CS 436 – Cloud Computing

Final Report

Project Title: Instagram Clone Project Deployed on Google Cloud

Group Members:

Baturalp Öztürk 26499

Emir Çolakbüyük 29477

Ulaş Noyan 27904

Yunus Tan Kerestecioğlu 28168

Date: 26.05.2024

1. ABSTRACT

This report details the development and implementation of a cloud-native architecture on Google Cloud Platform (GCP). The project aimed to design a robust backend system, plan the cloud architecture in advance, and deploy the solution on GCP. The process included backend selection, comprehensive architectural design, deployment, and extensive testing to evaluate the system's performance. The results demonstrate the efficiency and scalability of the implemented cloud-native solution.

2. INTRODUCTION

Cloud computing has become a cornerstone for modern businesses, enabling them to deploy, manage, and scale applications more efficiently and cost-effectively. The flexibility and scalability of cloud services allow companies to adapt quickly to changing market demands without the need for significant upfront investments in hardware and infrastructure. Cloud service providers became popular overtime, offering a range of services that cater to various business needs, from storage and computing power to advanced analytics and machine learning, making it a budget friendly option. Additionally, cloud computing reduces the responsibilities of IT staff, allowing them to focus on strategic initiatives rather than routine maintenance tasks.

This project was undertaken to create a cloud-native architecture and implement it on Google Cloud Platform (GCP). The primary goals were:

Backend Selection: Choose a backend technology that aligns with cloud-native principles and supports efficient scaling and performance.

Cloud Architecture Design: Plan and design a robust cloud architecture that ensures high availability, fault tolerance, and security.

Deployment: Implement all necessary steps for deploying the application on GCP.

Testing and Performance Evaluation: Conduct extensive tests to evaluate the system's performance, scalability, and reliability.

A budget limit of $300, provided free by GCP, was also set for the project. This constraint required careful planning and optimization to ensure that the deployment remained within budget while meeting all project objectives.

The significance of this project lies in its potential to demonstrate the practical application of cloud-native principles on GCP, providing a scalable and efficient solution for modern business needs.

3. TECHNOLOGY STACK

The project utilized a range of modern technologies and tools to achieve the desired cloud-native architecture. The following components were used for the development and deployment process:

Front-end: Next.js

Next.js was chosen for its powerful features such as server-side rendering, static site generation, and built-in support for API routes. Its seamless integration with React made it an ideal choice for building a dynamic and responsive user interface.

Back-end: Node.js and Express.js

Node.js was selected for its non-blocking, event-driven architecture, which is well-suited for scalable and high-performance applications. It facilitated the development of a robust backend capable of handling concurrent requests efficiently. For a social media backend, Node.js is particularly advantageous due to its ability to handle a large number of simultaneous connections with low latency. The event-driven nature of Node.js is ideal for real-time applications, such as notifications, live updates, and chat functionalities, which are critical features in social media platforms.

Express.js, a minimal and flexible Node.js web application framework, was used to build the RESTful APIs. Express simplifies the development process by providing a robust set of features for web and mobile applications. Its middleware capabilities and HTTP utility methods made it easy to create a highly efficient and organized backend.

Database: MongoDB

MongoDB was used for its flexibility and scalability. Its document-oriented structure allowed for easy storage and retrieval of data, making it an excellent choice for a cloud-native environment. MongoDB's schema-less design is particularly beneficial for handling the diverse and rapidly changing data typical of social media applications. The combination of MongoDB with Node.js and Express.js (commonly referred to as the MERN stack when used with React) allows for seamless data flow and handling, providing a powerful and efficient solution for the backend infrastructure.

Containerization: Docker

Docker was employed to containerize the application, ensuring consistency across different environments and simplifying the deployment process. Docker's containerization capabilities allowed for easy scaling and management of the application.

Deployment: Google Cloud Platform (GCP)

Google Cloud Platform provided the infrastructure and services needed to deploy and manage the application. These services will be provided in the Cloud Architecture part of the report.

4. CLOUD ARCHITECTURE

The cloud architecture for this project was designed to ensure high availability, scalability, and efficient management of resources. The architecture consisted of the following key components:

Cloud Architecture Components:

Virtual Machines (VMs):

Backend VM: This VM had Node.js installed to handle the backend logic of our social media application. Node.js was chosen for its scalability and ability to handle multiple concurrent connections efficiently. Our backend is capable of managing essential social media functionalities, including user logins, following other users, creating and deleting user accounts, viewing user information, and handling comments. This robust set of features ensures a comprehensive and interactive user experience.

Database VM: This VM had a local instance of MongoDB installed. MongoDB was connected to the backend VM, enabling seamless updates and retrieval of data by the Node.js application. The database is designed to handle instances of users and comments efficiently. MongoDB's document-oriented structure allows it to store complex data types, making it ideal for managing user profiles, relationships (such as followers), and comment threads. Additionally, MongoDB's powerful querying capabilities facilitate rapid retrieval and updates of user and comment data, ensuring that the backend can respond quickly to user interactions.

Frontend VM: This VM was responsible for running the front-end application built with Next.js, allowing users to interact with the features of our social media backend. We used a simple frontend to display and utilize the features provided by the backend, ensuring a user-friendly interface for interacting with the application.

Containerization with Docker

The backend was containerized using Docker, which provided several benefits. Containerization ensures consistency across different environments, simplifies the deployment process, and enables efficient resource utilization. Docker containers encapsulate the application and its dependencies, making it easier to manage and scale the backend services.

Deployment to Google Cloud Run

The containerized backend was deployed to Google Cloud Run using Artifact Registry. Google Cloud Run was selected for its ability to automatically scale the application based on incoming traffic, ensuring that the application can handle varying loads efficiently. Cloud Run also provides serverless execution, reducing the overhead of managing infrastructure.
Cloud Run was fortified with Identity and Access Management (IAM) authentication, ensuring secure access and management of the deployed services.

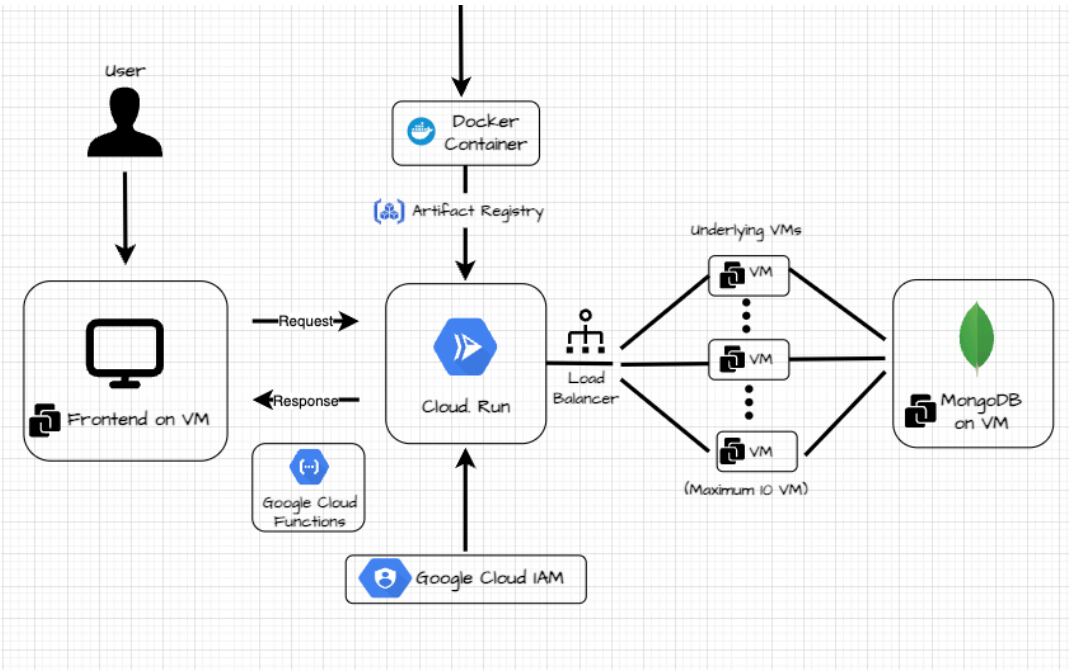
Load Balancer

The load balancer provided by Cloud Run was used to distribute incoming user traffic across multiple instances of the backend service. The load balancer ensures high availability and reliability by directing traffic to healthy instances, improving the user experience by reducing latency and preventing overload on any single instance.

Cloud Functions

Cloud Functions were integrated into the architecture to perform specific tasks such as comment censorship and text length checks. These functions are triggered by events, allowing for efficient and scalable execution of these tasks without impacting the performance of the main application.

This cloud architecture leverages the strengths of Google Cloud Platform, including its robust infrastructure, scalability, and comprehensive suite of services, to deliver a high-performing and reliable social media application.



Flow of the program:

User Interaction: Users interact with the front-end application hosted on a VM. This frontend sends requests to the backend to access various features of the social media application.

Backend Processing: The backend, running on a Node.js VM, processes these requests. It performs operations such as user authentication, user management, and comment handling.

Load Balancer: The load balancer provided by Cloud Run distributes incoming traffic across multiple instances of the backend, ensuring high availability and reliability.

Cloud Run Auto-scaling: Google Cloud Run automatically scales the containerized backend application based on incoming traffic, ensuring efficient handling of varying loads.

Database Operations: The backend communicates with the MongoDB instance hosted on a separate VM to retrieve and update user and comment data. This setup ensures that the application can efficiently manage and store data.

Cloud Functions: Specific tasks, such as comment censorship and text length checks, are handled by Cloud Functions. These functions are triggered by events and run independently.

Response: The processed data is then sent back through the same path to the front-end VM, which displays the response to the user.

5. TEST RESULTS

Experiment Design

The experiment was designed to actively test the cloud-native architecture's performance, scalability, and reliability. The primary focus was to simulate real-world usage scenarios and measure the system's response under varying loads. Key aspects of the experiment design included:

Scenario Definition: Common user interactions such as logging in, creating accounts, following users, posting comments, and viewing profiles were defined as test scenarios.

Load Simulation: Different levels of concurrent user activity were simulated to evaluate the system's ability to handle increasing loads.

System Parameters

The following system parameters were monitored during the experiments:

Response Time: The time taken by the system to respond to user requests.

Throughput: The number of requests processed per second.

CPU and Memory Usage: The resource utilization of VMs and containers.

Database Performance: Read and write latency for MongoDB operations.

Error Rate: The number and type of errors encountered during testing.

Auto-scaling Events: The number and timing of instances where Cloud Run auto-scaled the backend services.

Experimentation Tool

To conduct the experiments, we used the following tools:

Locust: For simulating user load and generating traffic to test the performance and scalability of the system. Different types of Locust tests were used with varying numbers of total active users and different rates of user increase for our scenarios.

Google Cloud Monitoring: For real-time monitoring of system parameters such as CPU and memory usage, auto-scaling events, and error rates.

Effective Stress Testing

Effective stress testing was conducted to push the system to its limits and identify potential bottlenecks. This involved:

Gradual Load Increase: Starting with a small number of users and gradually increasing to thousands of concurrent users to observe how the system scales.

Peak Load Simulation: Simulating sudden spikes in traffic to test the system's ability to handle peak loads.

Sustained Load Testing: Running the system under high load for extended periods to test long-term stability and resource utilization.

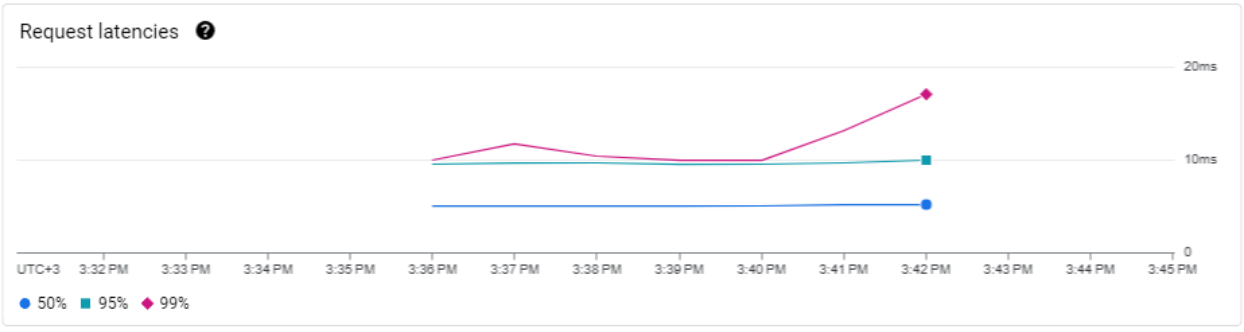
Locust Load Testing:

Test 1: For the test with a maximum of 1,000 users, the system showed a gradual increase in the number of users and handled the load efficiently with consistent response times and a high request rate. The maximum number of container instances were set to 5 and the concurrent response handling was set to 70

Test 2: For the test with a maximum of 9500 users, the system maintained high request rates with minimal failures. The response times increased slightly during peak loads but remained within acceptable limits. Auto-scaling was evident as more instances were created to handle the increased load.The maximum number of container instances were set to 10 and the concurrent response handling was set to 80

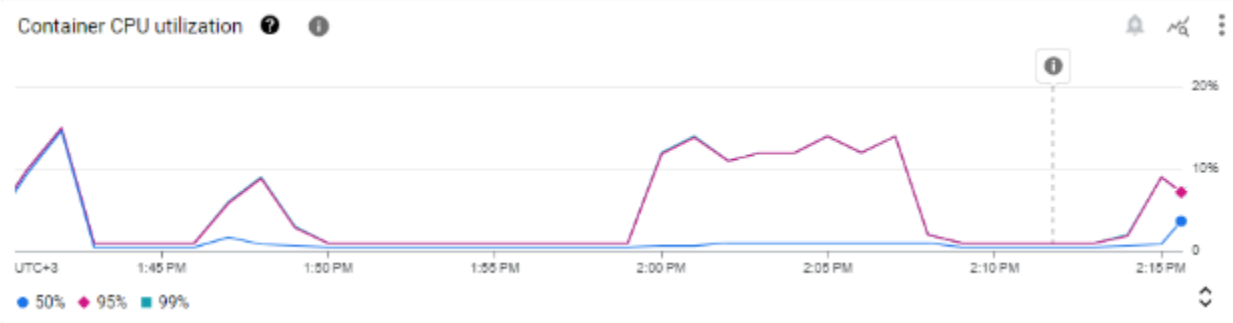
Results and Analysis

The results of the stress testing indicated that:

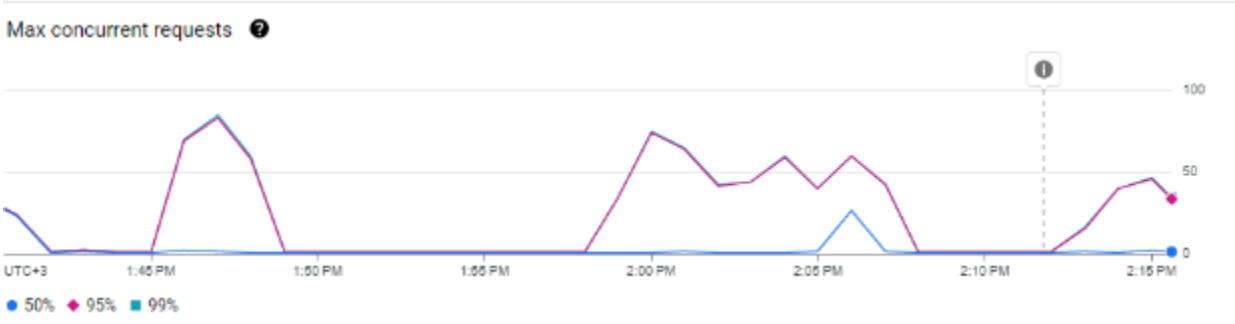


Request Latencies: The request latencies remained relatively low and stable under different load conditions, indicating efficient handling of user requests by the backend.

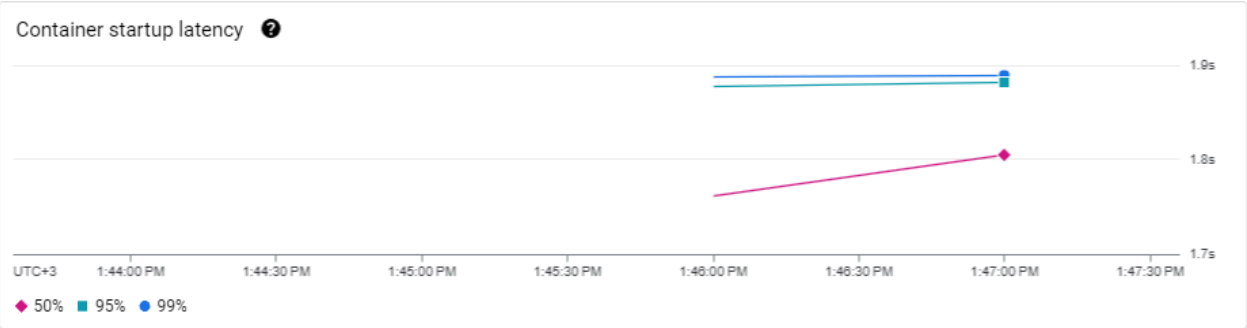
Container Instance Count: The number of container instances fluctuated according to the load, demonstrating the auto-scaling capabilities of Google Cloud Run. As user load increased, additional instances were automatically created to handle the increased traffic, and as the load decreased, instances were scaled down.



Container CPU Utilization: CPU utilization varied with the load, peaking during high traffic and remaining low during idle times, showing efficient resource management.



Max Concurrent Requests: The system handled a large number of concurrent requests effectively, with the load balancer distributing traffic across multiple instances.



Container Startup Latency: The startup latency for containers was minimal, ensuring quick scaling and availability of new instances.

6. SUMMARY

This project successfully demonstrated the development and implementation of a cloud-native architecture on Google Cloud Platform (GCP) for a social media application. By leveraging modern technologies such as Next.js, Node.js, Express.js, MongoDB, Docker, and various GCP services, we were able to create a scalable, efficient, and robust solution.

The architecture included three key VMs: one for the backend with Node.js, one for the local MongoDB database, and one for the frontend with Next.js. The backend and database VMs were interconnected, allowing for seamless data updates and retrieval. The backend was containerized using Docker and deployed to Google Cloud Run, which provided automatic scaling and serverless execution, reducing infrastructure management overhead.

Additionally, a load balancer ensured high availability and reliability by distributing incoming traffic across multiple backend instances. Cloud Functions were integrated to handle specific tasks such as comment censorship and text length checks, ensuring efficient and scalable execution.

Overall, the project demonstrated the practical application of cloud-native principles, providing a high-performing and reliable social media application that leverages the full benefits of cloud computing on GCP.

7. DISCUSSION

In the course of configuring Cloud Run to meet our project's requirements efficiently, we encountered significant challenges with the default settings. Initially, the performance was suboptimal, which necessitated extensive testing and adjustments. We experimented with various configurations, focusing on concurrency, the maximum number of instances, and the allocation of RAM and CPU resources. Each change required thorough evaluation to ensure that our adjustments were leading to improved performance without compromising other aspects of our application.

Additionally, to verify and showcase the effectiveness of our backend updates, we developed a simpler frontend interface. This frontend played a crucial role in our testing process, allowing us to quickly and clearly observe the impact of our backend modifications.

Despite these challenges, through our dedicated efforts and continuous testing, we successfully developed a cloud architecture capable of handling up to 10,000 users with low latency and seamless performance. By leveraging auto-scaling capabilities, our system can dynamically adjust to varying loads, ensuring that it remains responsive and efficient under high traffic conditions.

The implementation of auto-scaling was particularly crucial in managing resources effectively. It allowed our architecture to scale up during peak usage times, ensuring that user requests were handled promptly without any noticeable delay. Conversely, during periods of lower activity, the system could scale down, optimizing resource usage and reducing costs. This adaptability was key to maintaining a high level of performance and reliability.

Moreover, our focus on low latency ensured that users experienced minimal delays, enhancing the overall user experience. We achieved this by carefully configuring the backend and frontend interactions, optimizing our database queries, and ensuring efficient communication between services. The result was a smooth and responsive application that could meet the demands of a large user base without compromising on speed or performance.

Overall, the combination of auto-scaling, low latency, and efficient resource management allowed us to create a robust cloud architecture. This architecture not only supports a substantial number of users but also adapts to changing demands, providing a reliable and high-performing platform for our application.