CS436 Final Project Report

1. Abstract

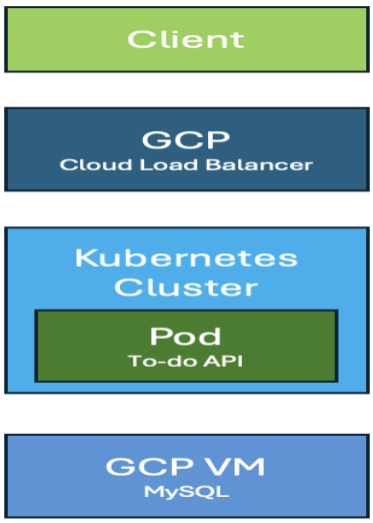
This report shows the design, deployment, and evaluation of a cloud-native to-do web application developed as part of the project. The application is built with Flask and deployed using Google Cloud Platform (GCP) services such as Kubernetes Engine, Cloud Functions, and a Compute Engine VM simulating a database backend. The purpose of the project is to demonstrate practical skills in deploying a containerized application using modern cloud-native tools.

2. Architecture Overview

The architecture illustrates how the system is structured on Google Cloud Platform (GCP). It consists of several main components working together.

At the top of the architecture is the Client, which represents the browser-based frontend interface responsible for sending HTTP requests. These requests are first received by the Cloud Load Balancer, which manages and routes the traffic to the appropriate backend services. The backend logic is hosted on Kubernetes (GKE), where a Flask-based RESTful API runs inside a Pod. This API handles application-level operations and interacts with the database. The database itself is hosted on a GCP VM Instance running MySQL, which stores and manages the relational data required by the system.

2.1 Cloud Architecture Diagram



3. Technologies Used

The system is built using a range of modern technologies to ensure scalability, efficiency, and performance. The core backend is developed with Python using the Flask framework, which provides a lightweight and flexible structure for building RESTful APIs. For deployment, the application is containerized using Docker, and Gunicorn WSGI is used as the production-ready HTTP server to serve the Flask application efficiently.

The infrastructure is hosted on Google Cloud Platform (GCP), leveraging its robust services for cloud deployment. Application orchestration is managed through Google Kubernetes Engine (GKE), which automates deployment, scaling, and management of containerized applications. To ensure the system performs well under load, Locust is used for performance and load testing, enabling the identification of potential bottlenecks and scalability limits.

Component Descriptions and Interactions

The architecture incorporates several core components, each playing a distinct role in the system's functionality. The Client serves as the user-facing interface, sending requests to the backend via a RESTful API. These incoming requests are handled by the Cloud Load Balancer, which distributes the traffic efficiently across various Kubernetes services to ensure reliability and scalability.

The backend logic runs within a Kubernetes Cluster (GKE), where a containerized Flask application—referred to as the flasky backend—hosts and serves the necessary API endpoints. For persistent data storage, the system utilizes a VM Instance running MySQL, which replaces the initial SQLite setup with a production-grade MySQL server deployed on a Google Compute Engine virtual machine.

Additionally, Cloud Storage and Cloud Functions are integrated optionally, providing support for file uploads or asynchronous background processing tasks that may be required in future development stages.

4. Deployment Workflow

The deployment workflow follows a structured process to ensure the Flask-based application is properly containerized, deployed, and made accessible in a secure and scalable manner on Google Cloud Platform.

The first step is Dockerization, where the Flask application is containerized using a custom `Dockerfile` along with a startup script (`boot.sh`). Once the container is built, the image is pushed to Google Container Registry, making it accessible for use within GCP services.

Next, a Google Kubernetes Engine (GKE) cluster is created through GCP. The containerized application is then deployed to the cluster using Kubernetes YAML configuration files, including Deployment and Service definitions.

For data persistence, a MySQL database is installed and configured on a GCP virtual machine. The Flask application is set up to connect to this database using proper connection strings to ensure reliable communication.

To expose the application to the internet, a Kubernetes Service with an external IP is configured. Port 80 and 443 are opened through firewall rules to allow HTTP and HTTPS traffic.

Finally, security and networking settings are established. This involves configuring GCP firewall rules to permit HTTP access and verifying that all necessary services are accessible via their assigned public IP addresses, ensuring the deployment is both functional and secure.

5. Application Overview

The application provides essential To-Do functionality along with placeholder pages designed for future development, such as user authentication and task submission.

The key pages include the home page ("/"), which serves as the landing interface for users. The login page ("/login") is a mockup, acting as a placeholder for future user authentication features. The task creation page ("/new") contains a form input allowing users to submit new tasks to the system.

In addition to these core pages, the application includes a Cloud Function endpoint that supports both GET and POST methods. This endpoint is currently configured to return a dummy email confirmation, simulating backend processing for tasks like user registration or notification services.

6. Performance Evaluation

We performed a performance test using Locust on our Flask-based web service, which is hosted at (http://34.122.56.78:5000). The purpose of this test was to evaluate how the system behaves when multiple users access it at the same time and to observe if any performance issues or bottlenecks arise.

Test Summary:
During the test, a total of 1190 GET requests were made to the home page (`/`). The system was able to handle an average of 19.5 requests per second (RPS), with a failure rate of 1% , which means that only a very small portion of the requests failed. At the time of the report, the current failure rate was 0, and the system continued to serve requests at 19.5 RPS.

Although most of the response times were acceptable, it was observed that the maximum response time reached up to 21 seconds in some cases. This suggests that while the system is generally stable under load, there may be occasional slowdowns when many users access it at once. These outliers could be examined further to identify potential causes and improvements.

7. Challenges

During deployment, we encountered several issues that required debugging and configuration adjustments.

Firstly, Gunicorn compatibility needed special attention. The server required explicit binding to 0.0.0.0 to ensure it could accept external connections, and response buffering had to be disabled to allow real-time data flow from the Flask application.

Secondly, a reference to a missing file caused deployment failures. The original deployment script pointed to a non-existent manage.py file, which led to pod crashes within the Kubernetes cluster. Removing or correcting this reference was necessary for stable deployment.

Lastly, there was a mismatch in the Cloud Function entry point. The initial main.py handler c onfiguration did not align with the expected structure, which delayed the deployment of th e cloud function until the correct handler was specified. These issues highlight the importa nce of aligning configuration files with the actual

8. Future Work

There are some improvements planned for the future to make the application better and more secure. First, the database will be moved to a real Cloud SQL instance to store data in a more reliable way. Next, user login and session handling will be added so that users can securely sign in and use the app. To make the app faster, Redis caching will be used to reduce the load on the database. Lastly, HTTPS will be set up using Ingressand TLS certificatesto keep all data safe while it's being transferred between users and the server.

9. Conclusion

This project successfully demonstrates containerized application deployment with Kubernetes and GCP tools. Although the functionality is basic, the system exhibits cloud-native best practices such as containerization, separation of concerns, autoscaling potential, and external service integration.

Demo Video Walkthrough

- Show terminal output of GKE pods/services
- Access app via browser
- Trigger Cloud Function endpoint

External IP: http://34.46.145.252