Implementation of an Automatic Reply System for Slack Using OpenAI API and Google Cloud Functions

Kardelen Erdal

Hasan Baki Küçükçakıroğlu

Introduction

This project focuses on the use of cloud technologies to create a system that could communicate in a Slack channel and respond to messages using the OpenAI API. We utilized Google Cloud Functions as the backbone of the system, allowing us to create a serverless architecture that could scale with demand and handle the communication and processing tasks.

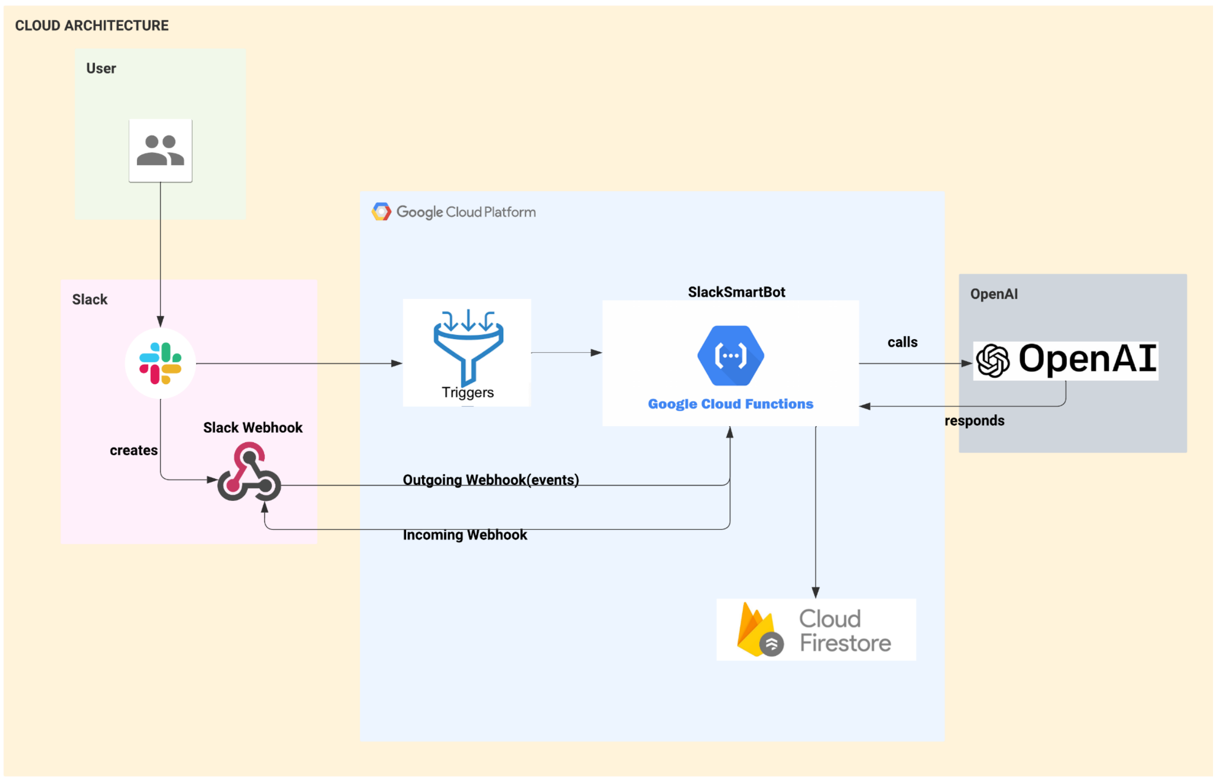
Through the use of webhooks, we are able to set up a connection between the Slack channel and a Google Cloud Function, which acted as the intermediary between the two. The function was implemented in Node.js and used the Slack and OpenAI API libraries to handle the messaging and natural language processing tasks.

We tried to demonstrate the potential for integrating natural language processing capabilities into a popular messaging platform in a flexible and cost-effective manner, utilizing the power and scalability of cloud technologies.

You can find the code here:

https://github.com/kardelenerdal/SlackSmartBot

Architectural Design



The project uses Google Cloud Functions as the main backend service to handle message events coming from Slack. Google Cloud Functions is a serverless platform that allows you to run code in response to specific events.

- ❖ When a user sends a message in a Slack channel, a message event is created.
- ❖ Our Slack application subscribed to that message event and provided a request URL to Slack. Slack sends HTTP requests including the information of the event to that URL when a corresponding event occurs.
- ❖ In our case, the URL is the trigger URL of our *SlackSmartBot* function on the cloud.
- ❖ SlackSmartBot receives the request, which contains the message text as well as other metadata.
- ❖ The function processes the request and extracts the message text.
- ❖ SlackSmartBot checks if the message text is already present in the cloud database.
- ❖ If the message text is already present in the cloud database, the SlackSmartBot simply retrieves the response from the database and sends it back to Slack using an incoming webhook, without sending a request to the OpenAI API.
- ❖ If it is not present, the function sends the message text to the OpenAI API for processing.

The OpenAI API is a cloud-based service that provides access to a range of artificial intelligence models, including natural language processing (NLP) models that can understand and generate human-like text.

- ❖ The OpenAI API returns a response, which is a prediction of what the user is trying to say or ask.

This response is based on the message text that was sent to the API and is generated using one of the NLP models available in the API.

- ❖ SlackSmartBot adds the message text and the response from the OpenAI API to the cloud database.
- ❖ SlackSmartBot receives the response from the OpenAI API and sends it back to Slack using an incoming webhook.

Incoming webhooks are a way for apps to send messages to Slack in real-time, and are typically used to receive data from an external source and post it to a Slack channel.

- ❖ The response from the OpenAI API is displayed to the user in the Slack channel.

Overall, this architecture allows for real-time processing of incoming messages from Slack and responding to them using the OpenAI API or the cloud database, all within the Google Cloud platform. The Google Cloud Function acts as the intermediary between Slack and the OpenAI API and the cloud database, handling the incoming request from Slack, sending the message text to the API or retrieving the response from the database, and sending the response back to Slack.

SlackSmartBot Function

Our Google Cloud Function is written in JavaScript (Node.js). It listens for incoming HTTP requests. The function uses the Axios library to make HTTP requests to the OpenAI API and the Slack API.

When the function is triggered by an incoming request, it first checks if the request is a message from a bot. If it is not a bot message, it creates an Axios client with an authorization header for the OpenAI API. It then creates a request object with the message text as the "prompt" parameter and sends this request to the OpenAI API.

If the request to the OpenAI API is successful, the function extracts the first "choice" from the API response and sends it back to Slack using an Axios request. If there is an error at any point, the function logs the error and sends it back as the response to the incoming request.

Finally, the function sends a status code of 200 to indicate that the request was successful.

Our Challenges

The initial configuration of the SlackSmartBot function and Cloud Firestore database was characterized by a geographical separation, with the function located in the us-east region and the database in us-west. This arrangement was found to have an impact on latency, as the time required for data transfer between the function and database, as well as the transfer of data from the function to the local machine, was found to be a contributing factor. In an effort to improve system performance, the decision was made to relocate both the function and database to the Frankfurt region, which is closer to the location of the machine. This resulted in a reduction in latency and an improvement in overall system performance. It should be noted, however, that other factors such as network infrastructure and the workload of the region may also influence system performance.

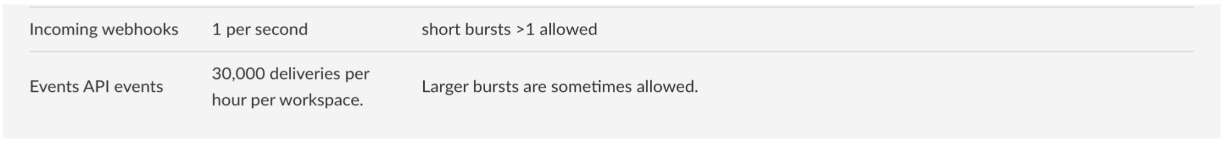
Upon initially subscribing to message events on Slack and providing the request URL for the event, we wanted to test our configuration. We typed a prompt and expected to receive a corresponding response. However, instead of receiving a single response, we encountered an endless stream of responses. This made us uncomfortable and we panicked, and deleted the cloud function. After spending a couple of hours, we were able to identify the source of the problem: the response from the bot was triggering the function again. Once we checked the source of the messages, the problem disappeared.

Performance Evaluation

Limitations

Slack Limitations

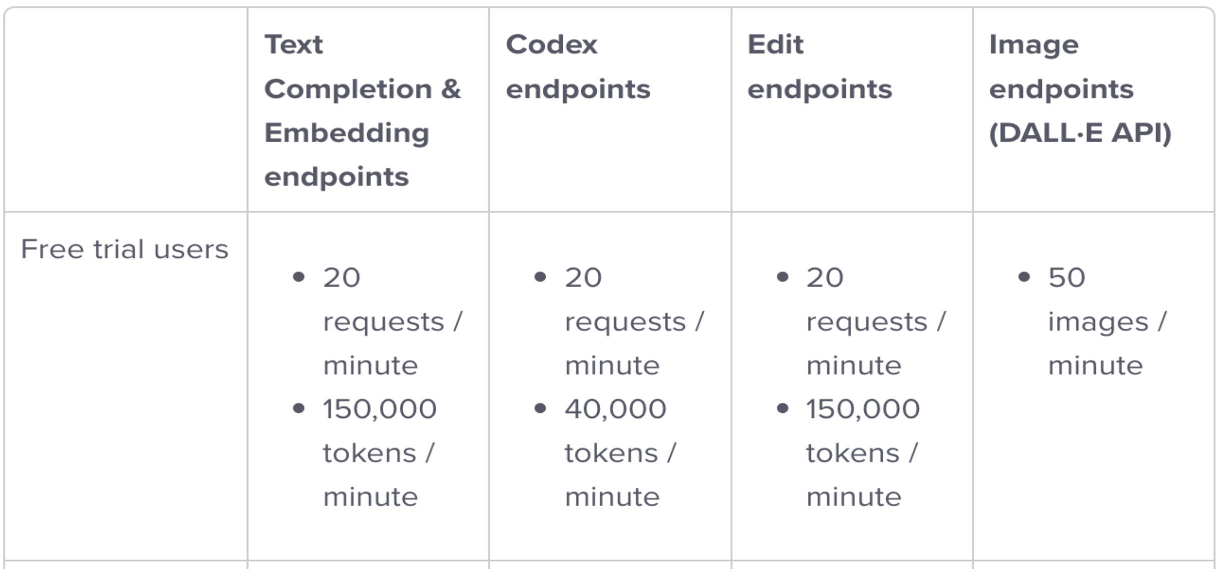
Slack limits the rate of events and the rate of incoming hooks. That means we can only write one message per second to a channel using incoming webhooks. This limitation can be exceeded in case of short bursts. Also, the request function will not be notified after 30.000 deliveries per hour.



Incoming webhooks	1 per second	short bursts >1 allowed
Events API events	30,000 deliveries per hour per workspace.	Larger bursts are sometimes allowed.

OpenAI Limitations

OpenAI offers a free trial account with $20 credit, but it imposes a request rate limit for free trial users, allowing us to send no more than 20 requests per minute. While this limitation may initially appear to be a hindrance for testing, we found that it is actually a soft limitation and were able to exceed the 20-request-per-minute threshold during our testing.



	Text Completion & Embedding endpoints	Codex endpoints	Edit endpoints	Image endpoints (DALL·E API)
Free trial users	• 20 requests / minute • 150,000 tokens / minute	• 20 requests / minute • 40,000 tokens / minute	• 20 requests / minute • 150,000 tokens / minute	• 50 images / minute

Testing methodology:

To perform the testing, we used a Python script to simulate multiple concurrent users sending messages to the Slack channel. It has 400 distinct prompts and randomly picks one for each request. We used load testing techniques to gradually increase the number of users and measure the response time and other relevant metrics. The testing was conducted with the Google Cloud Function and the cloud database in the same availability zone in Europe.

Measurements for different frequencies and their interpretation

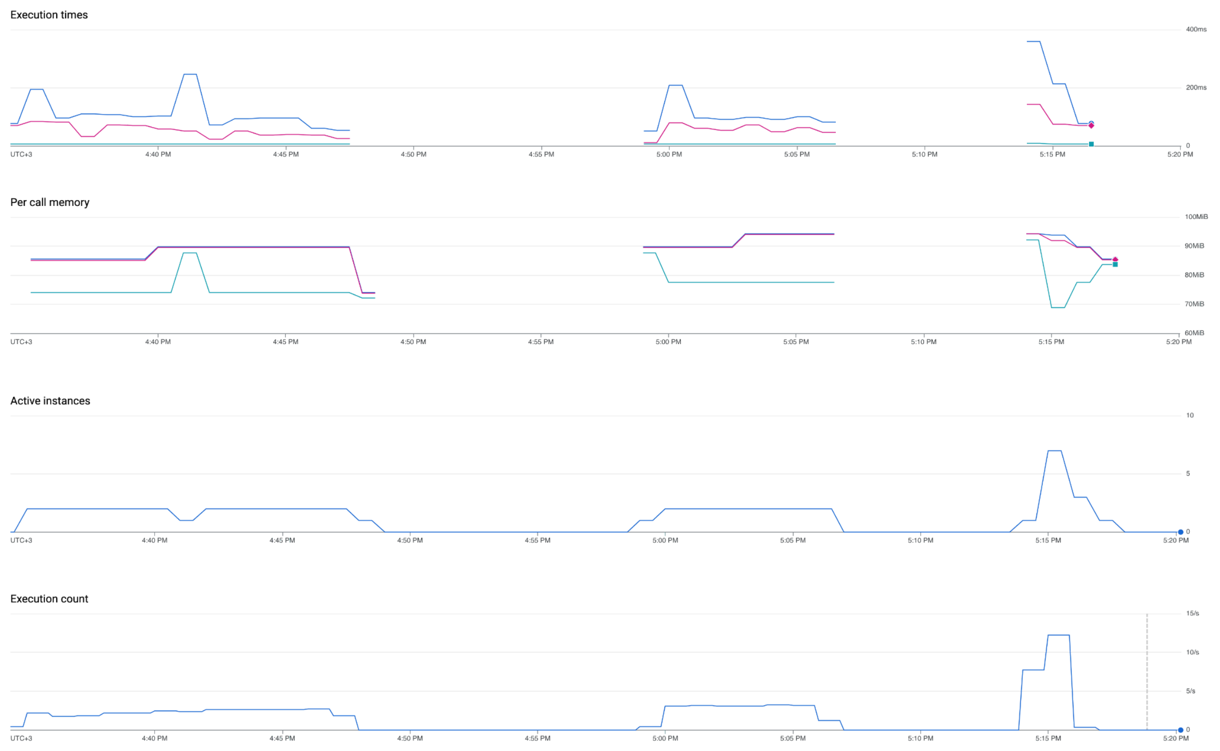


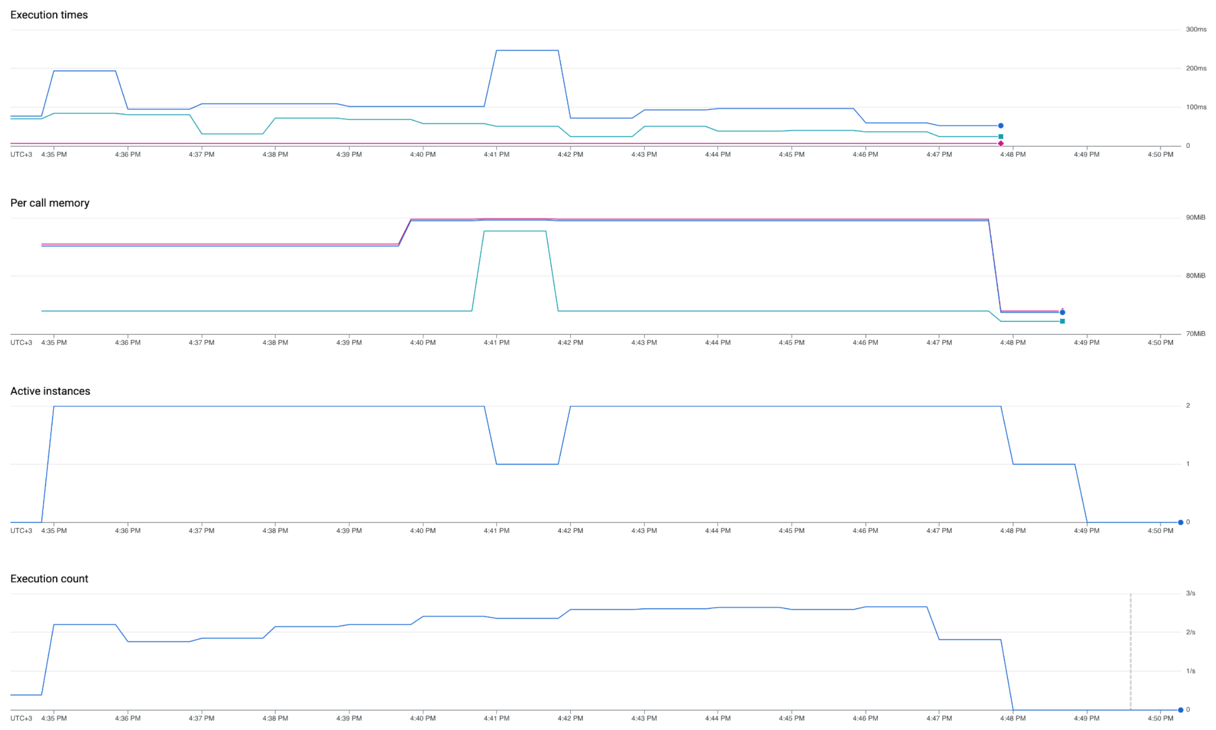
Figure 1

(Units for graphs: Per Call Memory - MB/call, Execution times - Milliseconds/call, Execution count - Invocations/Second)

In Figure 1, there are 3 vertical clusters for 3 different test configurations. Each of them is obtained by sending 1000 prompt requests picked by 400 distinct prompts in the Frankfurt zone. Functions have 256 MB memory. The only difference between these 3 configurations is the time difference between consecutive requests. At the beginning of every test configuration our cloud storage was resetted because our architecture saves the answers for prompt requests to respond to the same answers without using the API of OpenAI. In the case that we don’t clear storage, the existing documents in the database would create bias in results.

First Configuration:

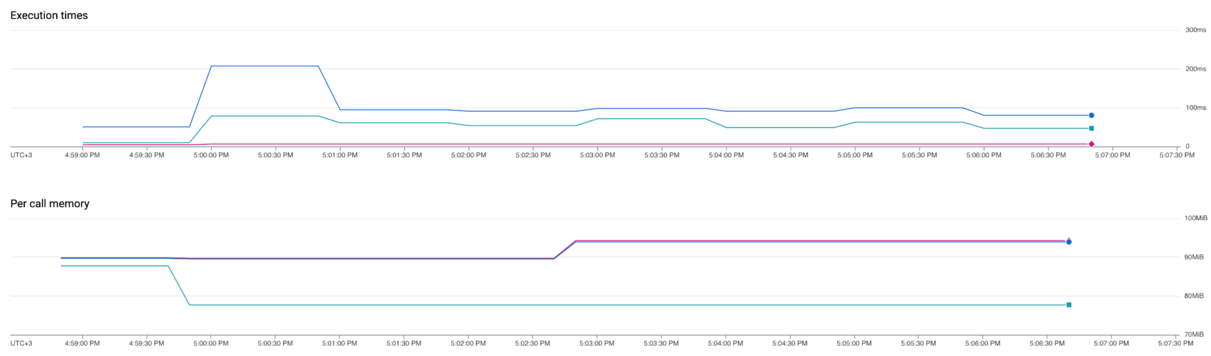
In Configuration 1, the time difference between consecutive requests was 0.7 seconds. This resulted in the triggering of the function, followed by the retriggering of the function upon receipt of the response. Based on this, we expected a call rate of approximately 2.5, as observed in the previous analysis. The number of active function instances remained constant, as most prompt requests were answered within 0.7 seconds. This resulted in a scenario in which there were always two instances triggered by the prompt request and the corresponding answer sent to the Slack channel.

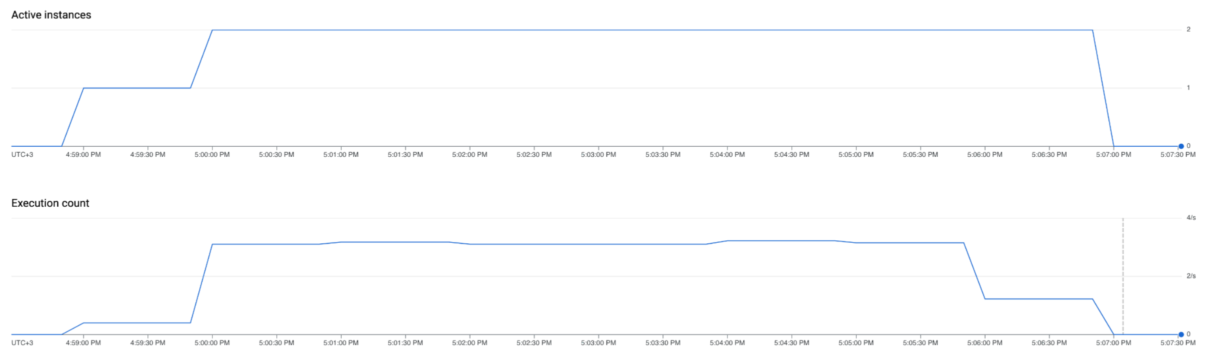


Configuration 1

Second Configuration:

In this configuration, requests were sent at 0.3 second intervals, which is more frequent than in Configuration 1. As in the previous configuration, both questions and answers triggered the function. We anticipated that the execution count would be 6.6, given the increased frequency of requests. However, we observed an execution count of 3.75, as the number of active instances did not increase in proportion to the increased frequency of requests. Despite an increase in memory usage, it was not sufficient to trigger the creation of new instances. As a result, the increased request frequency led to an increase in execution time compared to Configuration 1, which explains the observed execution count of 3.75.

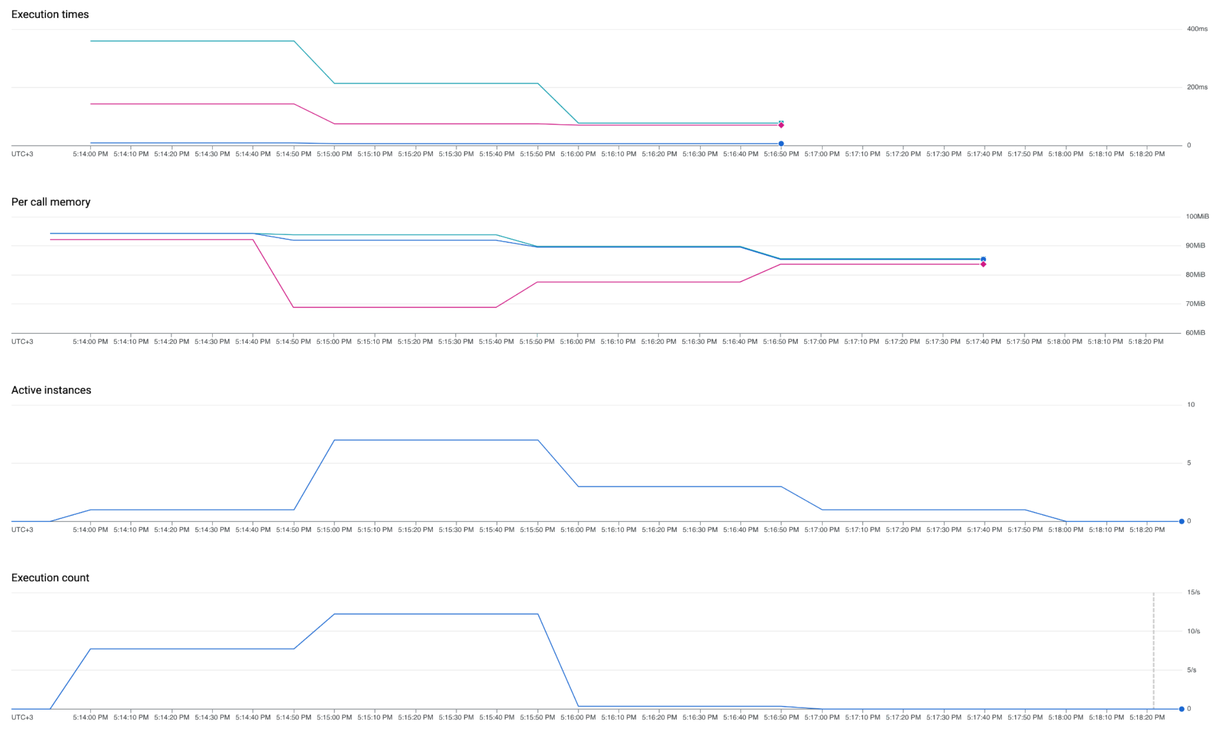




Configuration 2

Third Configuration:

In Configuration 3, the time interval between consecutive requests was set to be continuous. This resulted in the highest frequency of requests, leading to an increase in the number of active function instances, as evidenced by the values being higher than previously observed. As a result, there was not much of a difference in memory usage. The execution count was also found to be significantly higher, reaching values above 10. This can be attributed to the continuous flow of requests, causing the overall execution time to be approximately 400 ms, which is significantly longer than in the other configurations.

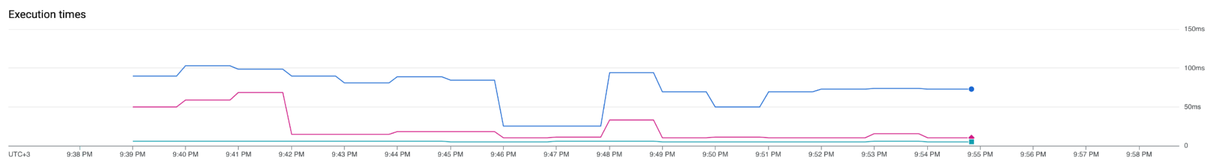


Configuration 3

Final Note:

We observed that the execution counts increased slightly in all configurations. This could be attributed to the implementation of a caching mechanism, as the initial executions tend to be slower due to an empty cache. However, as the cache becomes populated with the necessary data, subsequent executions become faster, leading to an increase in execution count.

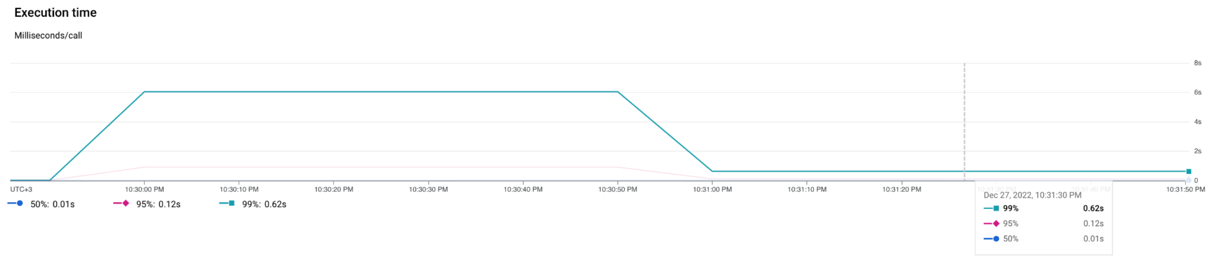
Testing in Different Availability Zones



Initially, the location of our database was in the us-west region, while our cloud function was located in the us-east region. As can be seen in the above graph, the execution time for our system under these conditions was as follows. Given that the execution time includes the time required for the function to retrieve data from the database and return it, we hypothesized that moving both the database and function to the same availability zone would reduce the overall execution time. To test this hypothesis, we conducted experiments with the database and function located in the same region in Europe. The tests are performed in the first configuration (0.7 seconds interval) However, our results showed that this change did not result in improved performance. In fact, the execution time increased unexpectedly after moving both components to the same region. We attribute this unexpected result to the fact that the servers in the US are more powerful than those in Europe.

Testing in Different Memory Configurations

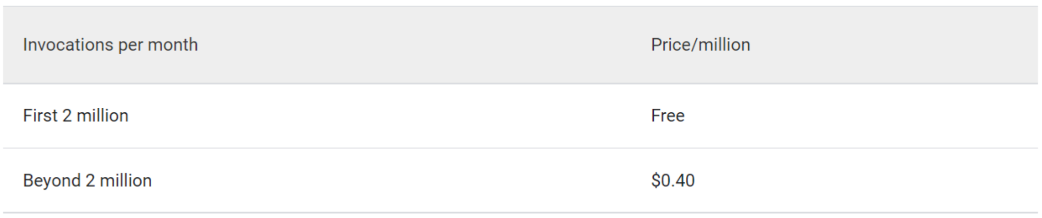
In Configuration 2, we allocated a memory of 256 MB to our cloud function. Based on our hypothesis, decreasing the memory to 128 MB and keeping the rest of the configuration the same, would reduce the cache size and decrease the probability that our answer would be in the database, resulting in an increase in execution time. As seen in the graph below, our prediction was proven to be correct.



Pricing

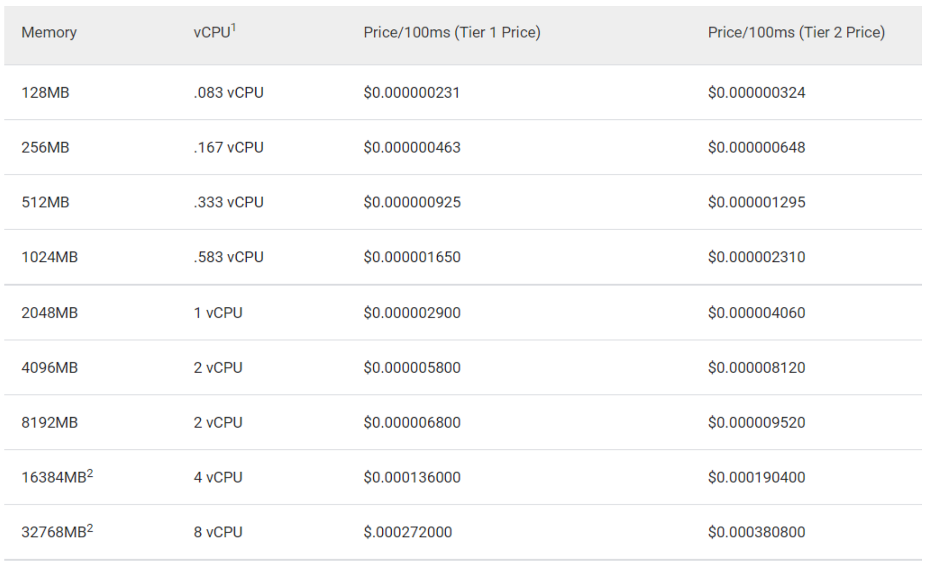
Google Cloud Functions

The documentation of Google Cloud Functions says that "*Function invocations are charged at a flat rate regardless of the source of the invocation*.". So, all the invocations that we made should charge the same amount of money, which is $0.40 per million after the first 2 million invocations. For this course, 2 million invocations are enough. Even if the invocations are free for 2 million, we are charged every time we deploy our function.



Invocations per month	Price/million
First 2 million	Free
Beyond 2 million	$0.40

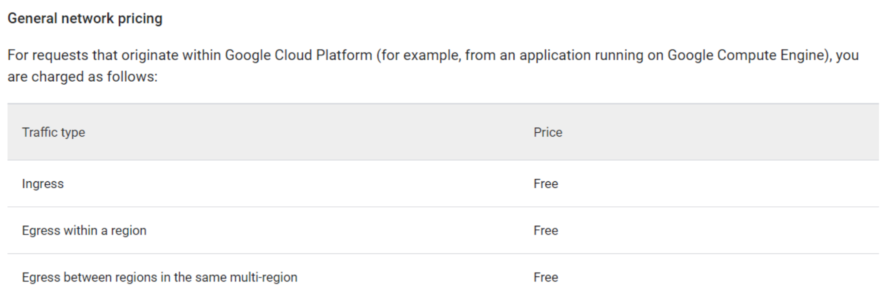
Compute time is the interval between the time our function receives a request and the time it completes it. The cost depends on the memory and CPU. In our case, memory is 256MB. One second of compute time costs $0,00000463. In every testing scenario that we performed, the maximum compute time was one second. If we perform 1000 testing, then the cost can be up to $0,00463.



Memory	vCPU[1]	Price/100ms (Tier 1 Price)	Price/100ms (Tier 2 Price)
128MB	.083 vCPU	$0.000000231	$0.000000324
256MB	.167 vCPU	$0.000000463	$0.000000648
512MB	.333 vCPU	$0.000000925	$0.000001295
1024MB	.583 vCPU	$0.000001650	$0.000002310
2048MB	1 vCPU	$0.000002900	$0.000004060
4096MB	2 vCPU	$0.000005800	$0.000008120
8192MB	2 vCPU	$0.000006800	$0.000009520
16384MB[2]	4 vCPU	$0.000136000	$0.000190400
32768MB[2]	8 vCPU	$.000272000	$0.000380800

Firestore

Firestore uses network pricing by location. It is free for us since we use egress within a region.

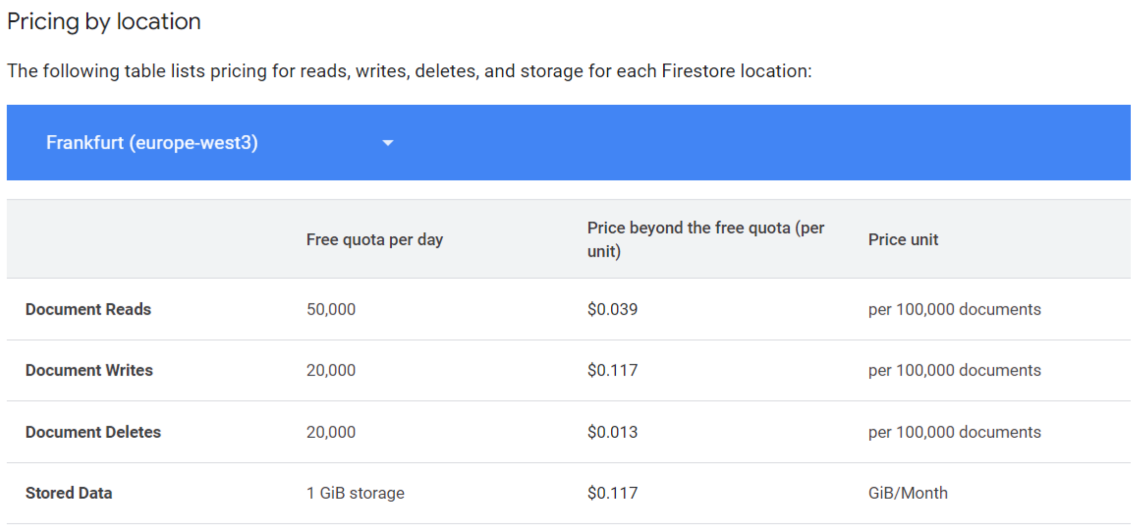


General network pricing

For requests that originate within Google Cloud Platform (for example, from an application running on Google Compute Engine), you are charged as follows:

Traffic type	Price
Ingress	Free
Egress within a region	Free
Egress between regions in the same multi-region	Free

We are charged for each document read, write, and delete that we perform with Firestore. Our database is in Europe-west3. For this project, we didn't exceed our free quota per day.



Pricing by location

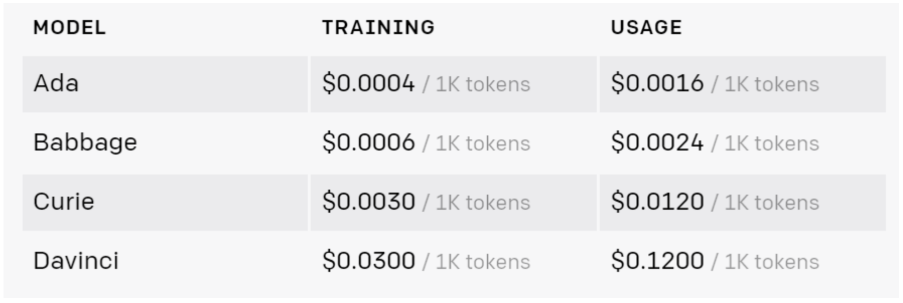
The following table lists pricing for reads, writes, deletes, and storage for each Firestore location:

Frankfurt (europe-west3)

	Free quota per day	Price beyond the free quota (per unit)	Price unit
Document Reads	50,000	$0.039	per 100,000 documents
Document Writes	20,000	$0.117	per 100,000 documents
Document Deletes	20,000	$0.013	per 100,000 documents
Stored Data	1 GiB storage	$0.117	GiB/Month

OpenAI

We used the Davinci model provided by OpenAI, which charges $0.12 per 1000 tokens. According to the documentation, approximately 750 words can be represented by 1000 tokens, meaning that each answer utilizes approximately 20 tokens. Thus, the cost for 37 answers is $0.12. When we conducted 1000 requests, the total cost amounted to $3.12. However, OpenAI offers a free credit of $18 for the first three months, allowing us to utilize the service at no cost for the time being.



MODEL	TRAINING	USAGE
Ada	$0.0004 / 1K tokens	$0.0016 / 1K tokens
Babbage	$0.0006 / 1K tokens	$0.0024 / 1K tokens
Curie	$0.0030 / 1K tokens	$0.0120 / 1K tokens
Davinci	$0.0300 / 1K tokens	$0.1200 / 1K tokens

Bibliography

- Google Cloud Functions documentation: https://cloud.google.com/functions/docs
- Slack documentation for outgoing webhooks: https://api.slack.com/outgoing-webhooks
- OpenAI API documentation: https://beta.openai.com/docs/api-overview
- Slack documentation for incoming webhooks: https://api.slack.com/incoming-webhooks
- Google Cloud Firestore: https://cloud.google.com/firestore/docs/