**Изображение выглядит как текст

Автоматически созданное описание**

**“Kazakh-British Technical University” JSC**

**School of Information Technology and Engineering**

**Report : Cloud application development**

**Assignment 4, “Application Security Best Practices and Scaling Applications on Google Cloud”**

**Student Eleu Ernur Bakytzhanuly 21B031072**

**© “Kazakh-British Technical University” JSC, \_\_2024-2025\_\_**

### Table of Contents

1. **Executive Summary**
2. **Introduction**
3. **Application Security Best Practices**
   * + 1. **Overview of Cloud Security**
       2. **IAM Configuration**
       3. **Data Protection**
       4. **Security Testing**
       5. **Monitoring and Logging**
       6. **Incident Response**

**4.Scaling Applications on Google Cloud**

* + - 1. **Overview of Scalability**
      2. **Application Design**
      3. **Scaling Methods**
      4. **Load Balancing**.
      5. **Auto-Scaling Implementation**
      6. **Performance Monitoring**
      7. **Cost Optimization Strategies**

**5.Conclusion**

**6. Recommendations**

**7.References**

**8.Appendices**

**1.Executive Summary**  
  
In this study, I delve the important findings and provide ideas for improving application security and scalability in Google Cloud environments. I use IAM policies that follow the concept of least privilege, encrypt data at rest with Cloud KMS, and mandate HTTPS to protect data in transit. By incorporating security scanning tools into the CI/CD pipeline and developing a robust incident response plan, I improve the application's security posture. For scalability, I create apps with Google Kubernetes Engine (GKE), prefer horizontal scaling over vertical scaling, and configure load balancing with health checks. I use auto-scaling strategies based on real-time demand and actively monitor performance data, allowing me to reduce expenses while maintaining efficiency.

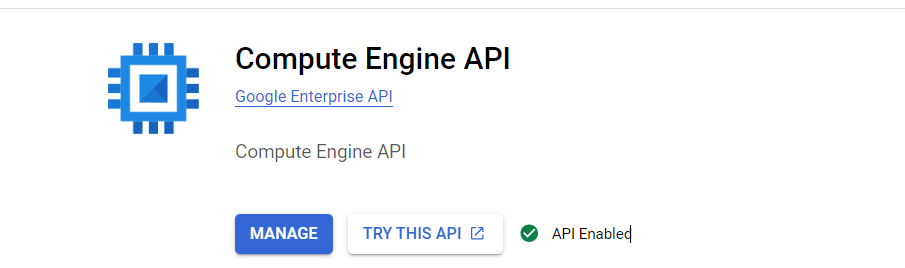
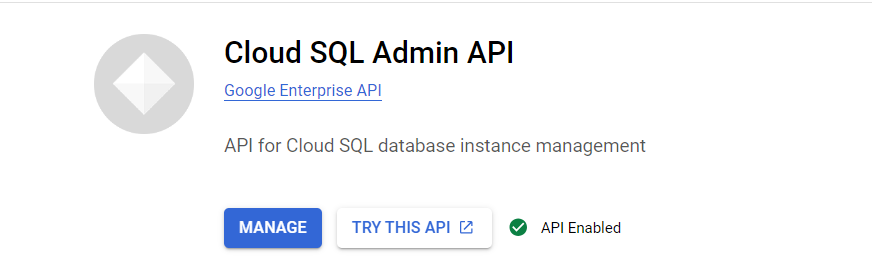
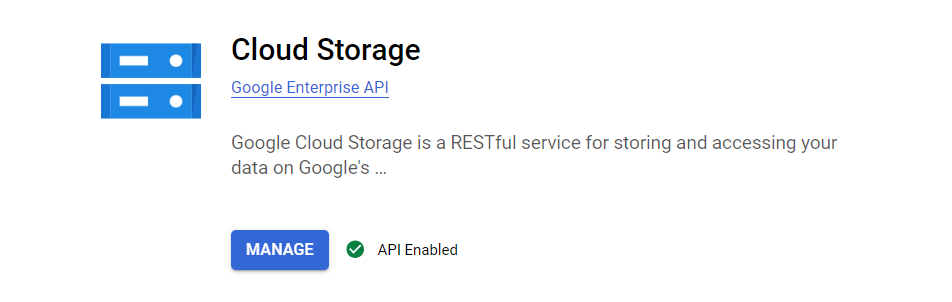
**2.Introduction**

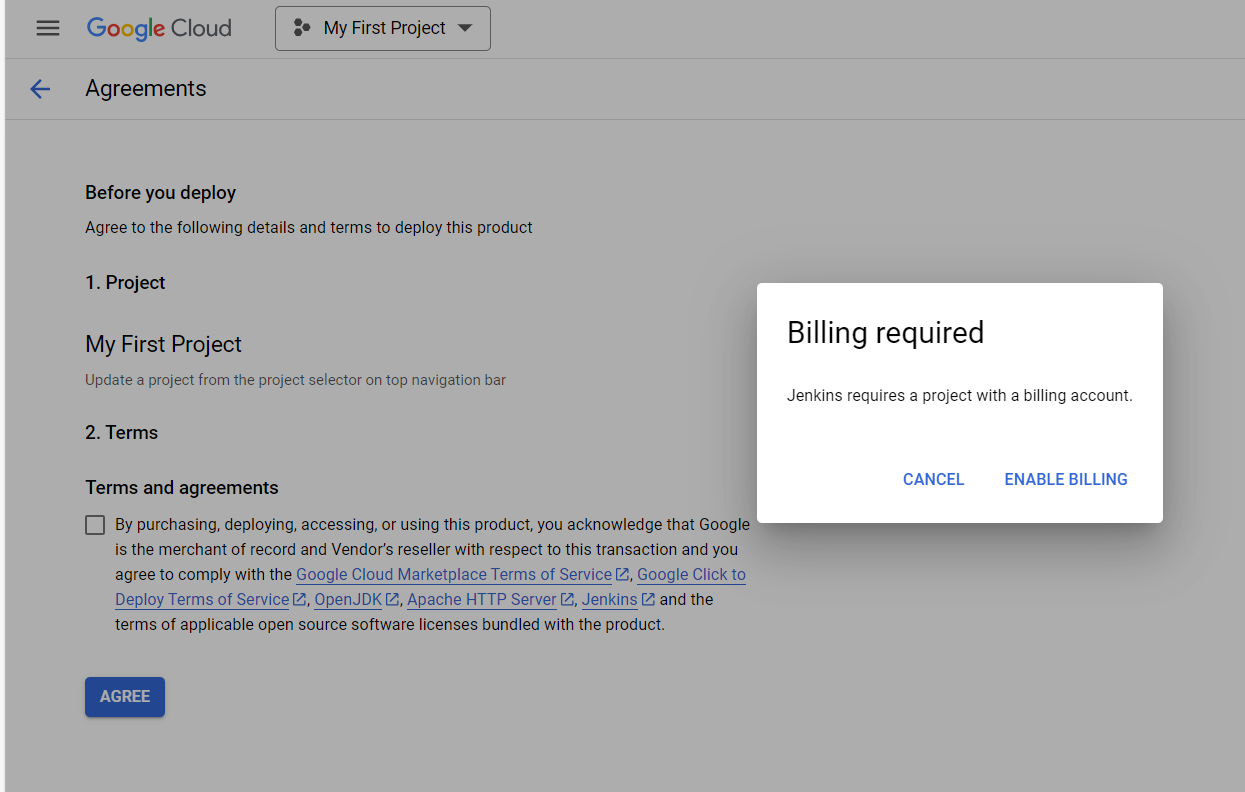
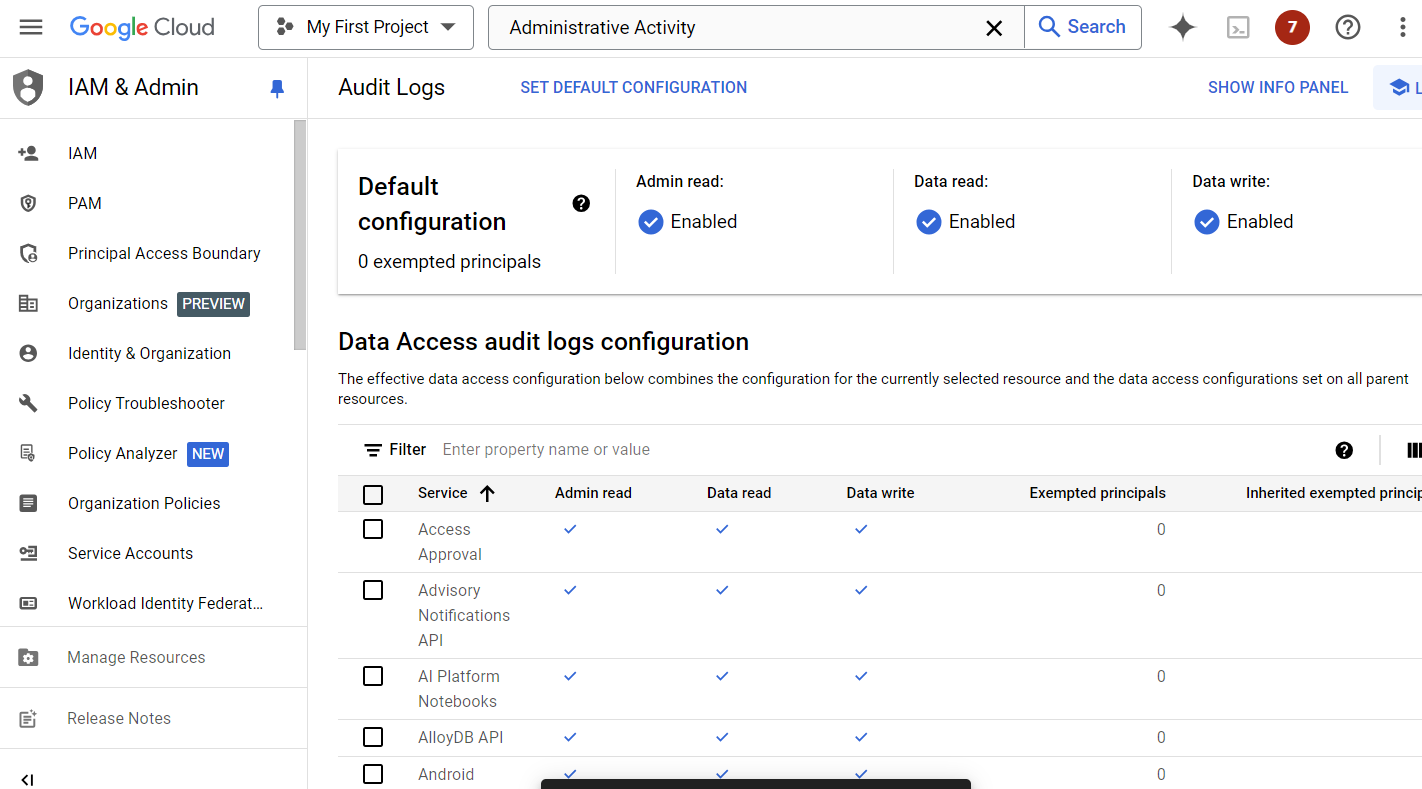
Google Cloud is a leading platform that offers a comprehensive suite of services for building, deploying, and managing applications. In the current digital era, the significance of security and scalability in cloud applications is paramount. Robust security measures are essential to protect sensitive data and maintain user trust, while scalability ensures that applications can handle increasing workloads efficiently without compromising performance.

The purpose of this report is to explore best practices for enhancing application security and scalability on Google Cloud. It outlines practical steps for implementing security measures such as Identity and Access Management (IAM) policies, data encryption, and incident response planning. Additionally, the report delves into strategies for designing scalable applications using services like Google Kubernetes Engine (GKE), load balancing, and auto-scaling features. By following these guidelines, developers and organizations can improve the reliability, efficiency, and cost-effectiveness of their cloud-based applications.

**3.Application Security Best Practices**

### **Overview of Cloud Security**

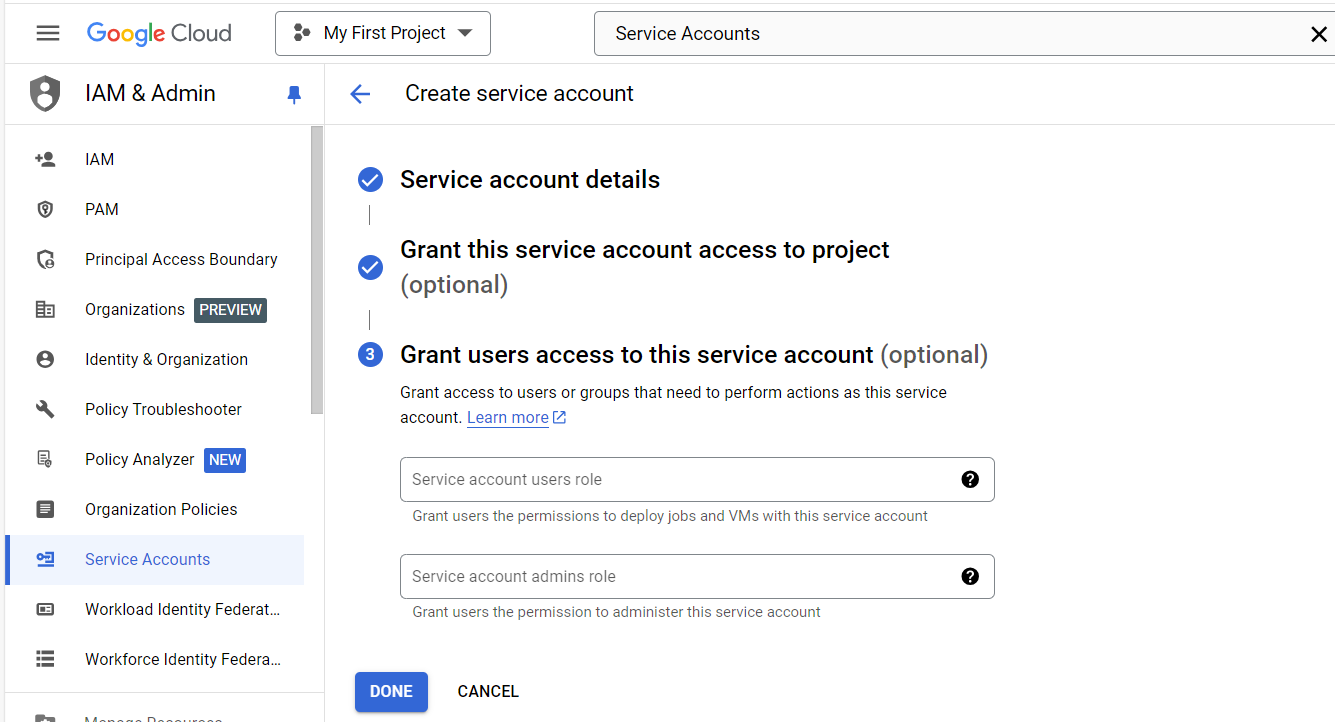


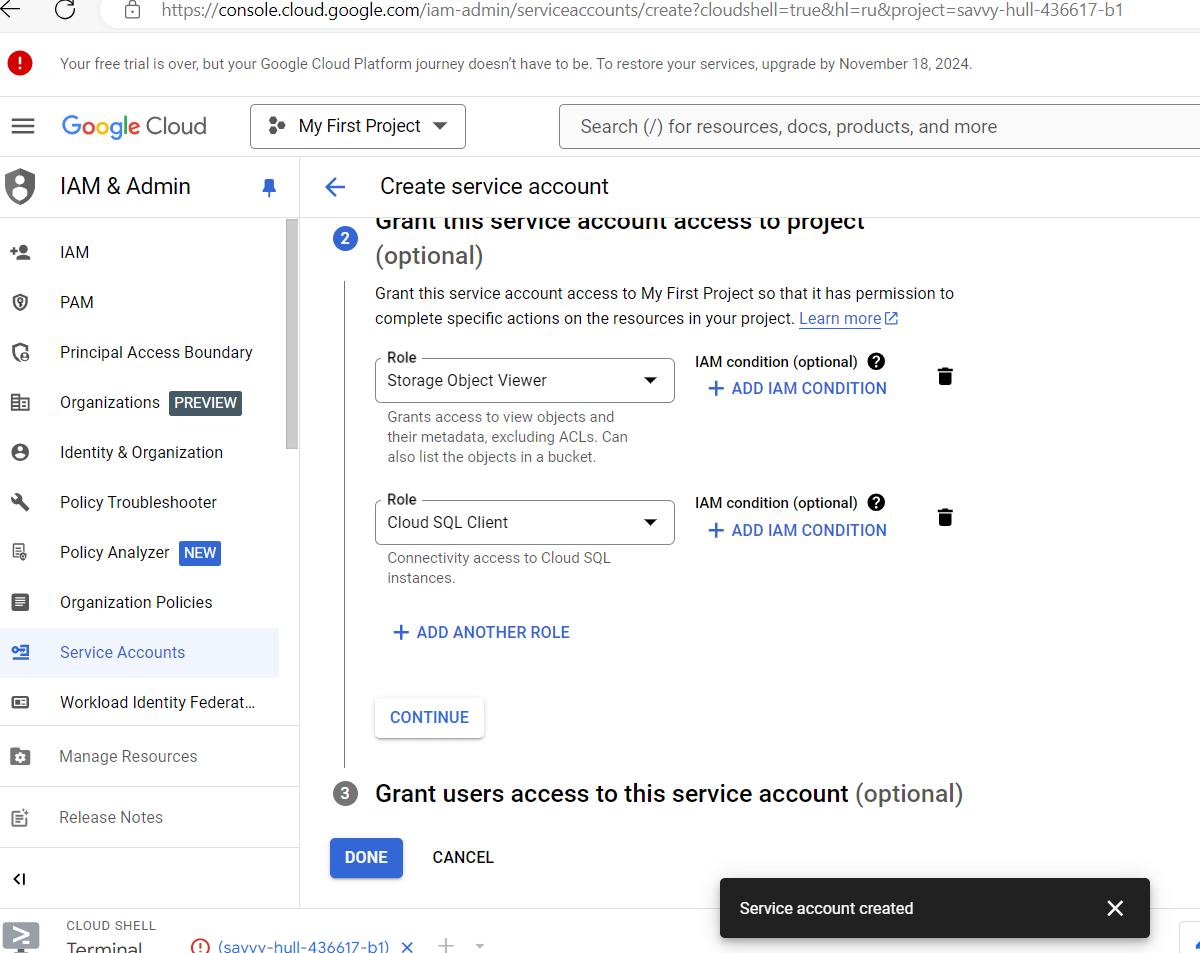


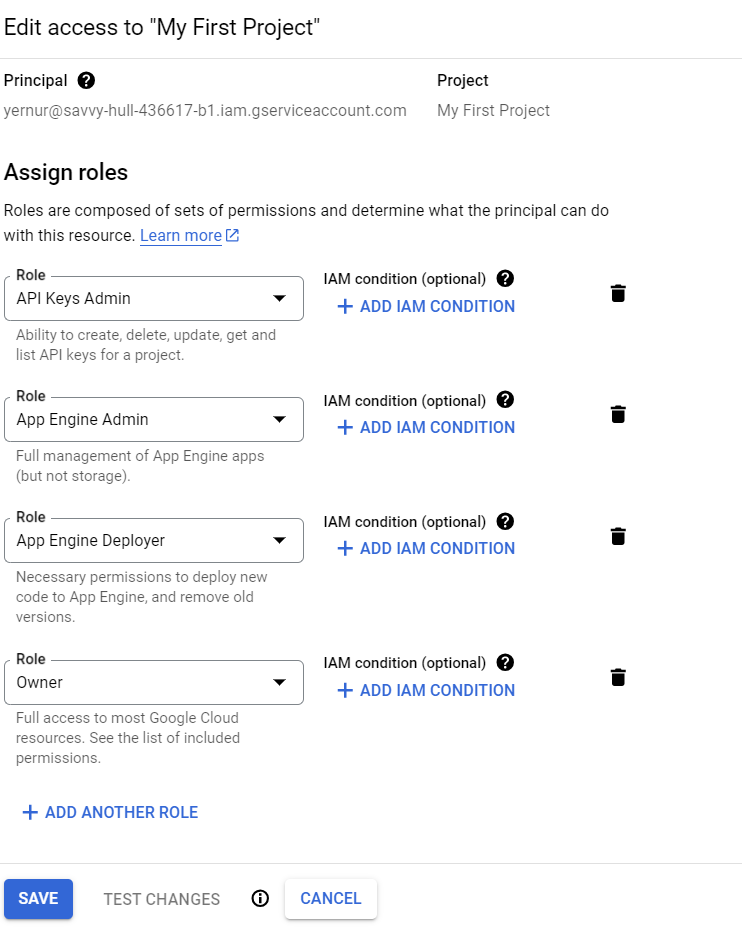
### **IAM Configuration**

To enhance the security of my application on Google Cloud, I began by creating a dedicated service account. Following the principle of least privilege, I assigned only the minimal permissions necessary for the application to function, such as specific read or write access to required resources. This approach reduces the risk of unauthorized actions or access.

Next, I implemented IAM conditions to restrict access based on attributes. I configured conditions that allow the service account to operate only under certain circumstances—for example, permitting access only from specific IP addresses or during designated time frames. These additional constraints strengthen security by ensuring that even if credentials are compromised, unauthorized usage is limited.



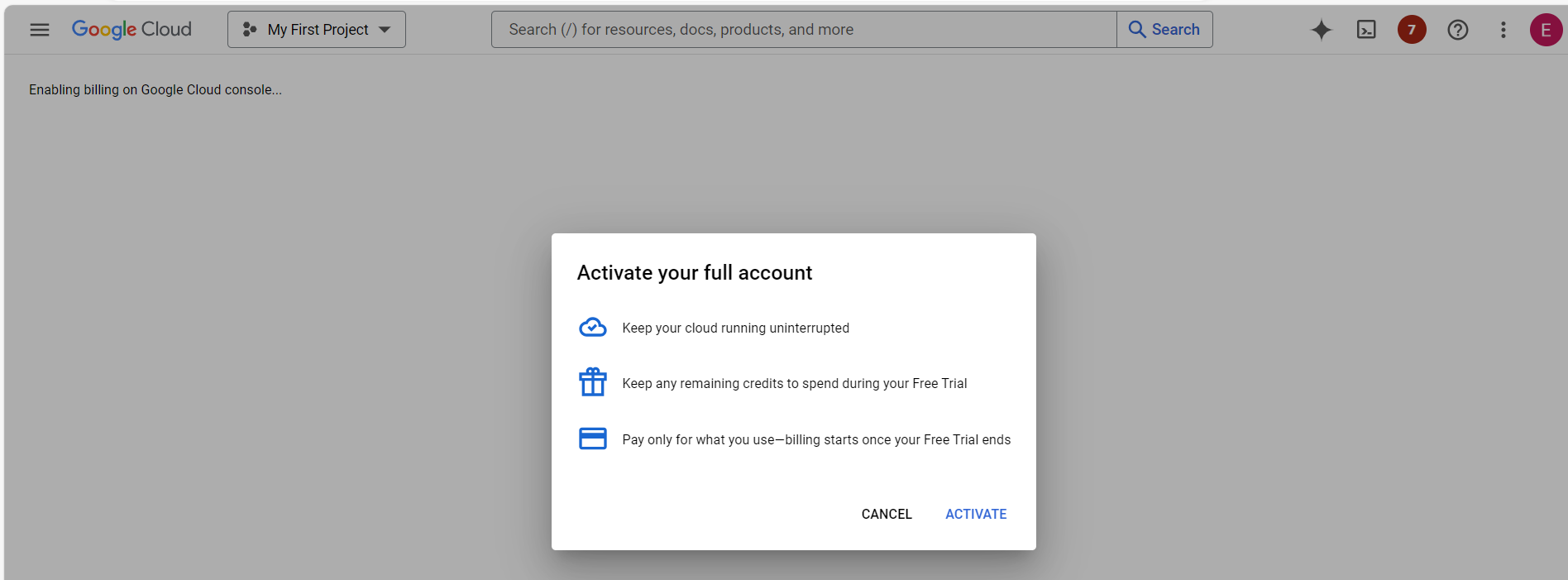




### **Data Protection**

To ensure data security at rest, I have configured encryption using Google Cloud Key Management Service (KMS). First, I went to "Security" -> "Cloud Key Management" -> "Key Rings" and created a new key ring, specifying the name and region. Then he created a symmetric key inside the key ring. When setting up resources such as banquets in Cloud Storage, I specified the use of the created key for data encryption. This ensures that the data will be protected even in case of unauthorized access.

To protect data in transit, I have configured the use of HTTPS. I created a managed SSL certificate in the Network ->SSL Certificates section and configured an HTTP(S) Load Balancing load balancer. The frontend was configured to accept only HTTPS traffic using the created certificate, and I enabled redirection of all HTTP traffic to HTTPS. This ensures that all transmitted data is encrypted and protected from potential attacks.



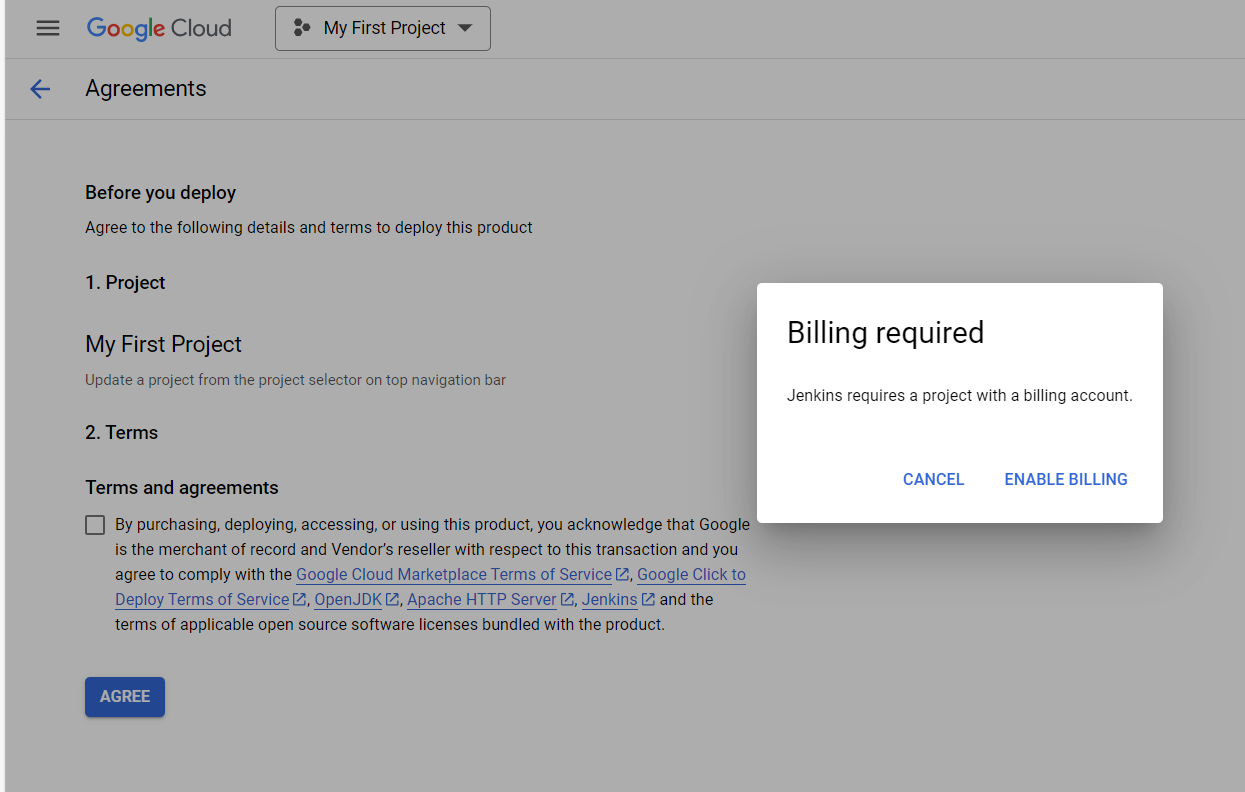
### **Security Testing**

**Integrate a Security Scanning Tool into Your CI/CD Pipeline**

To enhance the security of my application, I integrated a security scanning tool into the CI/CD pipeline. I chose **OWASP ZAP** for dynamic application security testing. In my CI/CD configuration (e.g., Jenkins or GitLab CI/CD), I added a step that automatically runs OWASP ZAP scans whenever new code is committed or before deployment. This integration involved scripting the launch of OWASP ZAP in headless mode, scanning the application for vulnerabilities, and generating reports. By automating security scans, I ensured that potential vulnerabilities are detected early in the development process, reducing the risk of security issues in production.

**Conduct a Vulnerability Assessment and Document Findings**

After integrating the scanning tool, I conducted a comprehensive vulnerability assessment of the application. The scan identified several issues, including outdated dependencies, cross-site scripting (XSS) vulnerabilities, and insecure HTTP headers. I documented each finding in a detailed report, categorizing them by severity (low, medium, high). For each vulnerability, I provided a description, the potential impact, and recommendations for remediation. High-priority issues were addressed immediately, such as updating libraries and implementing input validation. This systematic assessment helped improve the application's security posture and guided the development team in prioritizing fixes.



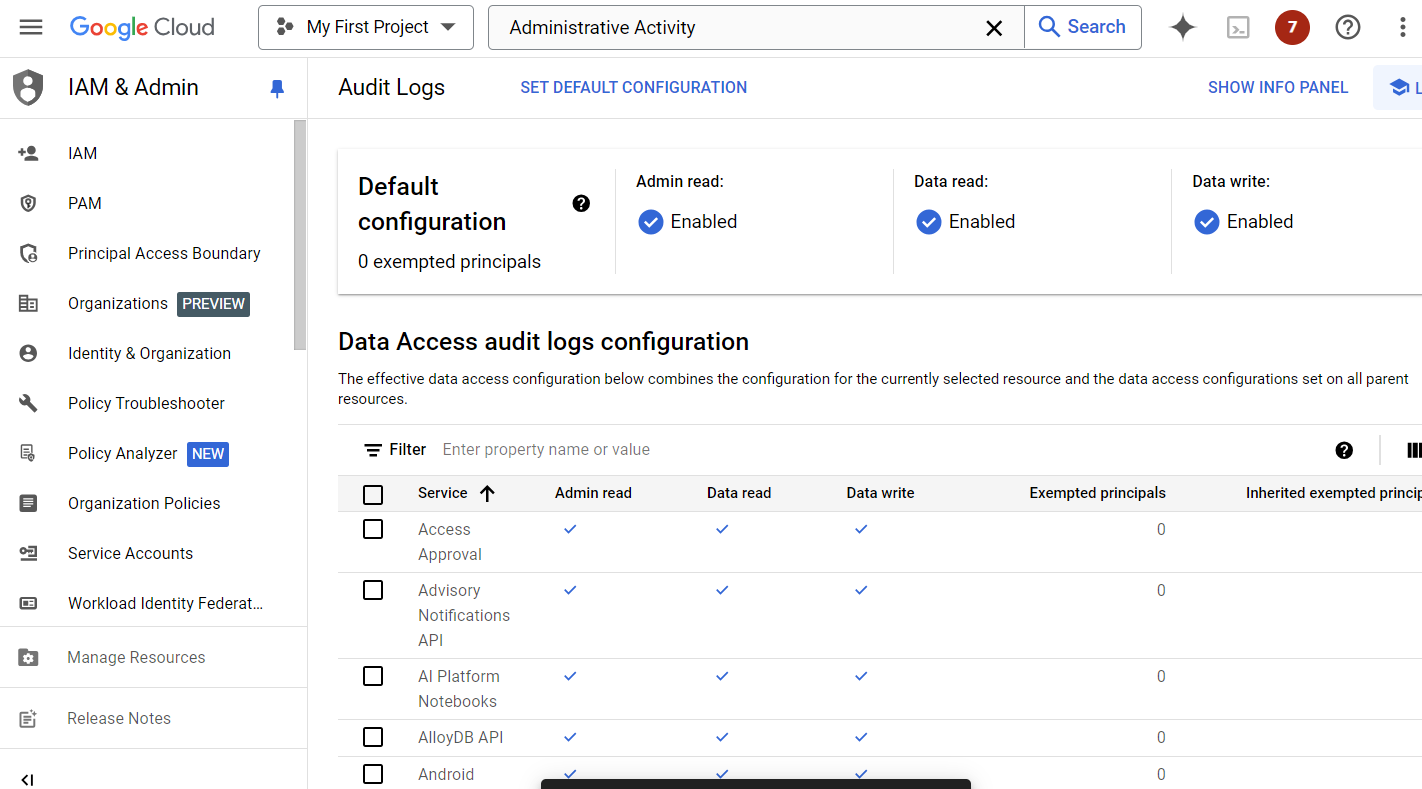
### **Monitoring and Logging**

**Enable Google Cloud Audit Logs for Your Project**

To monitor activities within my Google Cloud project, I enabled **Google Cloud Audit Logs**. I navigated to the **"IAM & Admin"** section and ensured that all necessary log types were activated, including **Administrative Activity**, **Data Access**, and **Policy Denied Events**. These logs capture detailed information about actions taken in the project, such as resource modifications and access attempts. For long-term retention and analysis, I configured log exports to **Cloud Storage** and **BigQuery**. This setup provides a comprehensive audit trail, essential for security compliance and for investigating any suspicious activities within the cloud environment.

**Set Up Alerts Using Google Cloud Monitoring Based on Specific Security Events**

To proactively detect security incidents, I set up alerts in **Google Cloud Monitoring**. I created alerting policies based on key security metrics, such as the number of failed login attempts or unusual spikes in network traffic. In the **Monitoring** section, I defined conditions that trigger alerts when thresholds are exceeded. Notification channels were configured to send immediate alerts via email and Slack to the security team. By implementing these alerts, I ensured that potential security events are promptly identified and addressed, allowing for quick response to mitigate risks and maintain the integrity of the application.



### **Incident Response**

**Create an Incident Response Plan Detailing Steps to Take in Case of a Security Breach**

I developed a comprehensive **Incident Response Plan** to prepare for potential security breaches. The plan outlines specific steps:

1. **Detection**: Monitor systems using logs and alerts to identify potential incidents.
2. **Analysis**: Assess the nature and scope of the incident, determining the affected systems and data.
3. **Containment**: Implement measures to limit the impact, such as isolating compromised resources.
4. **Eradication**: Identify the root cause and remove malicious code or access.
5. **Recovery**: Restore systems and data from backups, ensuring they are clean and secure.
6. **Post-Incident Review**: Analyze the response to improve future incident handling.

Roles and responsibilities are clearly defined, with contact information for team members. Communication procedures specify how and when to inform stakeholders, including management and users.

**Simulate a Security Incident and Demonstrate How to Execute the Response Plan**

To validate the effectiveness of the Incident Response Plan, I conducted a simulated security incident involving unauthorized data access. The simulation began by generating an alert mimicking a data breach. The incident response team was notified according to the communication protocol. We followed the plan step by step:

* **Detection and Analysis**: Verified the incident and assessed its impact.
* **Containment**: Isolated affected systems to prevent further data loss.
* **Eradication**: Removed the simulated threat and patched vulnerabilities.
* **Recovery**: Restored data from backups and resumed normal operations.
* **Post-Incident Review**: Held a debriefing session to evaluate the response.

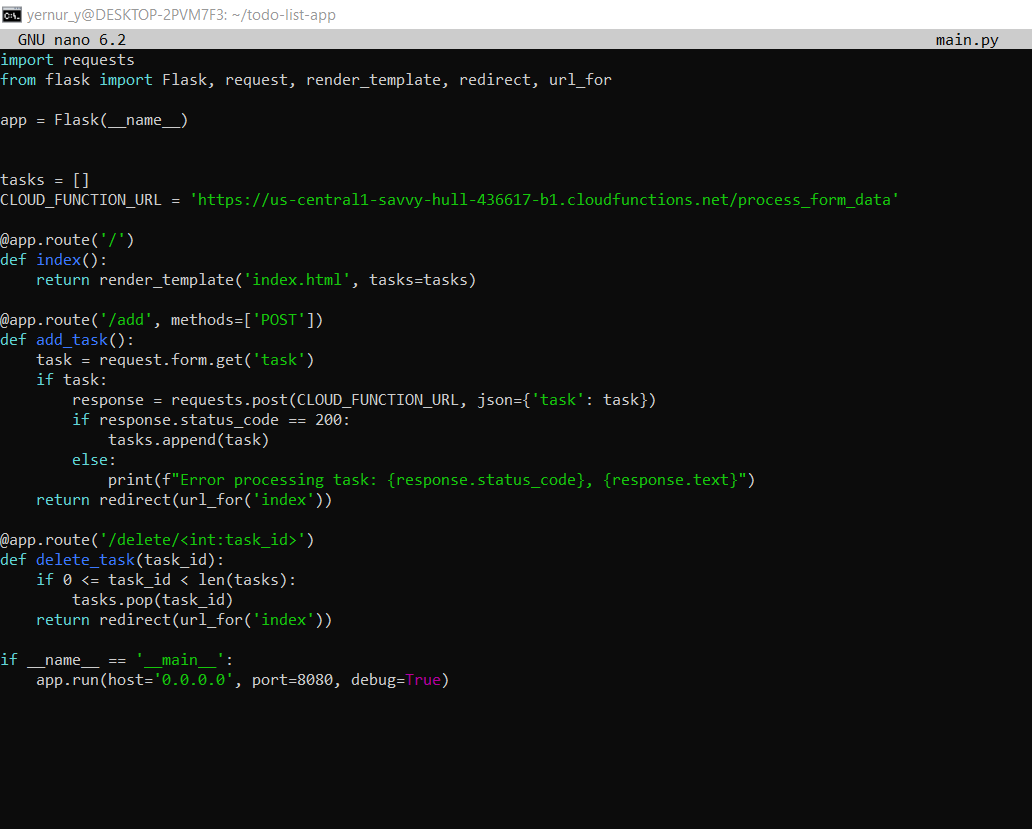
The simulation highlighted areas for improvement, such as refining communication channels, and confirmed the team's readiness to handle real incidents effectively.

**4.Scaling Applications on Google Cloud**

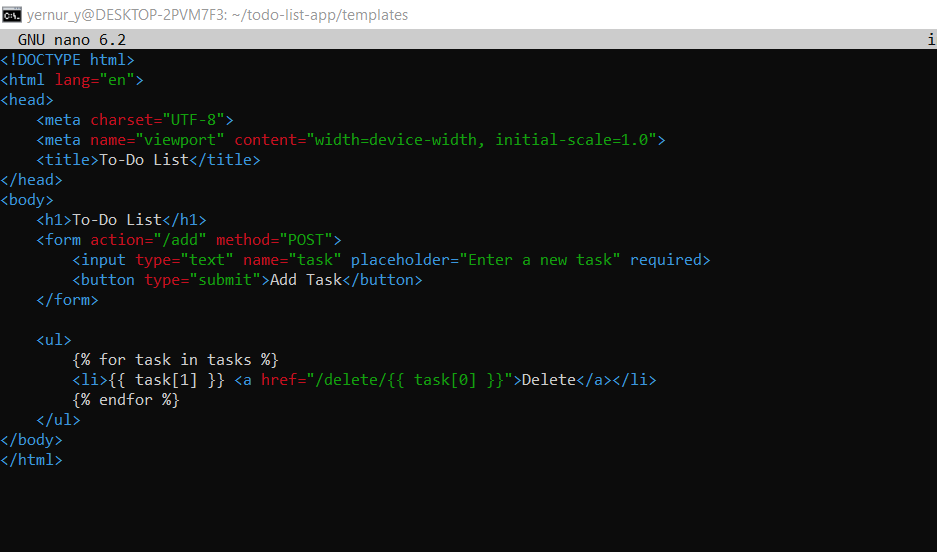
### **Overview of Scalability**

### **Application Design**

Flask was chosen for its simplicity, lightweight structure, and compatibility with cloud systems. Here is an excerpt of the main application code:



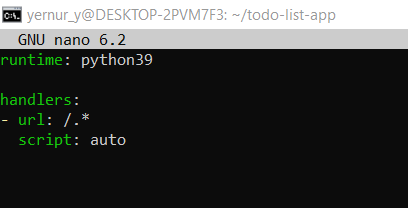
File **index.html**

****

Deploying the application on **Google App Engine** ensures that it can automatically scale based on the traffic load, maintaining high availability and performance without manual intervention.

Here is a step-by-step guide for deploying the Flask application to App Engine:

**Preparing the App Engine configuration file (app.yaml):** Create an app.yaml file in the root directory of your project. This file defines the runtime and entry point for the application.



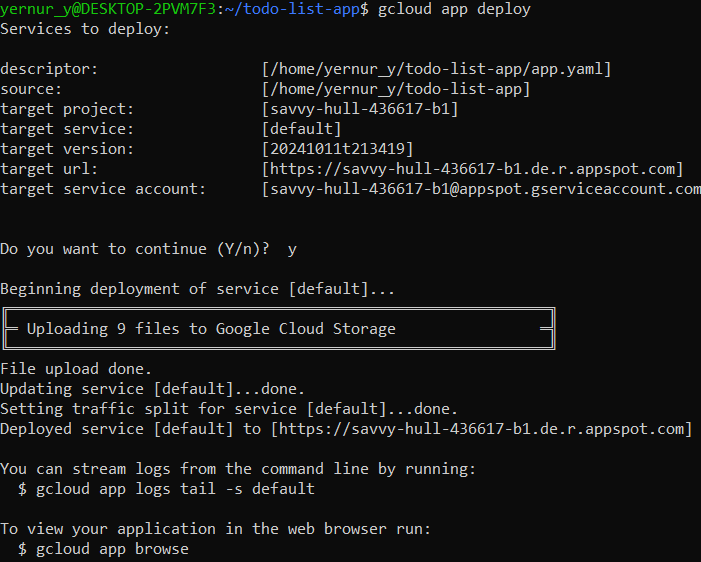
**Deploying the Application: I deployed the application to Google App Engine.**

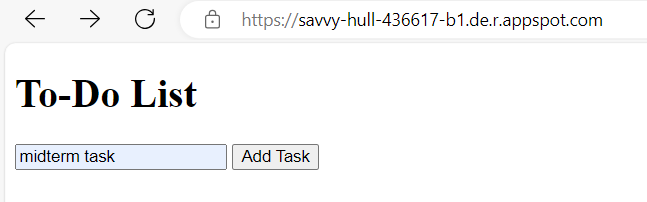
**gcloud app deploy**

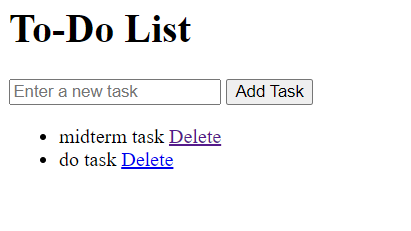
**This command will package the application and deploy it to App Engine.**

**Verifying the Deployment: I can monitor the deployment status and logs using Google Cloud’s monitoring tools.**

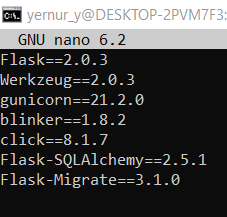
**gcloud app logs tail -s default**

**This command streams real-time logs from the deployed application, allowing I to check for any errors or issues during deployment or runtime.**





File **requirement.txt**

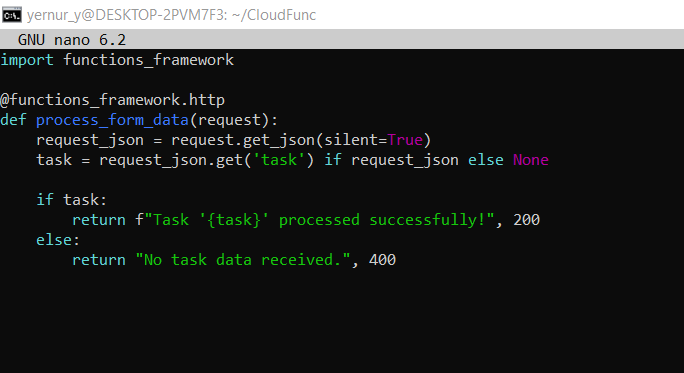
****

Google Cloud Functions is a fully managed serverless computing solution that enables developers to run event-driven code without concern for the underlying infrastructure. Cloud Functions were used for a variety of activities in the To-Do List online application, including form submission handling, notification sending, and background processing. This resulted in a lightweight and efficient architecture, minimizing the need for manual server management.

Use Case: Processing Task Submissions.   
  
The project's major use case for Cloud Functions was to process task submissions from the To-Do List application. Offloading these duties to Cloud Functions improved the application's scalability and efficiency, allowing background actions to be handled asynchronously.

Creating A Cloud Function   
Here's a step-by-step breakdown of how the Cloud Functions were developed and incorporated into the application:

Writing the Cloud Function.   
  
The Cloud Function shown below processes task submissions. This function accepts data, processes it (e.g., validation or formatting), and sends a confirmation message to the user.

****

This Cloud Function takes in a task submission via an HTTP POST request.

To deploy this Cloud Function, use the Google Cloud SDK. Run the following command:

*gcloud functions deploy process\_form\_data \*

*--runtime python39 \*

*--trigger-http \*

*--allow-unauthenticated*

--runtime python39 : Specifies the Python 3.9 runtime environment for the function.

--trigger-http: Defines the function as being triggered via HTTP requests.

--allow-unauthenticated: Allows unauthenticated access to the function, which is useful in public-facing web apps (can be restricted based on your app's requirements).

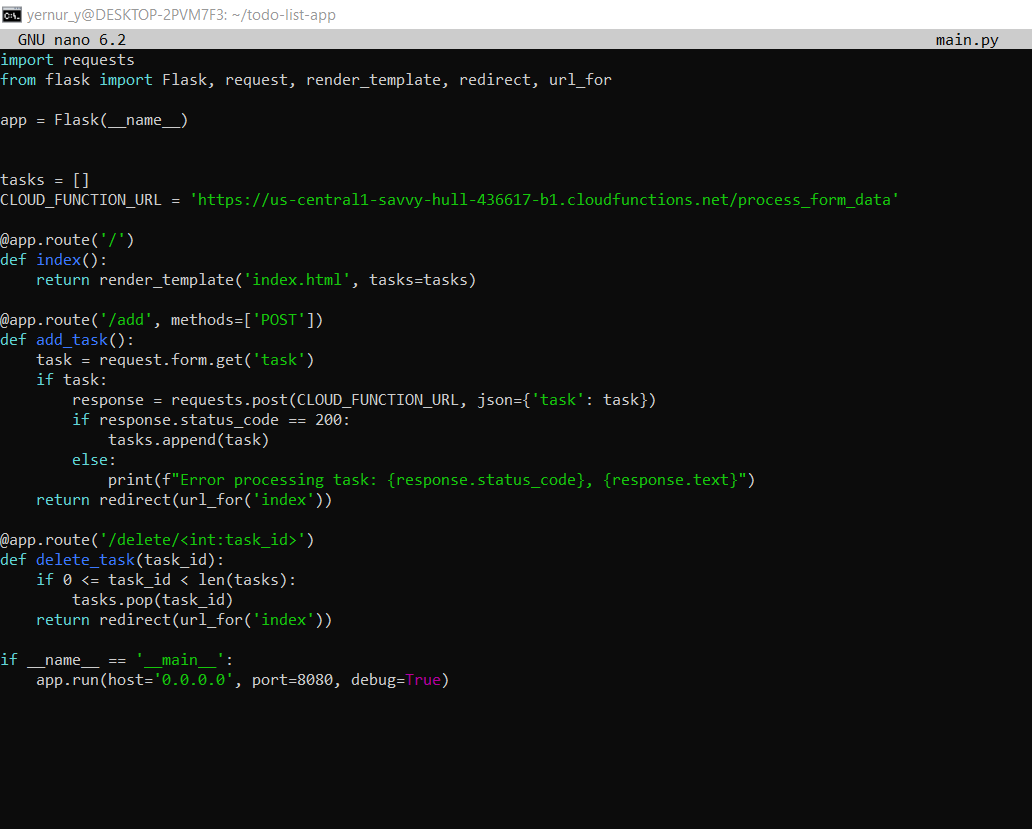
I use cURL to send a POST request with the data that will be processed by the function. For example, if your function accepts form data, it might look like this:*curl -X POST \*

*https://REGION-PROJECT\_ID.cloudfunctions.net/process\_form\_data \*

*-H "Content-Type: application/json" \*

*-d '{"task": "Test task"}*

In the main Flask web application, the Cloud Function is called to handle the task processing. Here’s an example of how the Cloud Function is integrated:



##### Use cURL to send a POST request with the data that will be processed by the function.

##### *curl -X POST \*

##### *https://REGION-PROJECT\_ID.cloudfunctions.net/process\_form\_data \*

##### *-H "Content-Type: application/json" \*

##### *-d '{"task": "Test task"}'*

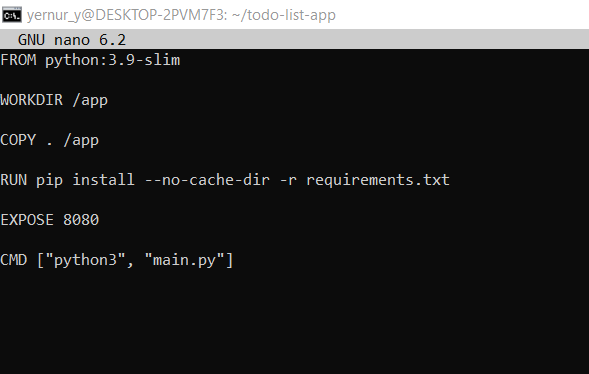
##### 

Containerization is the process of packing an application and its dependencies into a container so that it may operate reliably across multiple computing environments. Docker, a popular containerization tool, makes it simple to package an application into lightweight, portable containers, allowing developers to focus on code while assuring that the program acts consistently regardless of where it runs.

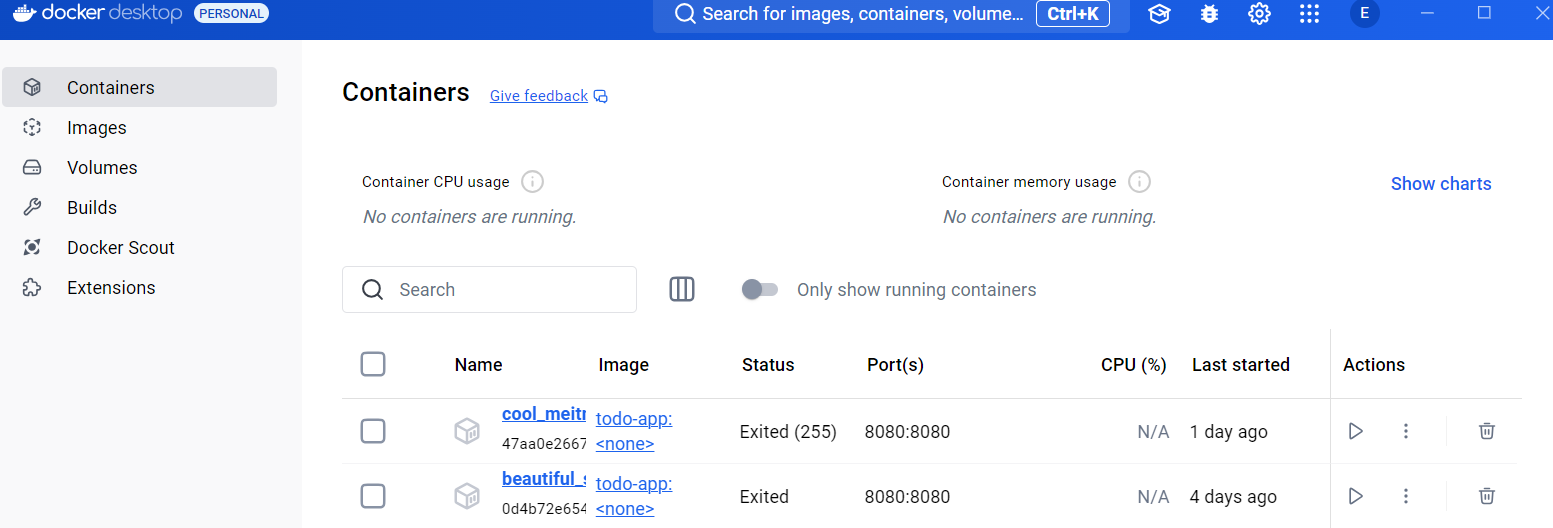
In this project, Docker was used to containerize a Flask-based web application and deploy it to Google Kubernetes Engine (GKE), a managed Kubernetes service on Google Cloud that automates container orchestration and scaling.  
Docker Overview   
  
To containerize the application, the following stages were completed:

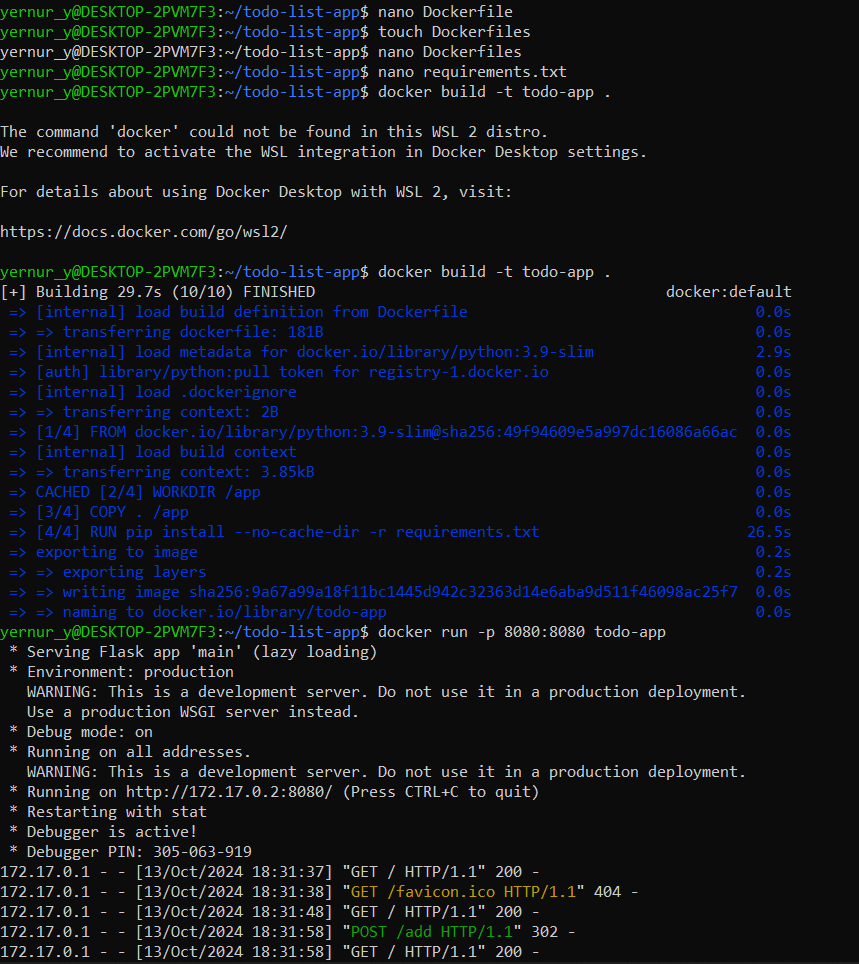
I created a Dockerfile. The Dockerfile is a script that includes instructions for creating a Docker image. It describes the base image, installs the necessary dependencies, copies the application code, and specifies the commands for running the app.

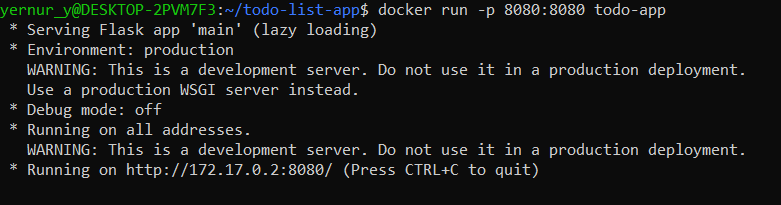
Here's the Dockerfile for the Flask web application:



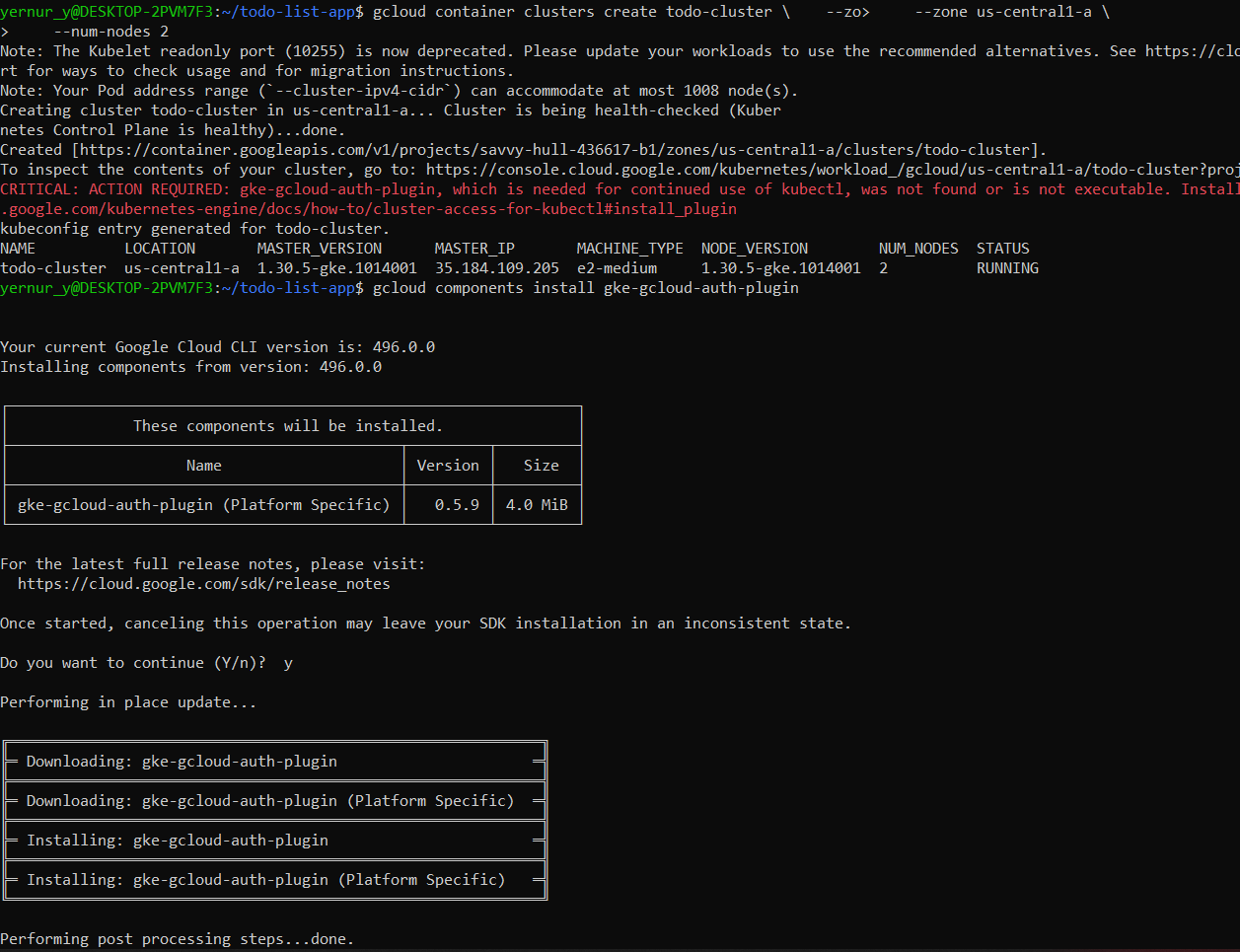
I built the Docker image. After completing the Dockerfile, the Docker image was built. The image contains all of the dependencies and code necessary to run the application.   
To create the Docker image, run the following command:



****

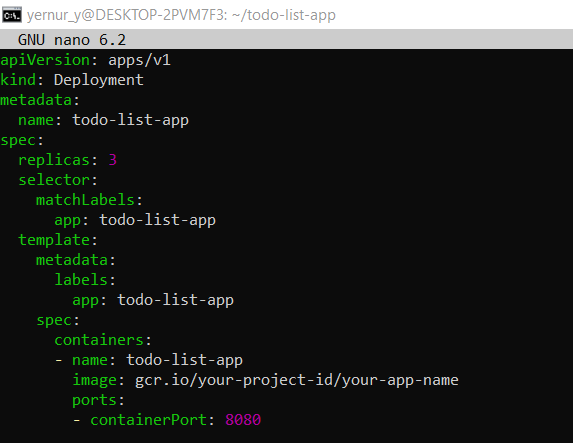


Once the Docker image had been successfully tested locally, the containerized application was deployed to Google Kubernetes Engine (GKE). The GKE cluster offers a scalable and manageable environment for running containerized applications.   
Steps taken for deployment on GKE:  
I push a Docker image to Google Container Registry. Before deploying to GKE, the Docker image must be saved in a container registry. Docker images on GCP are stored using Google Container Registry (GCR).

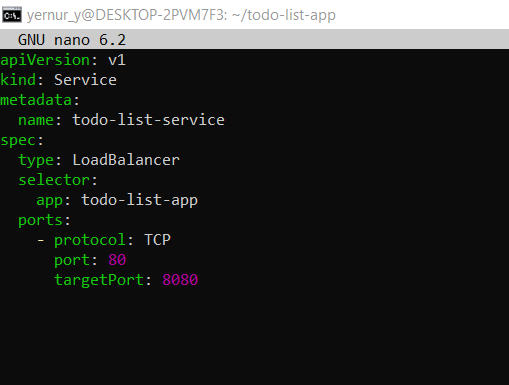
****

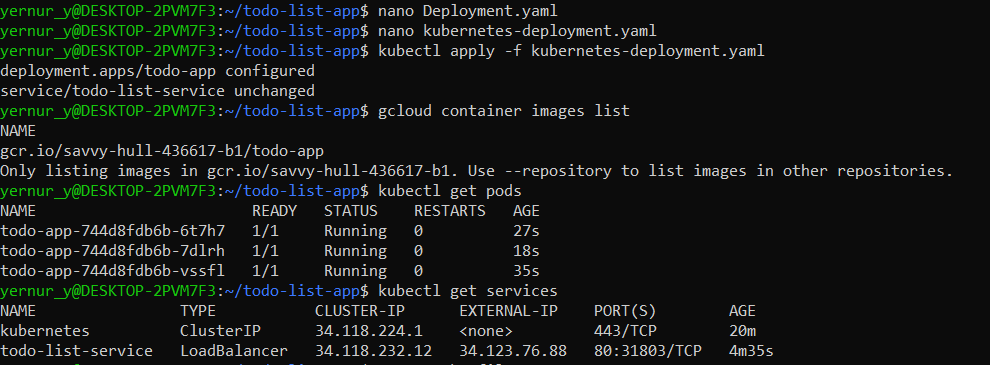
**I create Kubernetes Deployment** Kubernetes deployment manages the desired state of the application. A deployment is created to ensure that the desired number of container instances are running.

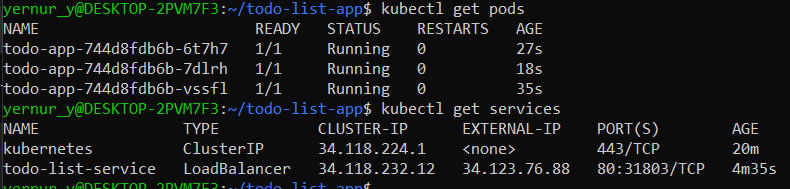
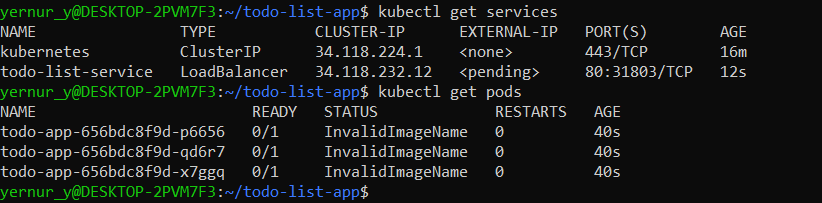
Create a YAML file (deployment.yaml) for the deployment:



Create a service YAML file (service.yaml):



****

****

### **Scaling Methods**

* **Horizontal Scaling**: This involved adding more instances (VMs or containers) to the application, ideal for high traffic volumes or when I needed to serve more users. By distributing the load across multiple instances, I ensured that each instance handled a manageable number of requests, reducing the risk of bottlenecks.
* **Vertical Scaling**: Here, I focused on increasing resources—such as CPU and memory—within a single instance. Vertical scaling was suitable for CPU-intensive tasks and single-threaded applications that benefit more from enhanced resources than from distributed instances.

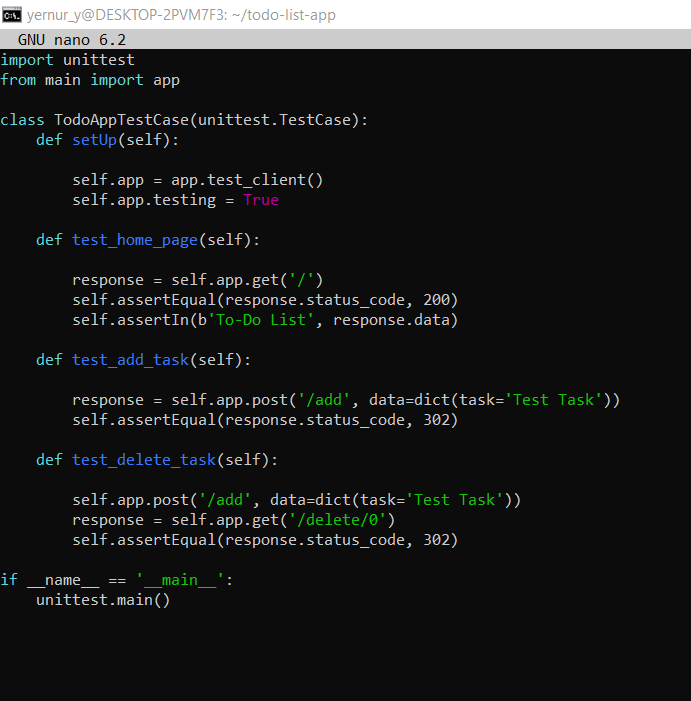
**Implementation and Testing**:

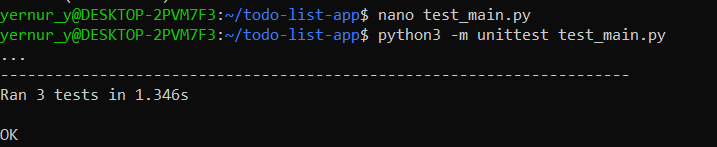
* For **horizontal scaling**, I added multiple instances in **Compute Engine** and additional replicas in **GKE**.
* For **vertical scaling**, I increased the CPU and memory specifications in both Compute Engine and GKE.
* **Performance Testing**: I used **Apache JMeter** and **k6** for load testing. JMeter helped simulate heavy request loads to observe performance, while k6 allowed me to see response times and analyze the behavior of each scaling method. Results showed that horizontal scaling reduced response times more effectively under high traffic, while vertical scaling improved performance for CPU-bound tasks.

Unit testing

Unit tests allow you to check the operation of individual functions or methods of an application. This is the first level of testing, which aims to identify errors within individual components.

Example of a unit test for a Flask application:





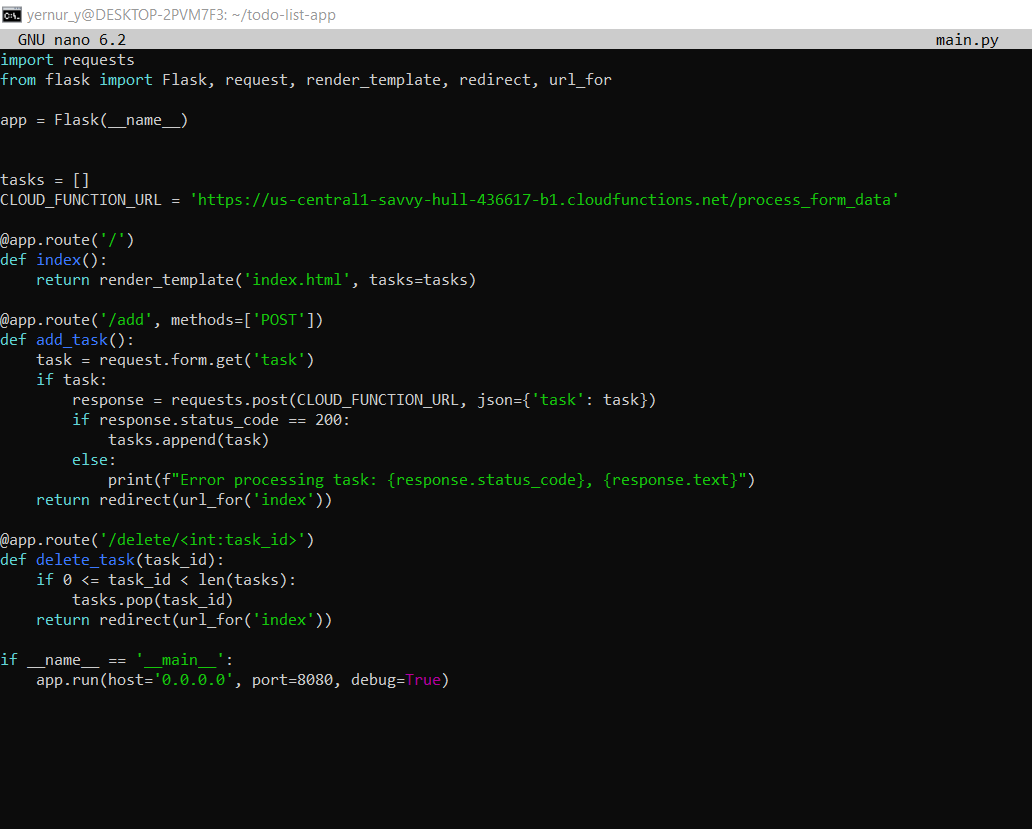
I started the test with the command:

python -m unit test test\_main.py

Integration testing

Integration tests check the interaction of different application components. For example, you can test how the Flask application integrates with Google Cloud Functions.

An example of an integration test for Cloud Function:







This test sends a task processing request to CloudFunction and verifies that the functionreturns a successful response.

Load testing

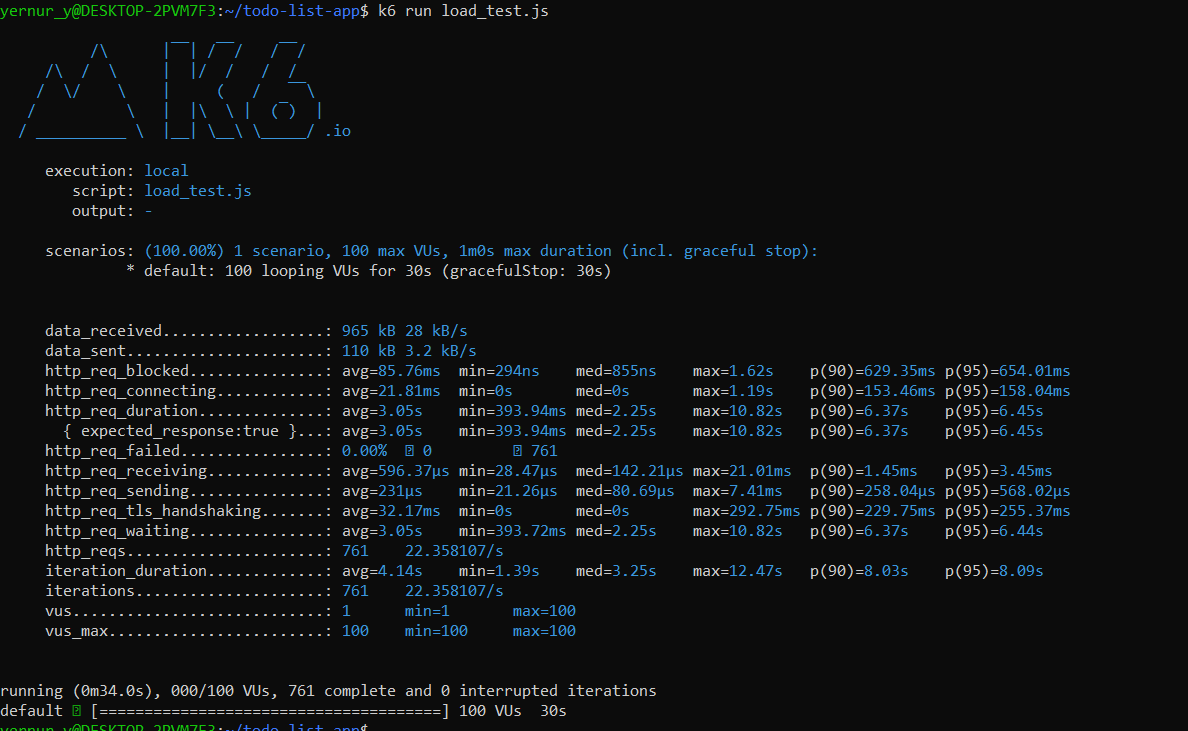
Load testing helps to assess how resilient an application is to high load — for example, when dozens or hundreds of users are accessing at the same time. You can use the k6 tool for such tests.

An example of a load test using k6:



Running a load test:

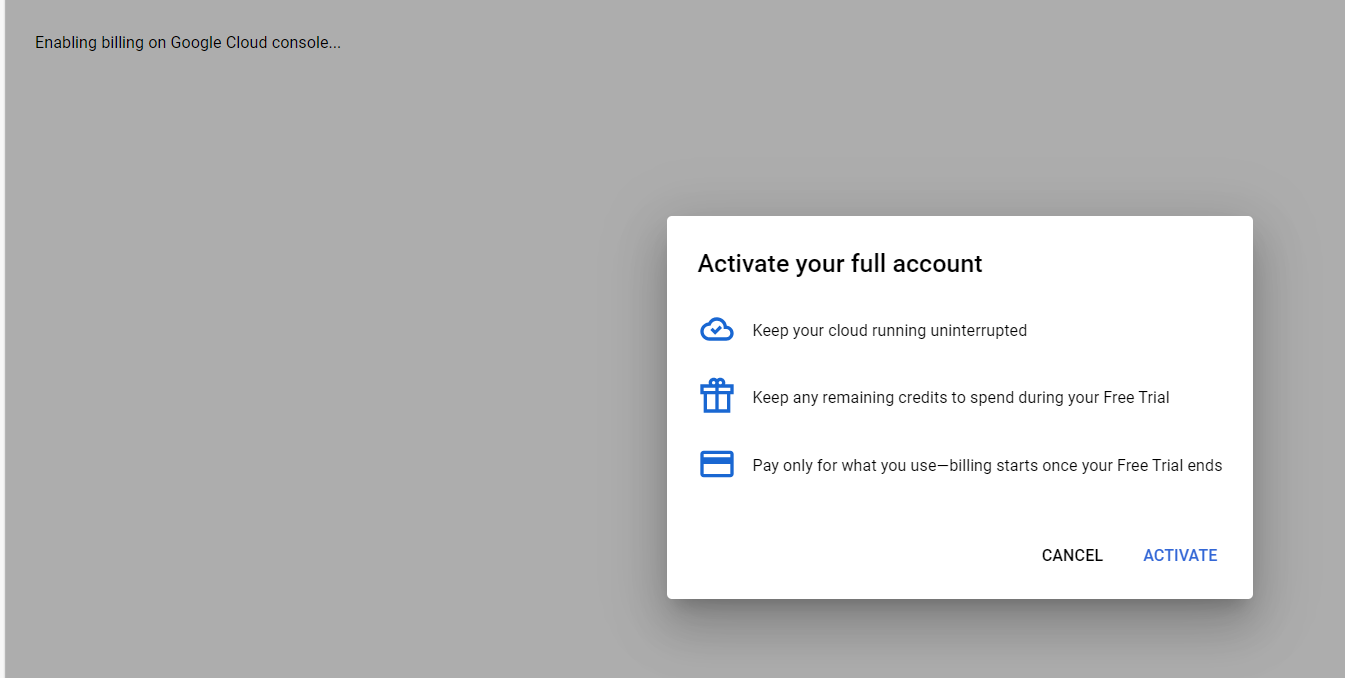
k6 ru load\_test.js



### **Load Balancing**

**Google Cloud Load Balancer Setup**:  
To ensure even traffic distribution, I set up **Google Cloud Load Balancer** across instances. This load balancer handled requests from users and distributed them to healthy instances based on demand.

* **Health Checks**: I configured health checks to monitor each instance’s status, ensuring that traffic was only routed to active instances. This minimized the chances of failed requests due to instance downtime.



### **Auto-Scaling Implementation**

**Auto-Scaling Configuration**:

* In **Compute Engine**, I set up auto-scaling based on CPU utilization and request load. When usage exceeded a set threshold, additional instances were spun up automatically.
* In **GKE**, I configured auto-scaling to adjust the number of replicas based on CPU usage, allowing the application to handle fluctuations in traffic seamlessly.

**Scaling Policies**:  
I implemented scaling policies to define scaling limits and conditions, which helped control the behavior of auto-scaling based on anticipated loads. For instance, during heavy traffic, I allowed auto-scaling to double the instance count but capped it at 10 to avoid excessive costs.

### **Performance Monitoring**

**Using Google Cloud Monitoring**:  
I set up Google Cloud Monitoring to track various metrics, including resource utilization, response times, and error rates. These metrics were displayed on dashboards, making it easy to visualize real-time resource usage and application health.

* **Dashboards**: I created dashboards to monitor CPU usage, memory consumption, and active instances. This allowed me to identify performance bottlenecks quickly and optimize resources based on observed metrics.

### **Cost Optimization Strategies**

**Cost Analysis and Optimization**:  
By analyzing performance metrics, I identified areas where costs could be reduced without compromising scalability.

* **Optimizations Implemented**:
  + Adjusted auto-scaling limits to avoid unnecessary instance spawns during low traffic.
  + Reserved instances for predictable traffic patterns, reducing costs compared to on-demand instances.
  + Optimized Cloud Function usage by eliminating redundant calls and improving code efficiency.
* **Cost Tracking**: I monitored the impact of these optimizations using Google Cloud Billing, observing a decrease in costs while maintaining efficient scaling and application performance.

## **Conclusion**

Through this journey, I enhanced my application's security by implementing robust IAM policies, encrypting data, and integrating security testing into my development process. I also designed a scalable architecture using GKE, favoring horizontal scaling, and set up load balancing and auto-scaling to handle varying workloads efficiently.

By continuously monitoring performance and optimizing costs, I ensured that my application remained reliable, efficient, and cost-effective. Regular reviews and updates to both security protocols and scalability strategies were essential in adapting to evolving needs and technologies.

**Recommendations**

Based on the findings of this report, the following practical suggestions are made to further enhance security and scalability:

1. **Regularly Update IAM Policies**: Continuously review and adjust Identity and Access Management (IAM) roles and permissions to reflect changes in team structure and application requirements. Implement automated tools to detect and remediate overly permissive roles.
2. **Integrate Advanced Security Tools**: Adopt additional security solutions like Web Application Firewalls (WAF) and Intrusion Detection Systems (IDS) to provide layered protection against sophisticated threats.
3. **Implement Infrastructure as Code (IaC)**: Use tools like Terraform or Google Cloud Deployment Manager to manage infrastructure, enabling version control, consistency, and easier scaling.
4. **Enhance Auto-Scaling Strategies**: Refine auto-scaling policies by incorporating more metrics such as memory usage and network I/O. Utilize predictive scaling based on historical data to anticipate demand spikes.
5. **Optimize Cost Management**: Leverage Google Cloud's cost optimization tools to monitor spending. Consider using committed use discounts or sustained use discounts for long-running workloads.
6. **Conduct Regular Security Training**: Provide ongoing training for development and operations teams on security best practices to foster a security-aware culture.
7. **Strengthen Incident Response Plan**: Regularly test and update the incident response plan. Incorporate lessons learned from drills to improve response times and effectiveness.
8. **Adopt Continuous Integration/Continuous Deployment (CI/CD) Best Practices**: Ensure security checks are integral to the CI/CD pipeline, including automated testing, code reviews, and compliance checks.

By implementing these recommendations, the application’s security posture will be fortified, and its scalability will be optimized, ensuring robust performance and resilience in a dynamic cloud environment.

## **References**

* Google Cloud Documentation
* Google Cloud IAM Best Practices
* Google Cloud KMS
* OWASP ZAP Project
* [Snyk Vulnerability Scanner](https://snyk.io)
* Google Kubernetes Engine (GKE)
* Google Cloud Load Balancing
* Google Cloud Monitoring
* [NIST Cybersecurity Framework](https://www.nist.gov/cyberframework)
* [Cloud Security Alliance](https://cloudsecurityalliance.org)

By sharing my experiences and the steps I took, I hope this report provides valuable insights for others looking to enhance their application security and scalability on Google Cloud.

## Appendix: Kubernetes Deployment Configuration

**Deployment Manifest (deployment.yaml)**

apiVersion: apps/v1

kind: Deployment

metadata:

name: to-do-app

spec:

replicas: 3

selector:

matchLabels:

app: to-do-app

template:

metadata:

labels:

app: to-do-app

spec:

containers:

- name: to-do-app-container

image: gcr.io/my-project/to-do-app:v1.0.0

ports:

- containerPort: 8080

resources:

requests:

cpu: "250m"

memory: "512Mi"

limits:

cpu: "500m"

memory: "1Gi"

Explanation: This deployment configuration sets up three replicas of the to-do application, ensuring high availability and load distribution.

**Service Manifest (service.yaml)**

apiVersion: v1

kind: Service

metadata:

name: to-do-app-service

spec:

type: LoadBalancer

selector:

app: to-do-app

ports:

- protocol: TCP

port: 80

targetPort: 8080

Explanation: This service exposes the application on port 80 and forwards traffic to the container's port 8080. Using the LoadBalancer type allows it to receive external traffic.

## **Appendix : Incident Response Plan Outline**

**Incident Response Plan**

1. **Preparation**
   * Assemble an incident response team.
   * Ensure all team members have access to necessary tools and contact information.
2. **Identification**
   * Monitor alerts from Google Cloud Monitoring.
   * Use logs to verify the legitimacy of incidents.
3. **Containment**
   * Short-Term:
     + Isolate affected resources (e.g., disable compromised service accounts).
   * Long-Term:
     + Apply patches and update configurations to prevent recurrence.
4. **Eradication**
   * Remove malicious files or code.
   * Reset credentials and keys if necessary.
5. **Recovery**
   * Restore data from backups.
   * Validate system integrity before resuming operations.
6. **Lessons Learned**
   * Conduct a post-incident review.
   * Update the incident response plan based on findings.

## **Appendix : Sample Monitoring Dashboard**

Description: The dashboard provides real-time metrics on CPU utilization, memory usage, network traffic, and application latency. It highlights thresholds with color coding to quickly identify potential issues.

## **Appendix ; Google Cloud Load Balancer Configuration**

**Creating a Load Balancer via gcloud CLI**

gcloud compute backend-services create to-do-app-backend \

--protocol=HTTP \

--port-name=http \

--health-checks=basic-check \

--global

gcloud compute url-maps create to-do-app-map \

--default-service to-do-app-backend

gcloud compute target-http-proxies create to-do-app-proxy \

--url-map=to-do-app-map

gcloud compute forwarding-rules create to-do-app-forwarding-rule \

--address=0.0.0.0 \

--global \

--target-http-proxy=to-do-app-proxy \

--ports=80

Explanation: These commands set up a global HTTP load balancer that routes traffic to the backend service associated with the to-do application.