item

test.py .                                                                                    [100%]

================================================== 1 passed in 0.26s ==================================================
(.venv) PS C:\Users\wwwis\Documents\bio\myapp>
```

Tested in local environment

app.config['TESTING'] – enable Flask's testing mode
yield – returns test client object
client.post() – simulates HTTP request
assert b'Integration task' in response.data – checks if string 'Integration task' appears in HTML response data.

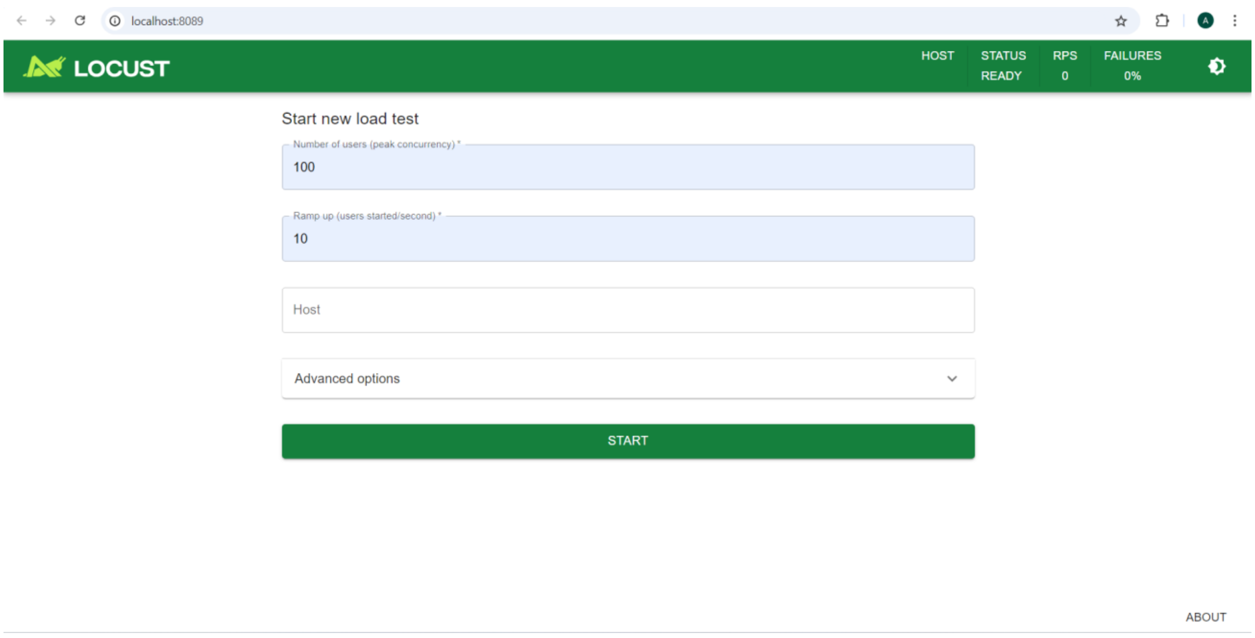
Load testing

Load testing simulates high levels of traffic to ensure our app can handle many requests.
In our application we can use locust for load testing. Let's create a separate locusttest.py file and implement basic load testing:



```
aipislam03@cloudshell:~/myapp (t1y2u3)$ cat locusttest.py
from locust import HttpUser, task, between

class WebsiteUser(HttpUser):
    wait_time = between(1, 5)

    @task
    def load_homepage(self):
        self.client.get("/")

    @task
    def add_task(self):
        self.client.post("/add", {"task": "Load test task"})
```

Locust allows us to configure numbers of users and ramp up:


localhost:8089

LOCUST

HOST	STATUS	RPS	FAILURES
	READY	0	0%

Start new load test

Number of users (peak concurrency) *
100

Ramp up (users started/second) *
10

Host

Advanced options

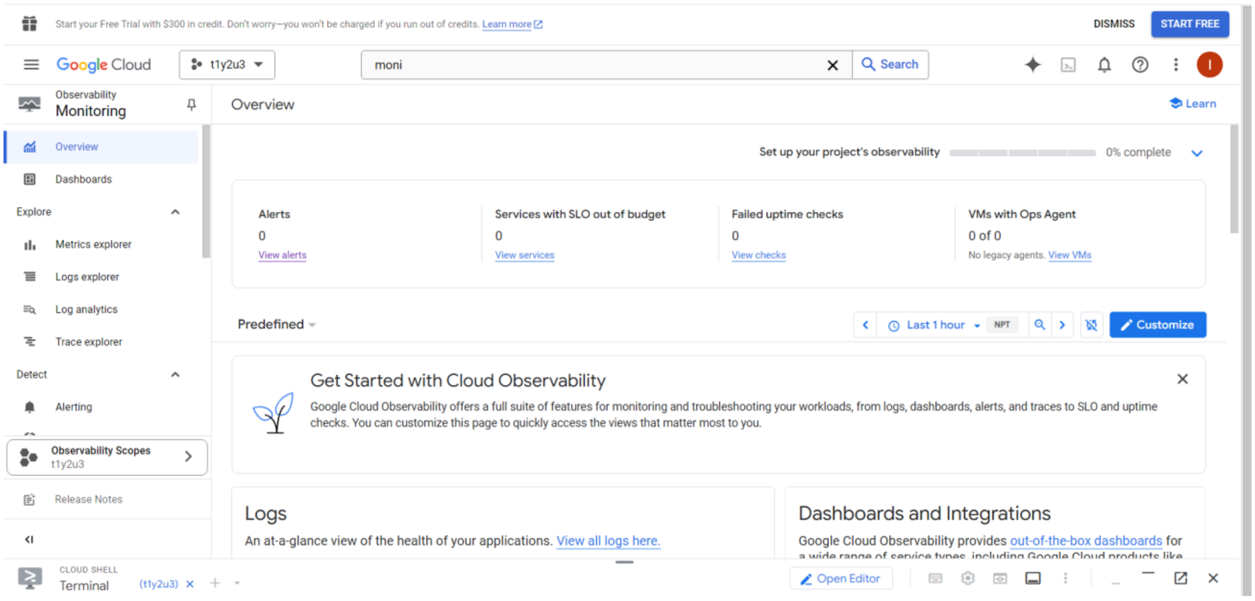
START

ABOUT

locust is run in local environment.

11.Monitoring and Maintenance

Monitoring and maintenance are crucial parts after deploying the application. Google provides built-in monitoring and maintenance services that users can use.


Start your Free Trial with $300 in credit. Don't worry—you won't be charged if you run out of credits. Learn more

DISMISS START FREE

Google Cloud t1y2u3 moni Search

Observability Monitoring

Overview

Set up your project's observability 0% complete

Alerts	Services with SLO out of budget	Failed uptime checks	VMs with Ops Agent
0	0	0	0 of 0
View alerts	View services	View checks	No legacy agents. View VMs

Predefined — Last 1 hour — Customize

Get Started with Cloud Observability

Google Cloud Observability offers a full suite of features for monitoring and troubleshooting your workloads, from logs, dashboards, alerts, and traces to SLO and uptime checks. You can customize this page to quickly access the views that matter most to you.

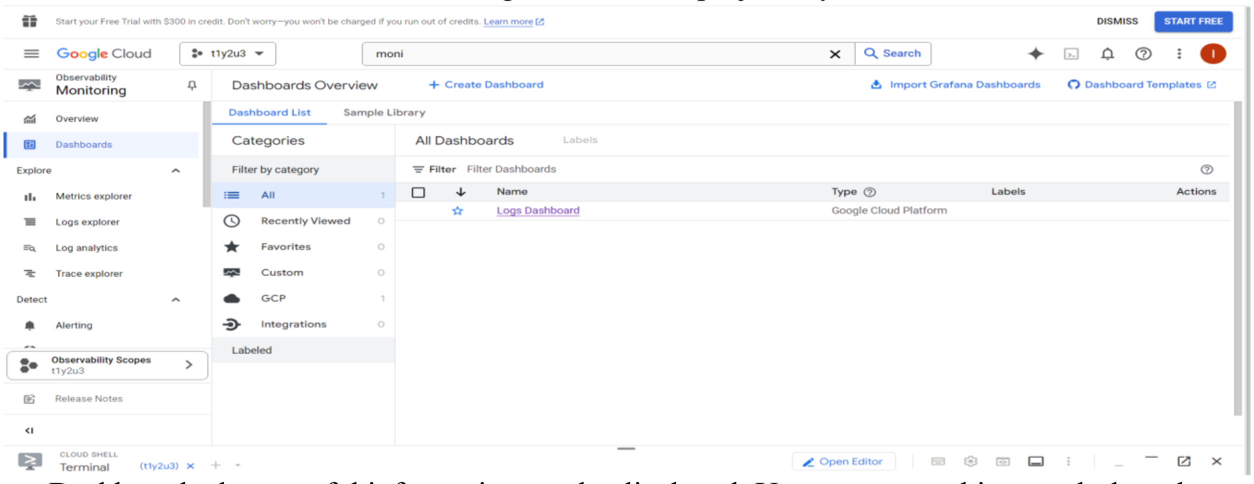
Logs

An at-a-glance view of the health of your applications. View all logs here.

Dashboards and Integrations

Google Cloud Observability provides out-of-the-box dashboards for a wide range of service types, including Google Cloud products like

Monitoring overview for project t1y2u3


Start your Free Trial with $300 in credit. Don't worry—you won't be charged if you run out of credits. Learn more

DISMISS START FREE

Google Cloud t1y2u3 moni Search

Observability Monitoring

Dashboards Overview + Create Dashboard Import Grafana Dashboards Dashboard Templates

Dashboard List Sample Library

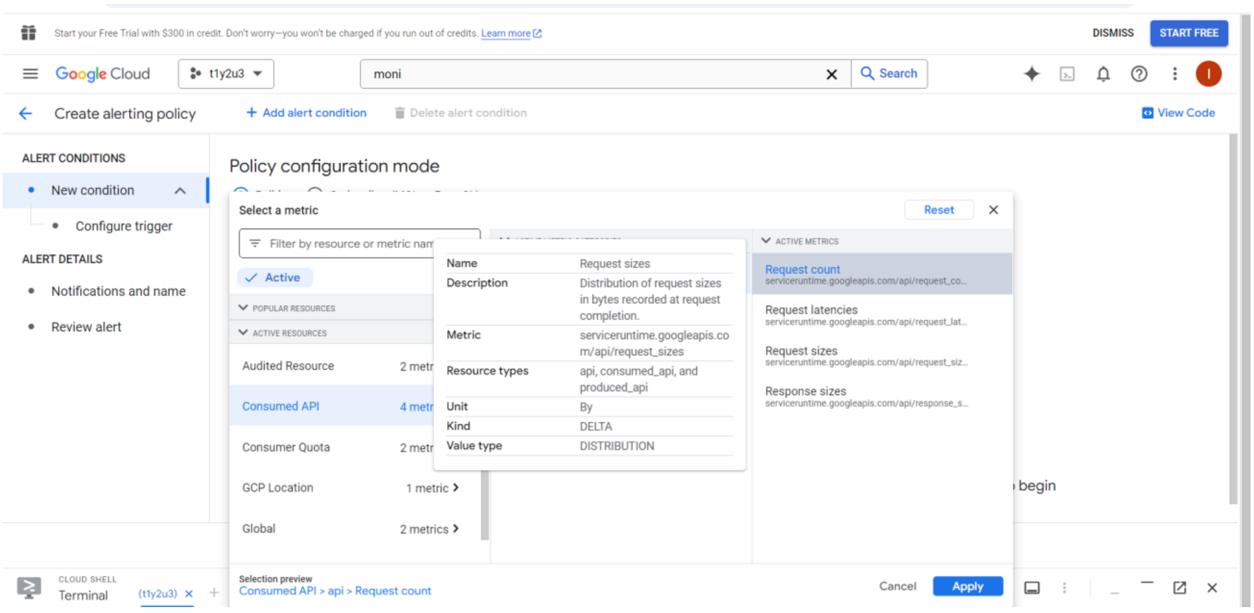
Categories	
All	1
Recently Viewed	0
Favorites	0
Custom	0
GCP	1
Integrations	0
Labeled	

All Dashboards

Filter Filter Dashboards

Name	Type	Labels	Actions
Logs Dashboard	Google Cloud Platform		

Dashboard where useful information can be displayed. User can create his own dasboards.


Start your Free Trial with $300 in credit. Don't worry—you won't be charged if you run out of credits. Learn more

DISMISS START FREE

Google Cloud t1y2u3 moni Search

Create alerting policy + Add alert condition Delete alert condition View Code

ALERT CONDITIONS
- New condition
  - Configure trigger

ALERT DETAILS
- Notifications and name
- Review alert

Policy configuration mode

Select a metric Reset

Filter by resource or metric name

✓ Active

POPULAR RESOURCES

ACTIVE RESOURCES

Audited Resource	2 metr
Consumed API	4 metr
Consumer Quota	2 metr
GCP Location	1 metric
Global	2 metrics

Name	Request sizes
Description	Distribution of request sizes in bytes recorded at request completion.
Metric	serviceruntime.googleapis.com/api/request_sizes
Resource types	api, consumed_api, and produced_api
Unit	By
Kind	DELTA
Value type	DISTRIBUTION

ACTIVE METRICS

Request count
serviceruntime.googleapis.com/api/request_co...

Request latencies
serviceruntime.googleapis.com/api/request_lat...

Request sizes
serviceruntime.googleapis.com/api/request_siz...

Response sizes
serviceruntime.googleapis.com/api/response_s...

begin

Selection preview
Consumed API > api > Request count

Cancel Apply

CLOUD SHELL Terminal (t1y2u3)

We can also create alerts to notify us if something is going wrong.

Setting up automated backups, health checks will ensure that our application is running correctly.

12.Challenges and Solutions

During the development of this project several challenges have occurred. Topics like deployment of docker images into Docker Kubernetes Engine, serverless functions and managing APIs in cloud environment were challenging, but documentation, websites and lecture materials helped to find solutions to them. Some challenges were:

- How can we define authentication in our API configuration file?
  We can use securityDefinitions and configure api keys.
- How can we deploy Containerized Application?
  By building and pushing our Docker Image into GCR and deploying it to GKE.

13.Conclusion

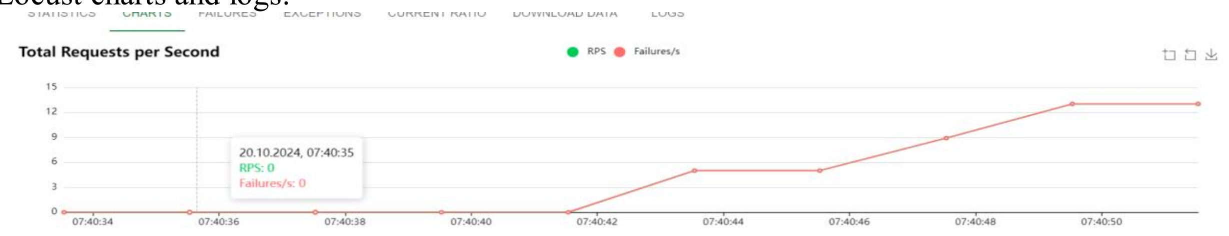
In this project, we successfully developed a Flask web application. By using Google App Engine we are now able to deploy our application. Through Google Cloud Functions, we integrated serverless function and by containerizing our application with Docker now we can deploy it by Google Kubernetes Engine. We configured APIs with Google Cloud Endpoints and through testing we ensured that applications functions are working correctly. Finally, we set up Google Cloud's monitoring and alerting tools.

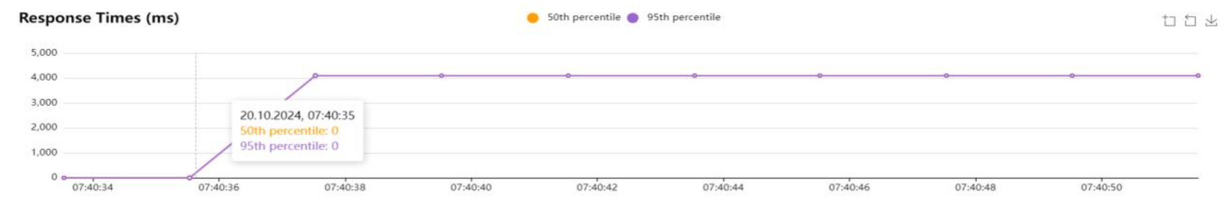
14.References

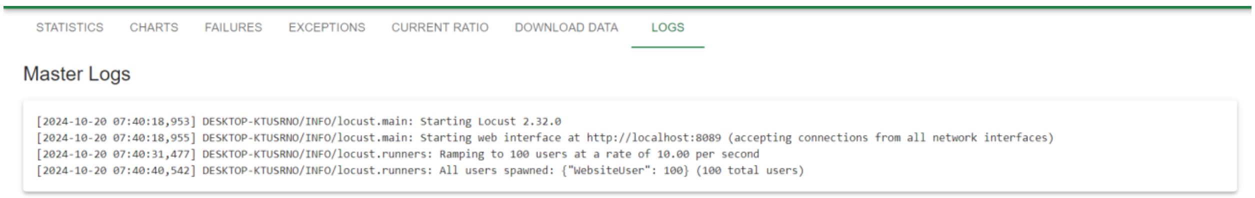
Google Cloud documentation - https://cloud.google.com/docs
Flask documentation - https://flask.palletsprojects.com/en/3.0.x/
Swagger API documentation - https://swagger.io/

15.Appendices

Locust charts and logs:







Master Logs

```
[2024-10-20 07:40:18,953] DESKTOP-KTUSRNO/INFO/locust.main: Starting Locust 2.32.0
[2024-10-20 07:40:18,955] DESKTOP-KTUSRNO/INFO/locust.main: Starting web interface at http://localhost:8089 (accepting connections from all network interfaces)
[2024-10-20 07:40:31,477] DESKTOP-KTUSRNO/INFO/locust.runners: Ramping to 100 users at a rate of 10.00 per second
[2024-10-20 07:40:40,542] DESKTOP-KTUSRNO/INFO/locust.runners: All users spawned: {"WebsiteUser": 100} (100 total users)
```