Introduction:

We designed a microservice-based e-commerce platform application and deployed it using serverless and serverful methods. Next, we measured different metrics (response time and CPU usage) to compare the deployments.

Design:

Our application consists of four microservices. The *User* microservice is responsible for managing the users and authentication. The *Order* microservice contains customers' orders and carts. The next microservice, *Financial,* takes care of accounts and invoices, and finally, the *Inventory* microservice tracks product stocks. Each Microservice uses its own database.

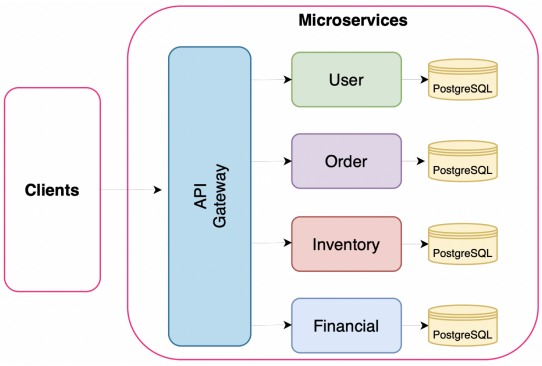

Figure 1: System Overview

Two of the API calls that engage multiple microservices are as follows:

- *add_item,* in which *Order* receives the request from the client, calls *User* for authentication and then checks with *Inventory* for availability and price
- *checkout,* in which *Order* receives the request from the client, authenticates with *User*, and sends a *payment* request to the *Financial* service.

Implementation:

Serverful: We implemented our microservices using Python, and Django Framework, and used PostgreSQL as the backing data store. Each microservice was containerized using Docker and deployed on a managed Kubernetes platform, which is discussed further below. The microservices use synchronous HTTP requests to invoke REST APIs to communicate with each other. All functionalities discussed previously during the design phase were implemented in the microservices. Th implementation could be accessed through Github.

Serverless: For our system to work serverless, we had to modify it to be suitable for the functional notion of serverless services. We have fully implemented the functions related to the APIs of our services. This implementation is not using Django anymore, as the functions are minimal and act as a single API, so there is no need for a full web framework. In fact, serverless services like Google Cloud Functions provide us with the required tools, such as Flask, to manage requests and responses. Like the serverful implementation, the serverless implementation uses separate databases for each service to comply with microservice

architecture. The full implementation of serverless functions can be accessed through Github.

Deployment:

Serverful: Our application was deployed on GCP's managed Kubernetes platform, GKE, with each service deployed through a separate deployment manifest. These services were accessible via a ClusterIP service and connected to a mono-pod statefulset of PostgreSQL database through a headless service. Also, to enable secure and efficient communication between our microservices and external clients, we relied on an API Gateway that was integrated with the ingress resources of K8S. This API Gateway technology provided a scalable and customizable solution, acting as a single entry point for external clients to access our microservices.

Moreover, For easier deployment and upgrading of the app, we implemented a Helm chart for the application. Before that, for every change in each service, we should have changed its deployment manifest and redeployed the service itself, which was quite distracting with a high risk for mistakes. By deploying our app with a Helm chart, all the necessary manifests are generated and deployed via a single yaml file, reducing the risk of error.

Furthermore, we utilized the managed monitoring of GCP for our managed Kubernetes cluster and developed some dashboards for monitoring the performance metrics of each service. This helps us reduce the work needed for monitoring our services and improves the reliability and accuracy of our monitoring dashboards.

Serverless: We have used Google Cloud Functions to deploy our serverless functions. Behind the scenes, Google Cloud Functions automatically associates a Cloud Run deployment to each function to set it up and running. On the other hand, we have used Google Cloud SQL to get our PostgreSQL databases up and running. One of the challenging parts in the deployment of our serverless program has been the connection between the functions and the databases, as they needed quite a few configurations done in the Google Cloud Console. Finally, we have leveraged the Google Cloud API Gateway to manage the serverless APIs and the requests to and between them.

Results and measurements:

We have used Locust and defined different scenarios to generate statistics required for comparing serverful and serverless response times. One of the most complicated scenarios to involve all the services is the following: (1) Create a user, (2) Get the list of products, (3) Add to cart and (4) Checkout.

By defining 100 users with a spawn rate of 1u/s sending the above requests sequentially to both the serverless and serverful applications, we got the following charts:

Figure 2: User rate and response time of serverful deployment

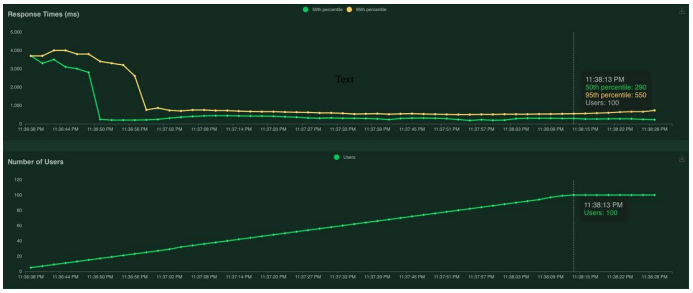

Figure 3: User rate and response time of serverless deployment

The comparison of figures 2 and 3 demonstrates that as the number of clients and client requests increase, the response time of the serverful version of the application also increases. However, the response time of the serverless deployment remains fixed due to the automatic scaling of resources as the request rate increases. The serverless deployment does experience a cold start at the beginning of the test, which is the time a serverless function takes to start running when it is invoked for the first time or after a certain inactivity period.

As depicted in figures 4-5-6 of backend service monitoring, there was a sudden surge in CPU usage during load testing across all three services. However, the "Users" service experienced a much higher increase in CPU usage compared to the "Financial" and "Products" services, causing it to reach its limit and become a bottleneck for our system. Despite the under-utilization of the other two services, the overconsumption of CPU resources by the "Users" service hindered the proper request rate of the system. Addressing this bottleneck would require extensive tuning and redeploying efforts.

Alternatively, by adopting a serverless deployment approach, we can overcome the need for such adjustments as the service provider offers elasticity and auto-scaling of separate APIs, thus eliminating the need for extensive tuning and redeploying over time.

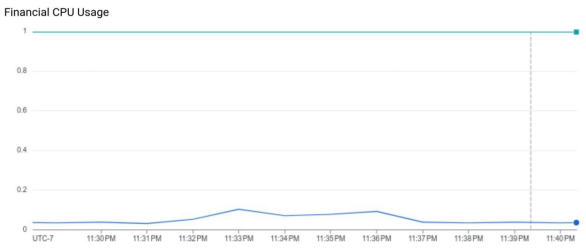

Figure 4: CPU usage of financial service

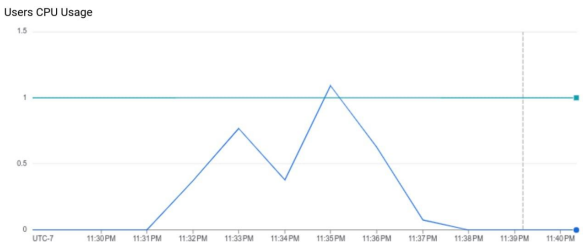

Figure 5: CPU usage of user service

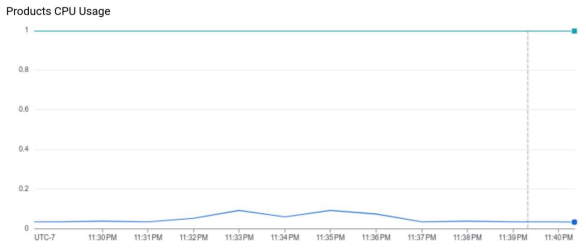

Figure 6: CPU usage of products service