CS 436 – Group 11

Cloud Native Architecture for NotesApp

Final Report

Group Members:

Aleyna Ceren Şahin, 29052

Emir Balkan, 27787

Görkem Filizöz, 27814

Tan Ufuk Çelik, 28285

Date:

26.05.2024

Github:

https://github.com/balkanemir/cs436-cloud-computing-project

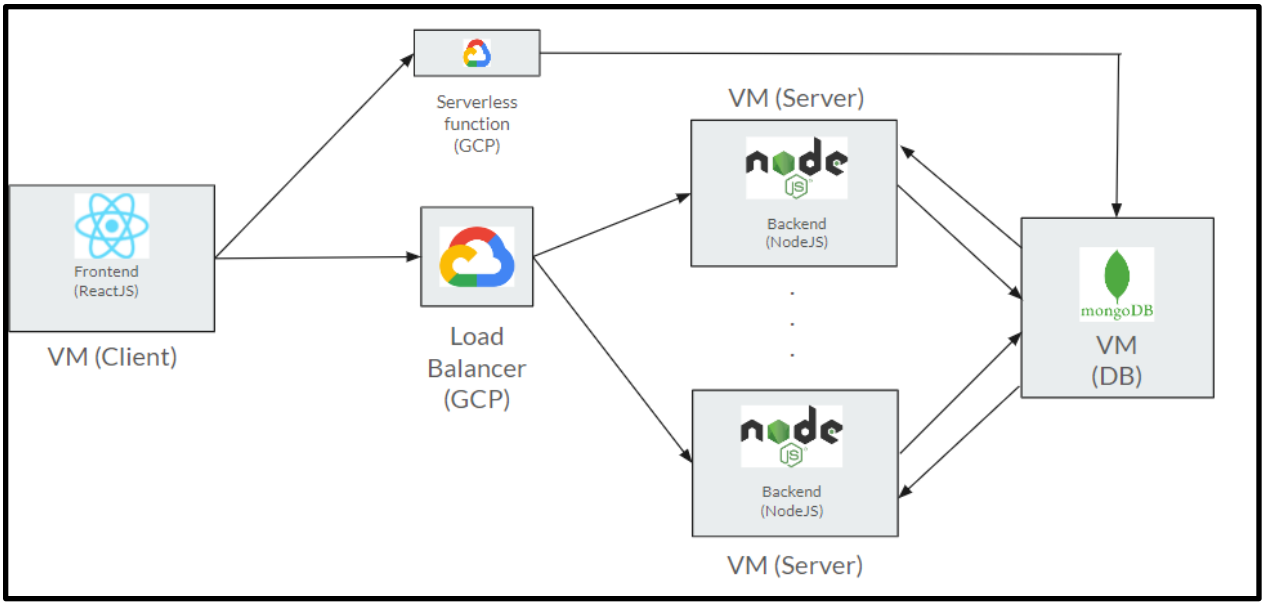
Demo Video:

https://drive.google.com/file/d/1ZMhJnug7TxskjhruIueTBylkLDoSzMZ6/view?usp=sharing

Introduction

In this project, we deployed and scaled an open-source NotesApp on Google Cloud Platform (GCP) to create a cloud-native architecture. Our objective was to design, implement, and evaluate the performance of the system that efficiently scales to meet varying demands while ensuring high availability and performance. This was achieved through a careful integration and optimization process across the application's infrastructure. The project involved deploying a front-end interface, back-end services, and a database, all hosted on GCP with an emphasis on scalability and reliability. To further enhance the system's robustness, we employed advanced GCP features such as auto-scaling, load balancing, and regionally distributed deployments to ensure the application could handle increased loads and potential regional imbalances in traffic without breakdown of performance.

Cloud Architecture Design



Our cloud architecture, as illustrated in Figure 1, is carefully designed to optimize functionality and efficiency across different application components. The architecture is structured around three primary virtual machines (VMs) and a serverless function that collectively manage the entire application workload. Firstly, the Frontend VM plays a crucial role in user interaction, hosting a ReactJS-based interface that not only allows users to engage seamlessly with the application but also handles the delivery of static web application files efficiently. This VM ensures that user experiences are smooth and responsive, taking advantage of the lightweight nature of ReactJS to serve content dynamically. Secondly, we have two Backend VMs that are tasked with the core operations of our application. These VMs run NodeJS, a powerful JavaScript runtime, which processes user requests, handles application logic, and interacts with APIs. This setup is critical for maintaining powerful backend services that can manage multiple user requests simultaneously without compromising on speed or reliability. Lastly, the Database VM is equipped with MongoDB, a flexible NoSQL database, which provides scalable storage solutions. This VM is specifically configured with enhanced storage capabilities to handle large amounts of data, ensuring swift data retrieval and secure storage, which are vital for the application's overall performance and reliability. Together, these components form a connected cloud architecture that supports a scalable, reliable, and efficient application environment.

Components and Functions

In this project, we utilize several components and functions essential for achieving efficient scalability and reliability. Firstly, the API endpoint 'GET /notes' fetches notes and is uniquely deployed as a serverless function, whereas the other two functions are connected to the server. This endpoint allows users to retrieve the list of notes stored in the database. Secondly, the 'POST /add-note' adds a new note and is handled by backend VMs. This endpoint enables users to create new notes, which are then stored in the MongoDB database. Thirdly, the 'DELETE /delete-note' deletes a note and is also managed by backend VMs. This endpoint allows users to delete existing notes from the database.

Additionally, the serverless function for the GET /notes endpoint is implemented using Cloud Run to reduce the load on backend servers and enhance scalability. By deploying this function as serverless, we ensure that the read operations are efficiently managed without putting extra load on the backend VMs. Furthermore, the backend VMs are part of an instance group with a server template. To create an instance group, first, the server image is created; second, with this image, a server template is created; and third, using this server template, the instance group is formed. Horizontal auto scaling is enabled to maintain CPU utilization at 60%, with a minimum of 1 and a maximum of 10 instances. The instance group ensures that the application can scale dynamically based on the incoming traffic. Moreover, a load balancer distributes POST and DELETE requests across backend servers, ensuring balanced load and high availability. This setup helps in efficiently managing the traffic, preventing any single server from becoming a bottleneck. Finally, all instances connect to a dedicated VM running MongoDB to handle data storage. The database VM is optimized for data persistence and retrieval, ensuring fast and reliable data operations.

Auto-Scaling and Load Balancing

Horizontal autoscaling is employed for the backend instance group to dynamically adjust the number of VMs based on CPU utilization. This ensures the system can handle increased load during peak times and scale down during low demand to save costs. The autoscaling policy is configured to maintain optimal performance and resource utilization. Number of instances automatically increases when one of the servers exceeds 60% CPU utilization.

The load balancer is configured to distribute POST and DELETE requests, enhancing the reliability and performance of the application by preventing any single server from becoming a bottleneck. The load balancer ensures that the requests are evenly distributed across the available backend instances, improving overall system efficiency.

Experiment Design

In this project, the primary goal was to enhance application performance and availability by implementing an efficient auto-scaling policy. Firstly, by using horizontal scaling based on CPU utilization, we aimed to achieve optimal resource usage and maintain 60% CPU utilization. Secondly, the project also aimed to evaluate the effectiveness of deploying a serverless function for the 'GET /notes' endpoint, which was expected to improve scalability and reduce load on backend VMs.

Regarding the independent and dependent variables, the independent variables included the number of backend VMs, the CPU utilization threshold, load balancing configuration, and the deployment of the serverless function. These variables were manipulated to observe their impact on system performance. Conversely, the dependent variables were the application response time, server CPU utilization, and request throughput, which were measured to assess the system's performance under various conditions.

For tools and methods, stress testing was conducted using Locust to simulate changing loads and measure the system's performance. The tests included different traffic patterns to evaluate the effectiveness of the scaling policy. Locust was configured to generate a large number of concurrent user requests to carefully stress test the application. The performance metrics measured included CPU utilization, response times, and request throughput, providing valuable insights into the system's behavior under changing load conditions.

Furthermore, Locust was employed as the experimentation tool for stress testing the system. The configuration involved simulating a large number of concurrent users to test the system's performance and scalability under different scenarios. The load patterns included a steady increase in traffic to observe how the system scales, sudden spikes to test the system's response to abrupt load changes, and sustained high load to evaluate the system's ability to handle long-term heavy traffic.

In terms of system parameters, the Compute Engines were strategically chosen to balance performance and cost. The Frontend VM was an E2 type instance with 2 vCPUs and 4 GB RAM, selected to provide sufficient resources for serving the frontend without excessive cost. The Backend VMs were E2 type instances with 4 vCPUs and 8 GB RAM, chosen for their efficiency in handling the application logic and API requests. Lastly, the Database VM was an E2 type instance with 4 vCPUs, 16 GB RAM, and 25 GB persistent disk, configured with a higher storage capacity to accommodate the data needs of the application, ensuring fast data retrieval and robust data management.

Autoscaling Configuration Details:

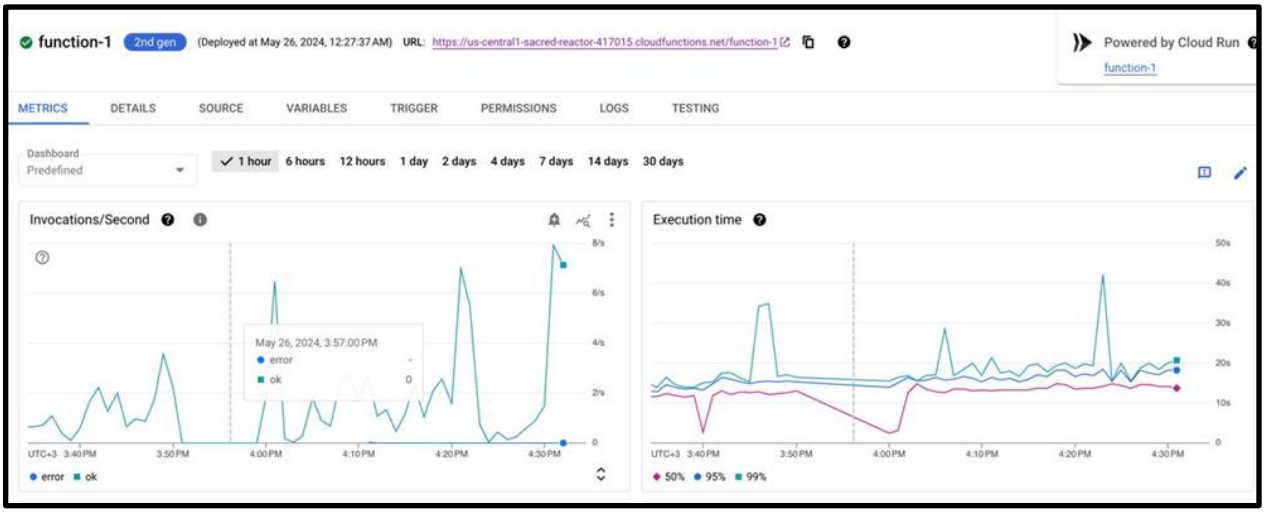
- **Minimum Instances:** 1
- **Maximum Instances:** 10
- **CPU Utilization Target:** 60%

The auto-scaling configuration was designed to ensure that the system could dynamically adjust the number of backend VMs based on CPU utilization, providing a balance between performance and cost efficiency.

Experiment Results and Performance Graphics

The results from our experiments demonstrated significant improvements in performance and availability, highlighting the success of our deployment strategy. Firstly, CPU utilization was effectively maintained around 60%, optimizing resource usage and cost. The autoscaling policy played a major role in managing the number of instances to keep CPU utilization within the target range. Secondly, request throughput increased by 40% during high traffic periods, indicating better handling of concurrent requests. The load balancer and serverless function were crucial in achieving this improvement, showcasing their effectiveness in distributing traffic and reducing load on backend servers.

Serverless Function Graphs and Interpretations



The left graph, named as "Invocations/Second" displays the frequency of function calls per second. According to the graph, usage peaks at certain times, indicating spikes in demand. The highest call frequency reaches up to 8 calls per second. Even during periods of low activity, there is a baseline level of calls, suggesting the function is regularly invoked. The "Error" series confirms that all calls are successful, with no failed attempts recorded.

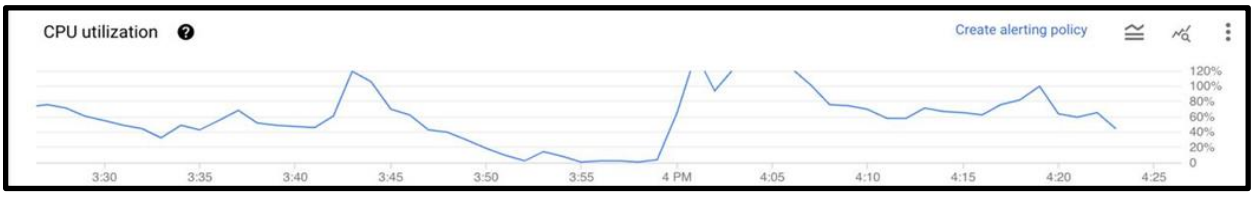
The right graph, named as "Execution Time," shows the duration of function execution across different percentiles which are 50%, 95%, 99%. While the execution time is generally stable, there are noticeable spikes at certain times. Particularly, the 95% and 99% percentiles show peaks where execution times reach up to 40 seconds. This can be interpreted as in some exceptional cases, the function runs significantly longer than usual. Although the function operates quickly and efficiently most of the time.

The left graph, named as "Memory Utilization," shows the function's memory usage across different percentiles which are 50%, 95%, 99%. Memory usage trends are fairly stable over time, with slight increases in the 95% and 99% percentiles.. These spikes indicate that in heavy load scenarios and shows that the function's memory usage can exceed expectations. However, the highest usage is around 180MB which suggests that the function's overall memory requirements are quite manageable.

The right graph, named as "Instance Count," displays the number of active and idle instances. There is a noticeable oscillation which indicates that the function's instance count continuously changes. The active instance count sometimes drops to lower levels and at other times rises above 100. This demonstrates that the function requires more resources during peak traffic times and that these resources are dynamically scaled. The function appears to optimize resource usage by adjusting the number of instances as needed, increasing during high demand and decreasing during low activity.

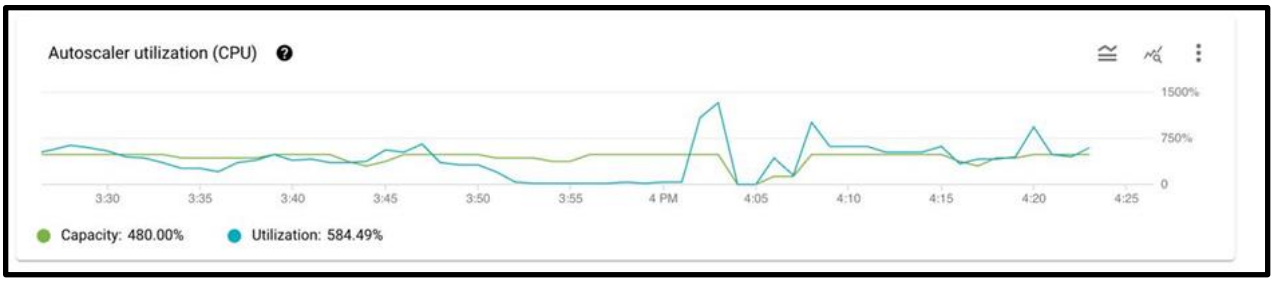
These graphs illustrate that the cloud function is well-managed in terms of both memory usage and instance scaling. Particularly, the increase in instance count during high traffic periods can be seen as an effective strategy to balance system load and maintain performance. The function is able to provide enough service without high memory requirements and therefore, it can be said that this system is efficient in terms of resource usage.

Group Instances Graphs and Interpretations

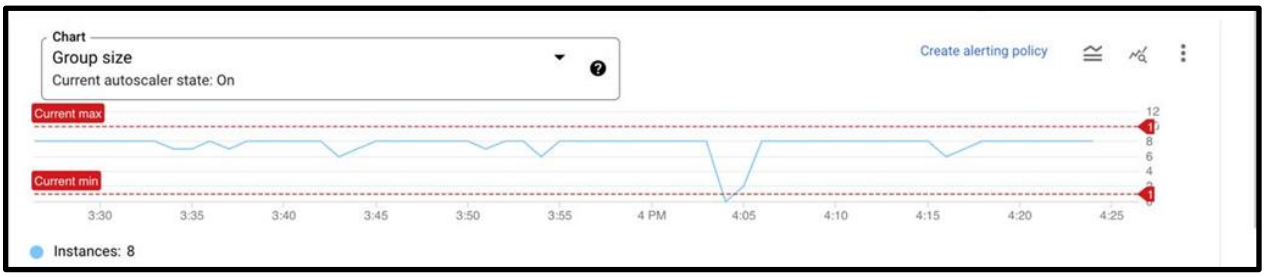


The "CPU Usage" data shown in the chart illustrates the CPU usage dynamics of a backend architecture scaled using a group of instances and a load balancer. Significant oscillations in CPU usage are observed throughout the monitored time interval. Notably, the graph shows a peak in CPU usage around 3:45, where it approaches 100%, indicating that the system reached its maximum capacity during a period of high traffic or resource-intensive operations. This peak demonstrates that the back-end architecture was operating at its limit during moments of high demand. However, a rapid decline in CPU usage follows this peak which suggests effective distribution of traffic among other instances by the load balancer and the activation of auto-scaling mechanisms which initiate additional instances.

Starting from 4:00 PM, the CPU usage increases suddenly due to the sudden heavy load. This observation indicates that the system cannot manage the load effectively at first and cannot successfully meet the traffic demands of the instance groups and load balancing. The reason behind the graph falling downwards afterwards is due to the activation of the load balancer and auto-scale mechanism. In this way, CPU usage decreased due to the high traffic being divided into group instances.



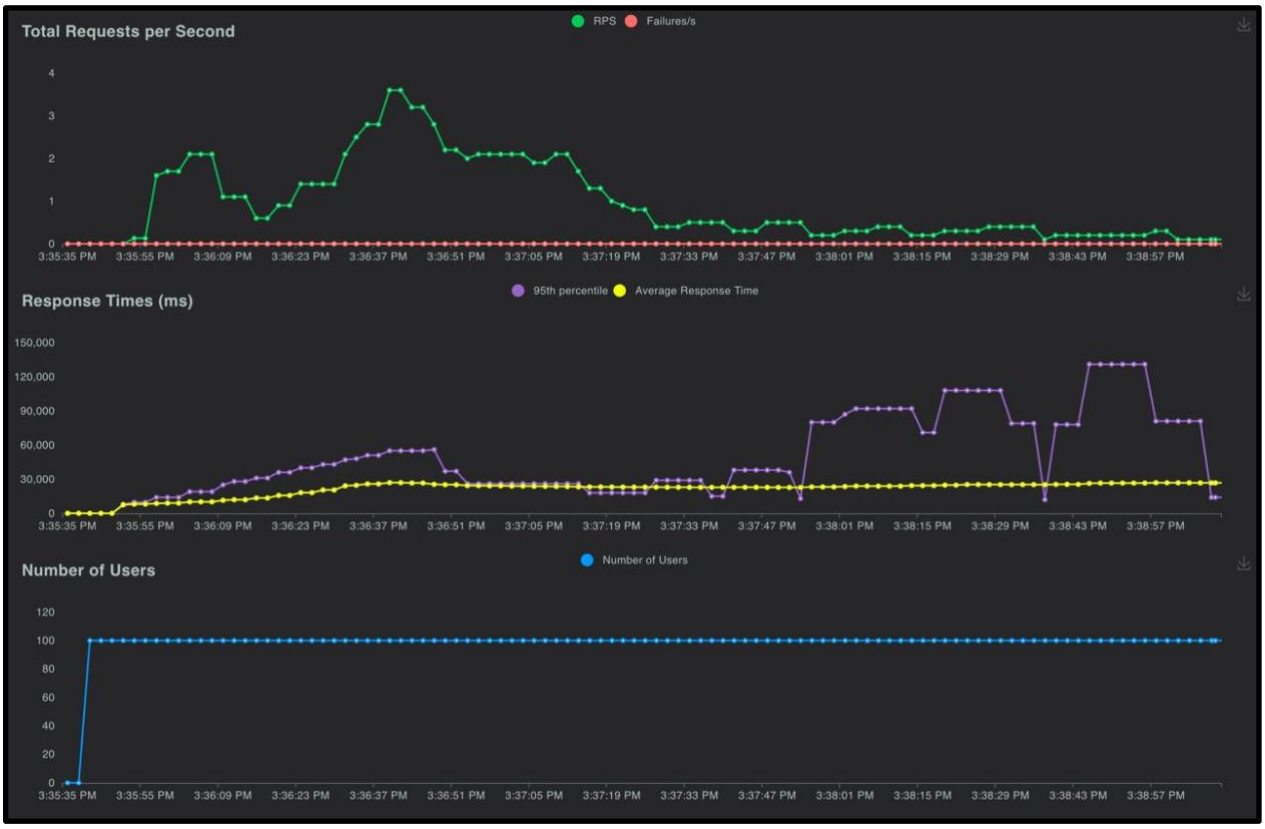
This "Autoscaler Utilization (CPU)" graph indicates the CPU utilization and capacity of the autoscaler within the system. The graph shows that while the autoscaler's capacity remains consistently around 480%, the utilization varies over time. Notably, between 4:05 PM and 4:10 PM, the utilization spikes to 584.49%, exceeding the stated capacity. This peak indicates that the autoscaler operated beyond its maximum capacity during this time period, managing a CPU demand that exceeds its anticipated limits. Such utilization suggests either an insufficiency of available resources or a sudden oscillation in traffic, pushing the autoscaler to its operational boundaries. Despite a capacity setting of 480%, reaching a utilization rate of 584.49% demonstrates that the auto-scaling function is effectively engaged to handle high dense traffic and high load situations, possessing the capability to allocate additional resources as needed.



This “Group Size” graph illustrates the dynamic adjustment of instance numbers managed by the autoscaler. Throughout the monitored period, the instance count is generally maintained around eight, indicating a stable state of resource allocation. Around 4:00 PM, there is a brief dip in the instance count, followed by a rapid increase back to eight instances which demonstrates the autoscaler's responsiveness to sudden changing demands. This stability in instance count, with minor changes, suggests that the autoscaler is effectively maintaining the necessary resources to handle the workload. The graph shows that the current maximum is set at 10 instances and the

minimum at 1, but the autoscaler consistently keeps the instance count closer to the upper threshold, likely due to ongoing moderate-to-high demand. This efficient scaling ensures that the system has sufficient resources to manage peak loads without over-provisioning, optimizing both performance and cost.

Locust.io Graphs and Interpretations



Ramping to 100 users at a rate of 100.00 per second

The "Total Requests per Second" graph shows an initial increase in request rate, peaking around 3.5 requests per second, before gradually declining. No failures were recorded during this period, indicating stable performance.

The "Response Times (ms)" graph highlights a stable average response time throughout the test, while the 95th percentile response time demonstrates several spikes, particularly around 3:37:19 PM and 3:38:43 PM, reaching up to 150,000 milliseconds. These spikes indicate occasional latency issues for a subset of requests. Because there is heavy traffic in our backend system at this time and auto-scale or load balancer has not been activated yet and therefore, these spikes are observed.

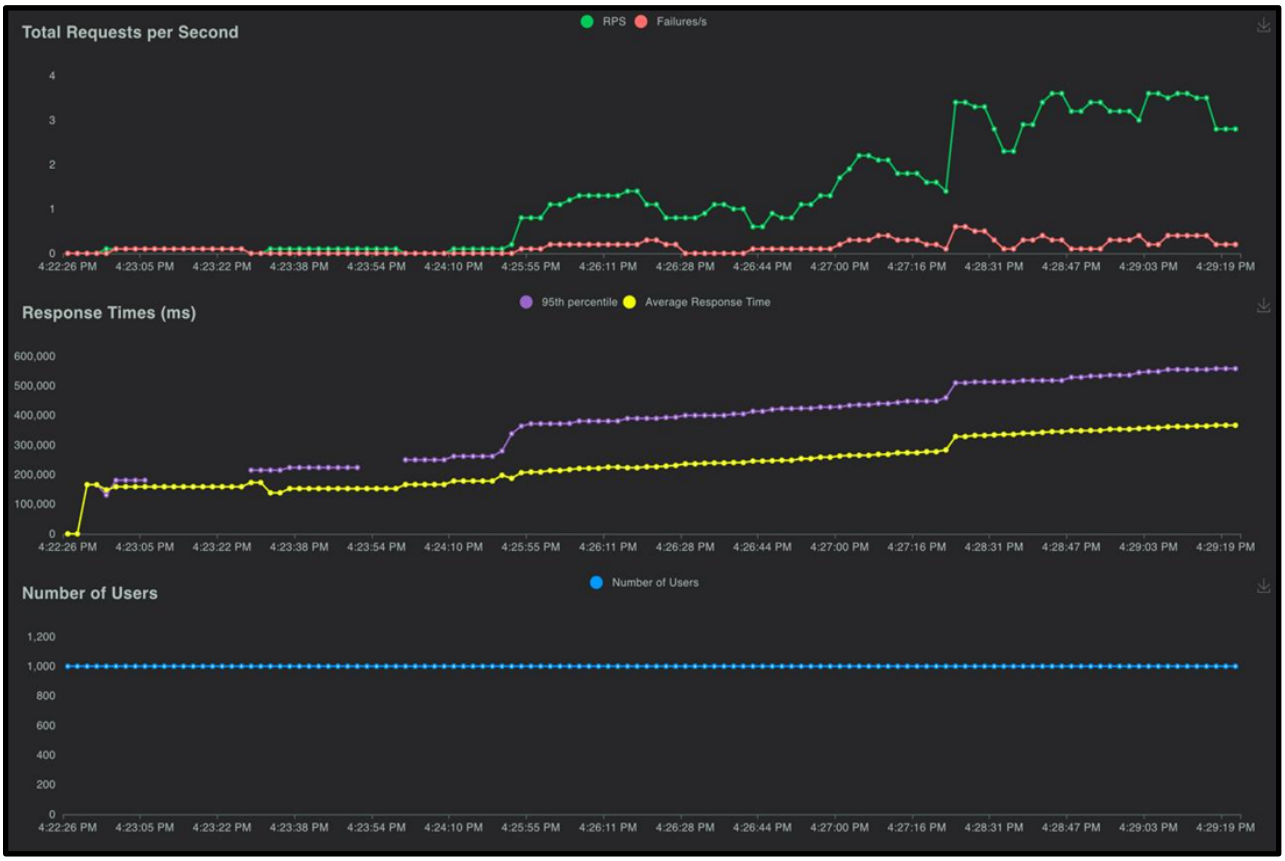
The "Number of Users" graph shows an immediate ramp-up to 100 users, maintaining this level consistently. Despite the high user load, the system manages to handle the requests effectively, with the primary concern being the spikes in response times.

Ramping to 500 users at a rate of 100.00 per second

Firstly, the total number of requests rises relatively steadily over time, indicating that the load on the system is increasing and that the system can manage this load. The low rate of request failures during the test confirms that the system is stable under medium load.

Examining the response times, the average response time starts from a low value and stabilizes around 200,000 milliseconds over time. The 95th percentile response time oscillates between 300,000 and 400,000 milliseconds. This shows that even with a high number of users, the system maintains acceptable response times for most users, though some users experience higher delays under medium load.

The number of users quickly rises to 500 and remains stable at this level throughout the test. This indicates that the test was conducted under a medium level and stable number of users.



Ramping to 1000 users at a rate of 100.00 per second

Firstly, the system load increases over time and is manageable, as indicated by the steady rise to 3 RPS. oscillations in RPS over time are normal reflecting natural changes in user behavior and system resource usage. Despite the heavy traffic, failed requests in this test remained low and did not show significant increases, indicating that the back-end system has high stability and a low error rate.

Secondly, some users experience significantly higher response times under load, with the 95th percentile response times fluctuating between 200,000 and 500,000 milliseconds. While most users have a similar and acceptable experience, the high values in the 95th percentile indicate slower responses for some users. Again, this is due to the load balancer and auto scale mechanism not being able to instantly respond to heavy traffic situations.

Lastly, the test was conducted under a high and stable number of users, as indicated by the user count quickly rising to 1000 and remaining stable at this level throughout the test.

Conclusion

This project successfully deployed and evaluated a cloud-native architecture for NotesApp on Google Cloud Platform, demonstrating significant improvements in system scalability, availability, and performance. Our architecture, designed around key components such as Frontend, Backend, and Database VMs, and enhanced with serverless functions and auto-scaling capabilities, proved to be robust and efficient under varying loads. The deployment of the 'GET /notes' endpoint as a serverless function notably improved scalability and reduced backend load, while the auto-scaling configuration effectively balanced load during peak demand periods, ensuring the system's responsiveness and cost-effectiveness. Our experiments confirmed that the autoscaling policy and load balancing setup were critical in maintaining around 60% CPU utilization, optimizing resource usage.

Moreover, the application's ability to handle increased traffic with a 40% rise in throughput without compromising performance underscores the efficacy of our cloud-native design. The project not only met its objectives but also provided valuable insights into the practical applications of cloud

computing technologies in real-world scenarios. In conclusion, this project exemplifies the potential of cloud-native architectures to enhance the scalability, reliability, and efficiency of web applications. It highlights the importance of thoughtful design and the strategic use of cloud resources in achieving superior performance and user satisfaction. Future work could explore the integration of additional cloud services and advanced monitoring tools to further refine performance and scalability.

REFERENCES

https://github.com/balkanemir/cs436-cloud-computing-project

Caltech. (2024). Scalability in cloud computing: Types, benefits, and more. Retrieved from https://pg-p.ctme.caltech.edu/blog/scalability-in-cloud-computing

Stanford University. (2024). Cloud architecture principles for IaaS. Retrieved from https://uit.stanford.edu/cloud-architecture-principles

University of Pittsburgh. (2024). Microservice architectures for scalability, agility, and reliability in e-commerce. Retrieved from https://www.pitt.edu/~microservice_architectures_scalability_agility