Topic Categories of Eksisozluk’s gundem

CMPE48A Cloud Computing Project

Sabri Mete Akyuz

1. Introduction

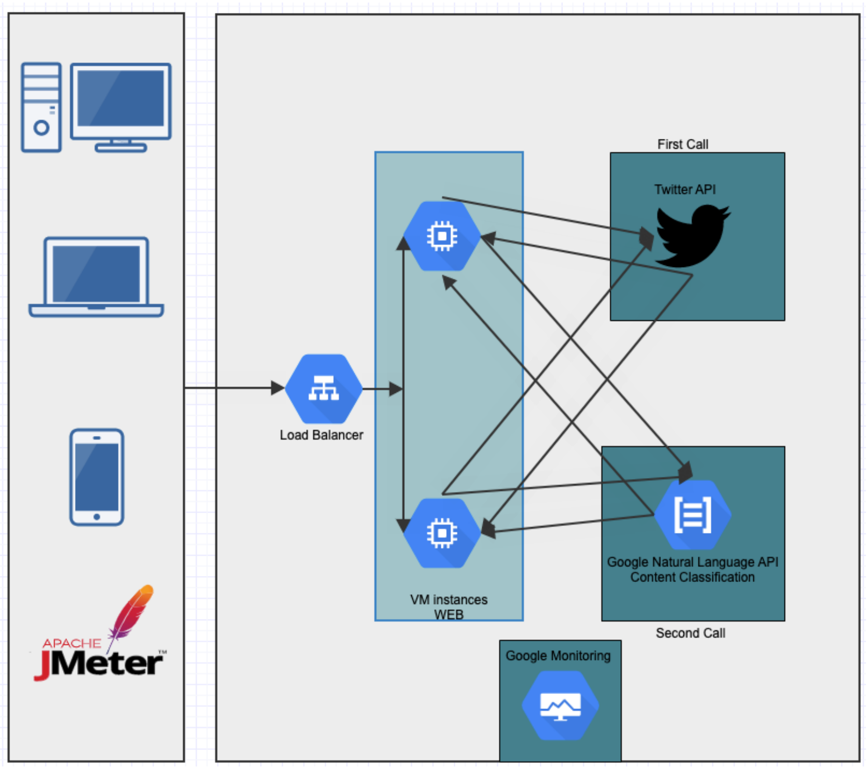
Eksi Sozluk is a collaborative hypertext dictionary based on the concept of Web sites built up on user contribution. It has a lot of Turkish users. One can track what is happening in Turkey from its ‘gundem’ pages.

This project aims to find the topic categories of Eksisozluk’s gundem by using Google Natural Language API to classify contents. To prepare the data, web scraping from eksisozluk and translating topics into English are required.

Codes can be found on: https://github.com/sabrimete/eksi-gundem-categorizer

2. Architecture Design

a. Initial Architecture

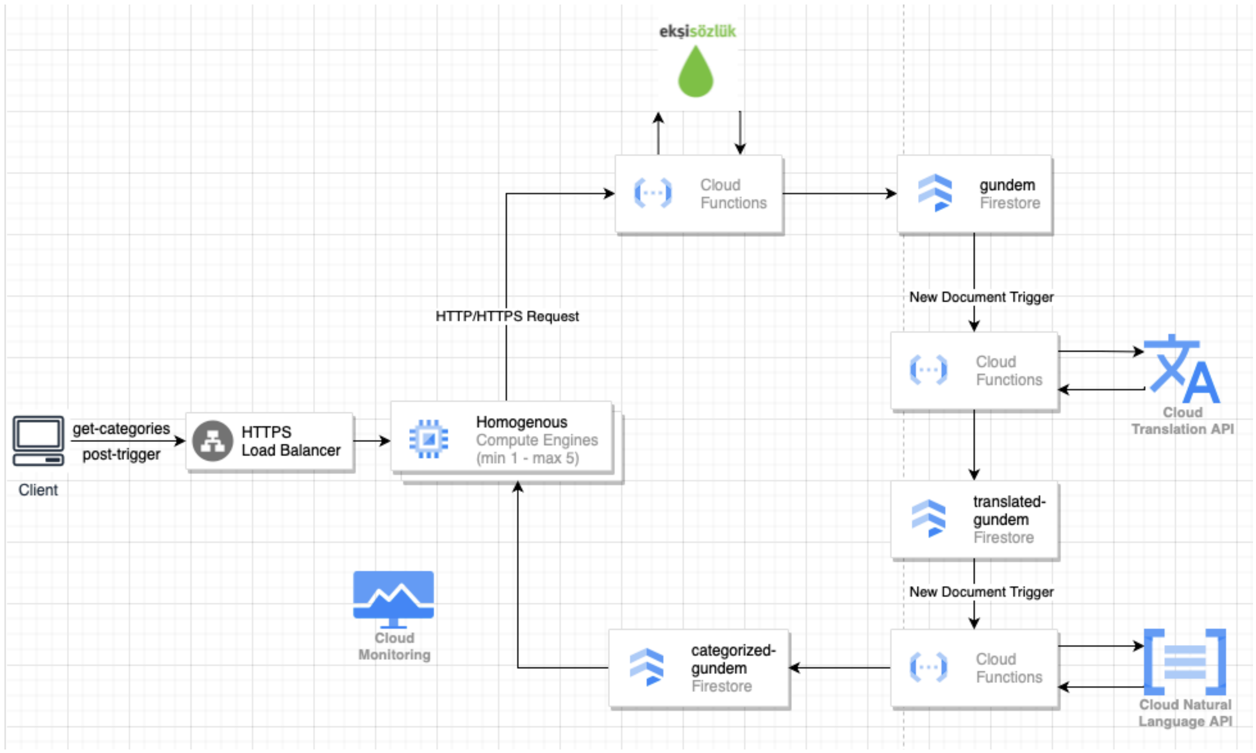


Initial architecture consists of one Load Balancer, two VM instances that make API calls to Twitter API and then sends the results into Google Natural Language API for content classification.
To evaluate the performance of the system, I planned to use JMeter at that time.
For monitoring Google's monitoring dashboard was planned to use.

Reasons To Change This Architecture:

i. In the initial plan, Twitter was used instead of Eksisozluk. Change was needed because Twitter didn't give a developer access to its API.
ii. It's a quite simple architecture. Almost of all of the works are handled by the two VMs. Adding more cloud components would be more meaningful for this project, since its main goal is to make us have experience about cloud programming.

b. Final Architecture



Components:

i. Load Balancer: Google's HTTP Load Balancer is used to distribute the load between VM's. It gets information about the VMs by using predefined Health Check method. By using that information, it redirect requests to separate VMs.

ii. Homogenous VMs: Google's compute engines are used. They are initialized under one Instance Group. It has auto scaling ability to scale the system between 1 and 5 VMs according to the CPU utilizations. The Instance Group uses a Instance Template to create required VMs while scaling. Startup script was also provided while configuring the Instance Template to create VMs that can start responding requests.
   They accept two types of requests. One is for triggering an event chain to find the topic categories. The other one is for getting the latest categorized topics from 'categorized-gundem' collection in Firestore.

iii. Google Cloud Functions:
   1. Function that is triggered by an HTTP request. It fetches the 'gundem' from eksisozluk and create a new document with topics and some of its entries under the 'gundem' collection in Firestore.
   2. Function that is triggered by a creation of a document under the 'gundem' collection. It takes the document and sends a request to Google Cloud Translation API to translate it into English. Then saves the result under 'translated-gundem' collection in Firestore.
   3. Function that is triggered by a creation of a document under the 'translated-gundem' collection. It takes the document and sends a request to Google Natural Language API to classify the topics. Then saves the result under 'categorized-gundem' collection in Firestore.

iv. Firestore: It is used to store intermediate and final results of the process. It has 3 different collection. 'gundem', 'translated-gundem' and 'categorized-gundem'

v. Cloud Monitoring: It is for monitoring mainly the CPU utilizations of VMs, inspecting the logs and statuses of Cloud Functions.

3. Performance Evaluation

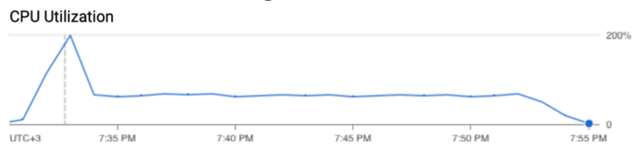
To create load on the system, Python script is used. For multithreaded request sending, aiohttp and asyncio libraries are used.
To monitor the results, usually Google Monitoring Dashboard is used but also Python script is used for time measuring.

There are two parts of the system to measure:

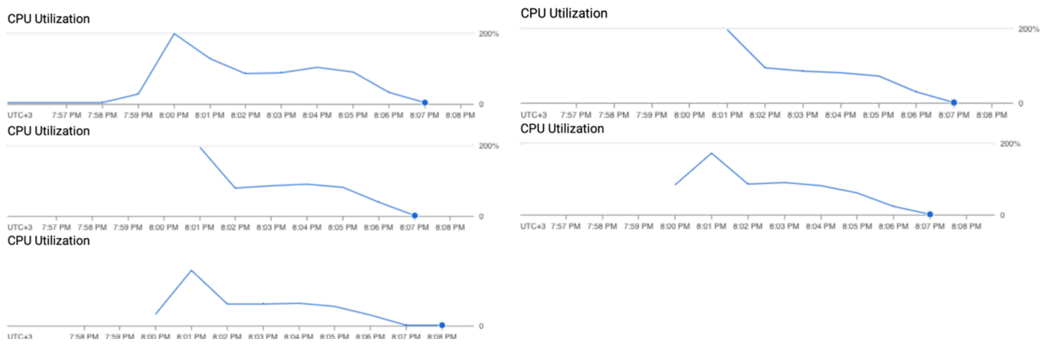
a. VMs

VMs are being created and deleted automatically according to their CPU utilizations in the auto-scalable instance group.
Test scenario is sending 50k requests (sent 5k by 5k) to only one VM instance and also the auto-scalable VM instances. Measure the time spent for responding all of the requests and observe the CPU utilizations to understand the benefits of using auto-scalable VM instances with a load balancer.

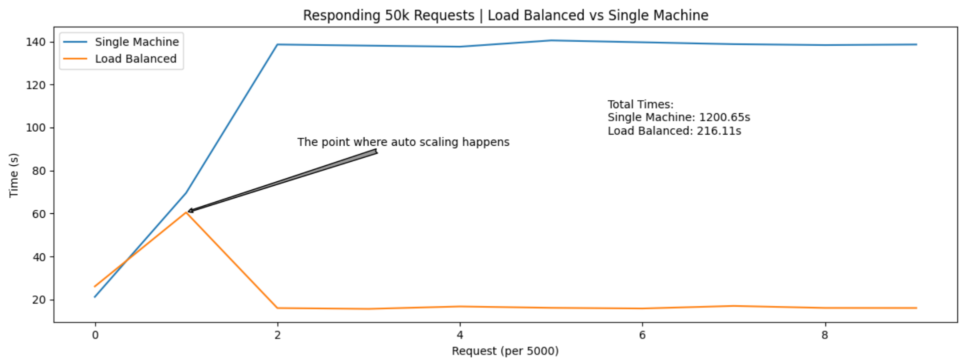
- CPU Utilization – Single VM



- CPU Utilizations – Load Balanced VMs



- Time Spent for Responding All Requests



As it can be seen from the graphs, auto-scalable instances performed much better compared to a single VM. Single VM almost took 6 times more time.
Single VM responded in 1200 seconds, auto-scalable VM instances responded in 216 seconds.
From the CPU utilization graphs, we can see that VM's created at different times, this is because the auto scalability requires time to happen. At first, it needed to be sure that VMs are required, after that it creates VMs.
While VMs are initializing, their CPU utilizations went really high, probably because of the startup script. It installs libraries and creates files to have the system up and running.
After second batch of requests, all of the 5 VMs were created, that's why time required to respond the batch of requests decreased and stabilized. In the meantime, single VM loaded by a lot of requests and tried its best to perform. Until this requests are cumulating more and more.

b. Functions

There are three functions that are trigged by successively in the system.
First one takes approximately 90 seconds
Second one takes approximately 0.4 seconds
Third one takes approximately 0.3 seconds

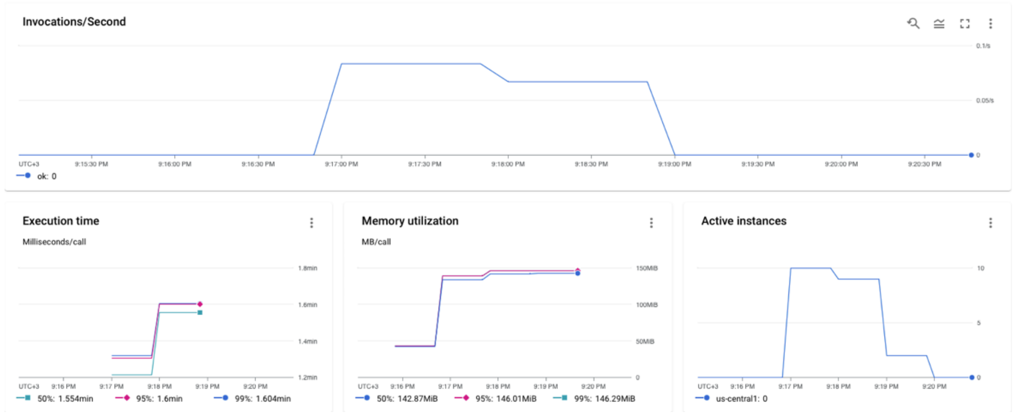
The bottleneck is unarguably the first one. That’s why only the first one measured with different parameters. The others can not affect the system that much, so it is not very meaningful to test them.

There are two meaningful parameter to change for the functions to test them: Memory and Max Number of Instances.

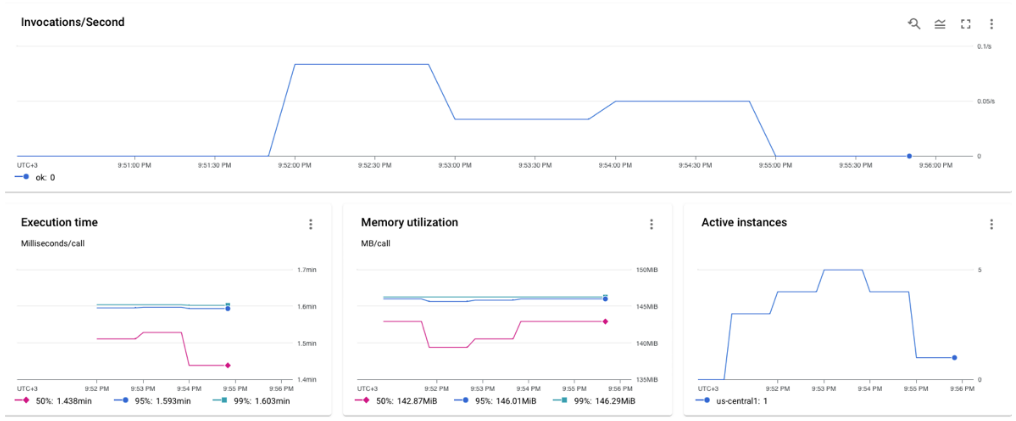
Default value for the Memory is 256MB. I tried to reduce it to 128MB but the function couldn’t work properly. So, I kept it on 256MB.

My test scenario was sending 10 requests to trigger functions while changing maximum number of instances. Values tried are 10 (means no limit for the test case), 5 and 2.

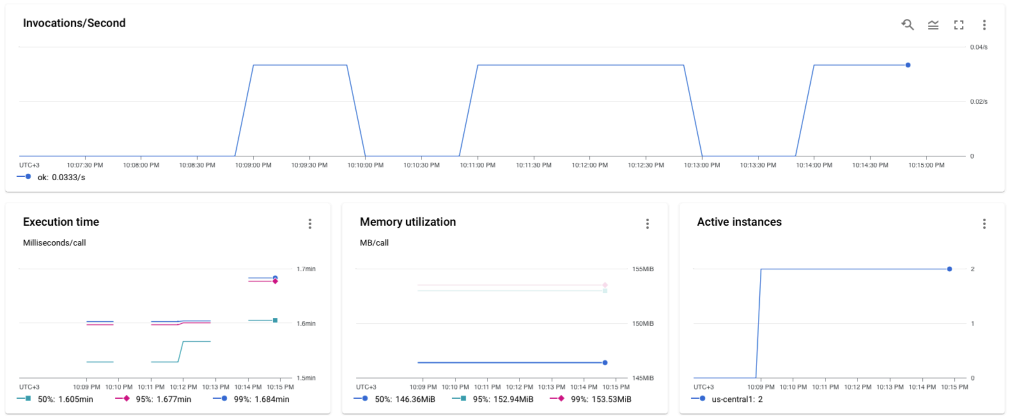
- Max 10 instances



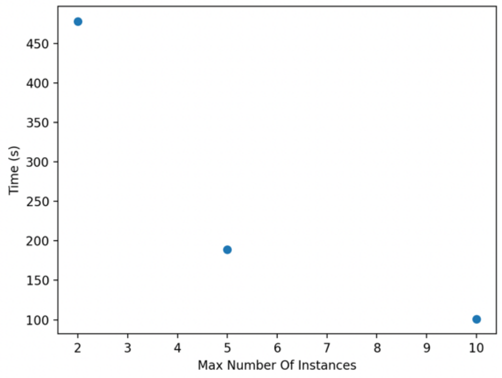
- Max 5 instances



- Max 2 instances



- Time Spent to execute the function 10 times



As it can be seen from graphs, execution times (for each function) and memory utilizations are almost the same for each case.

Active instances are, as expected, most of the time at the maximum limit. In max 2 case, it stays at 2 until all of them finishes. However, in other case the number of instances first increases up to limit and then decreases while functions are completed one after each other.

Time required to complete all of the requests took 478 second with max 2 instances, 189 seconds with max 5 instances and 101 seconds with max 10 instances

4. Conclusion

After seeing the performance evaluation results, here are the conclusions made:

- Auto-scalable VM instances perform better than a single VM. Test scenario didn't compare the auto-scalable VM instances with fixed size multiple VM instances. However we can say that, it would probably increase the cost. Since auto-scalable instances can reduce the VM numbers as well, if system doesn't have a high load, it saves cost.
- Keeping max number of instance parameter as high as possible gives better response times. If we consider their cost, since we pay for the time lambda function runs, it doesn't increase the cost while we are increasing the max number of instances. This statement is true if cold-start time for Google function is not that much. If it requires relatively longer compared to the actual task it should make, then creating a lot of function may increase the cost.

To make the system better, maybe combining some of the Google Functions would be useful. As I mentioned above, when we separate the functions, it will increase cold-start time eventually and this cause higher costs.

To increase usability of the system, we can add new functionalities for the get method. For example, right now it only returns the latest result from the Firestore but it can be modified to return the results from given date.