Instituto Superior Técnico

Cloud Computing and Virtualization

RadarScanner@Cloud

(Final Submission)

Group 17

Sara Machado

86923

Rafael Figueiredo

90770

Ricardo Grade

90774

1. Introduction

The objective of the CCV project is to design and implement a simulated radar scan service hosted on an elastic cluster of WebServers.

The system receives requests from clients, where each request performs a scan, using different algorithms, on a given map based on the starting point and the maximum intended range for the scan.

For the system to be scalable and efficient, it uses AWS instances that are created and terminated according to the system workload and contains a LoadBalancer that routes incoming requests to the instance with the lowest workload.

2. Architecture

The project consists of 5 modules:

- The **Solver**, which simulates the radar scan service.
- The **Load Balancer**, which receives requests from the client and distributes them among all available instances running the WebServer, so that the instances maintain an equal level of workload.
- The **Auto-Scaler**, that decides when instances need to be created or terminated.
- The **WebServer**, which executes the request and saves its cost in the MSS.
- The **MSS**, which keeps track of the costs associated with requests.

3. Implementation

3.1. Instrumentalization

Our first implementation instrumented Solver with several metrics, such as:

- Number of Instructions.
- Number of Basic Blocks.
- Number of Routine Calls.
- Number of Stores.
- Number of Loads.
- Number of News.
- Stack Depth.

However, the instrumentation code imposed a significant overhead on our system, which is why studies have been carried out to select the most appropriate metrics that best describe the cost that the request imposes on the WebServers instances.

The number of instructions produces a great deal of overhead in executing a request, as a function call is inserted before each instruction.

The number of instructions and the number of basic blocks give the same overhead because calculating the number of basic blocks is done by making a call before each basic block to count the number of instructions that are executed on them.

As these two metrics produce enormous overhead, we tried to find the best alternative for these metrics. For that, we evaluated each metric for each of the strategies and, then, we made a correlation between these values and the number of instructions. The results are shown in Annex 1.

Based on the results, we choose the metric with the closest correlation to 1.0 and that imposes the least overhead on the system, that is, it generates less function calls:

- Number of Routine Calls.

3.2. WebServer

Each WebServer receives requests forwarded by the LoadBalancer.

Each request received is then executed by the solver, which is instrumented and, therefore, will collect the requested metric.
Requests are made by different threads.

The WebServer can collect a metric of a given request at the end of processing, because until the metric is fetched, it is kept on a static map in the instrumentation class, where the key is the thread id where the request is being executed.

The collected metric, which is the number of routine calls, is used as a cost.

This cost will be uploaded to the MSS, to allow LoadBalancer to estimate costs of future requests with better precision.

3.3. Load Balancer & Auto Scaler

The LoadBalancer and the AutoScaler are running on the same physical machine, because given the context of the project it is not necessary to run them on different machines, which facilitates the exchange of information.

Therefore, both entities are executed on different threads sharing a map of the available instances and their state, this way, the AutoScaler can act when an instance is marked as unhealthy by the LoadBalancer, and the LoadBalancer can take into account when a new instance is launched, or terminated by the AutoScaler.

3.4. Load Balancer

When LoadBalancer is started, it creates the RequestsCosts table in DynamoDB that is used by the WebServer to upload costs associated with each request and, by itself, to fetch costs for requests made recently, every 30 seconds.

In addition, every 30 seconds, the LoadBalancer executes HealthChecks for the WebServers, which returns 200 OK if it was successful.

When a WebServer fails to respond to a HealthCheck, the LoadBalancer increases the number of fails associated with that WebServer and stops considering the WebServer as suitable for forwarding new requests.

If the WebServer accumulates 2 fails, the instance will be Unhealthy and will be replaced by the AutoScaler.

If the WebServer accumulates 4 successful HealthChecks in a row, the number of accumulated fails will be

reset and the WebServer will return to healthy and therefore the LoadBalancer can forward requests to it again.

When the LoadBalancer receives a request, it first estimates its associated cost (Explained in detail in the next subsection). Then it forwards the request to the instance that has the lowest associated estimated cost, which is the sum of the estimated costs of the requests that were forwarded to that instance and have not yet been completed. Finally, it updates the current estimated cost of that instance.

3.4.1. Request Cost Estimation

The estimated cost of a request is the cost of the most similar known request. If the cache contains the same exact request, then this is the most similar, otherwise, it iterates over the cache and for each request, it calculates the similarity percentage that is calculated based on the following weighted percentage:

- **Strategy**: 40%, if it is the same.
- **Scan Range Size**: 30%, if it is the same or a percentage of it based on size similarity.
- **Image Name**: 20%, if it is the same.
- **Starting Point Proximity**: 10%, if it is the same or a percentage of it based on closeness.

The strategy is the parameter that most influences the necessary amount of computation that the request requires, the second most important is the size of the scan range, because the larger it is, the more exhaustive the calculation is, the third most important is the name of the image, the smaller the image mosaics are, the more efficient the calculation is, finally, the last most important parameter taken into account is the proximity to the starting point, which can indicate which mosaics are part of the scan range.

3.4.2. Fault Masking

When a request that has been forwarded to an instance expires or returns with an error, the LoadBalancer registers it as an unsuccessful health check (Marking the instance as not completely healthy), forwarding the request in the same way that it would if it arrived from the client.

3.5. Auto Scaler

AutoScaler is used to detect when the instances running the WebServer are overloaded and, therefore, start new instances, or when it is underloaded, where it shuts down the instances with the least amount of workload.

When the AutoScaler starts, it starts a timer that will automatically scale every 10 seconds to see if an action needs to be taken and replace any instance marked as unhealthy by the LoadBalancer.

After an action is taken, there is a 30-second scale cool down before another action can be taken.

An action can be taken by analyzing two different types of metrics:

- Current Estimated Cost of Instances.
- CPU Utilization of Instances.

Every 10 seconds, the estimated costs are checked.
Every 30 seconds, CPU usage is checked.

3.5.1. Instance Estimated Cost

The system has the sum of the estimated costs of all instances, as the LoadBalancer maintains the amount of workload that each instance has.

When the sum of the estimated costs of all instances $(\sum_{i=o}^{N} Cost(Instance[i]))$ is:

- $> 1.10E + 06 \times N$: It launches a new instance.
- $< 3.00E + 06 \times N$: It removes the instance with the lowest estimated cost associated with it.

Where N is the number of instances running.

3.5.2. Instance CPU Utilization

The AutoScaler asks CloudWatch for datapoints for the average CPU Utilization of each running instance, this request gets all data points from the last 10 minutes. This is because AWS sometimes delays when updating datapoints, although detailed monitoring is enabled, and supposedly would have data points every minute.

We then obtain the most recent datapoint from the list of data points.
When the sum of the average CPU Utilization of all instances $(\sum_{i=o}^{N} CPUUtilization(Instance[i]))$ is:

- $> 80 \times N$: It launches a new instance.
- $< 30 \times N$: It removes the instance with the lowest CPU Utilization associated with it.

Where N is the number of instances running.

To find the thresholds used to perform an action in AutoScaler (either by cost or CPU Utilization), a study was done to understand the relationship between cost and CPU Utilization. The results are presented in Annex 2.
The chosen thresholds correspond to the low workload and high workload limits for a single instance.

Initially, our idea was to use CPU Utilization and the current sum of costs together to decide if an action needed to be taken, but the data points that Cloudwatch sends, most of the time, were out of date or not present, and not accurately represented the system workload, so to increase decision-making efficiency, we decided to act even when only the sum of costs or CPU Utilization required it.

We wanted at least 2 instances to be always running, so initially we launch 2 instances, and one instance can only be removed if more than 2 instances are running. This is done to avoid long delays when a request is made after a period without requests.

3.6. Metrics Storage System

The metrics that the WebServer collects must be maintained so that LoadBalancer can assess the cost of new requests and must persist to preserve all knowledge of previous requests, even in the event of crashes. For this purpose, the request-cost pairs are stored in a table in DynamoDB containing 2 entries:

- **RequestQuery**: Query of the Request.
- **RequestCost**: Cost of the Request.

4. Results

To show the results, we conducted a study where we monitor the instances running on the system and how the workload is being assigned to each instance. The results are presented in Annex 3.

The graphic (**Annex 3.A**) shows how the instances are launched and terminated, based on the amount of workload in the system.

The table (**Annex 3.B**) shows how requests made are being distributed across instances.

Annex 1: Metrics measured by the strategy, and the correlation of each with the number of instructions

Grid									
Dataset	512x512			1024x1024			2048x2048		
Scan Range	256x256	384x384	512x512	512x512	768x768	1022x1022	1024x1024	1536x1536	2046x2046
Routine Calls	1.04E+07	1.09E+07	1.86E+07	2.93E+07	6.03E+07	1.02E+08	4.85E+08	1.14E+09	2.08E+09
Stack Depth	5.00E+00	5.00E+00	5.00E+00	5.00E+00	5.00E+00	5.00E+00	5.00E+00	5.00E+00	5.00E+00
Basic Blocks	1.38E+07	2.92E+07	4.99E+07	7.49E+07	1.55E+08	2.64E+08	1.16E+09	2.73E+09	4.95E+09
Stores	3.06E+07	6.78E+07	1.19E+08	1.25E+08	2.77E+08	4.87E+08	6.63E+08	1.47E+09	2.60E+09
Loads	8.11E+07	1.74E+08	3.00E+08	3.63E+08	7.70E+08	1.33E+09	3.31E+09	7.43E+09	1.32E+10
News	5.08E+03	5.08E+03	5.06E+03	5.06E+03	5.06E+03	5.06E+03	5.06E+03	5.06E+03	5.06E+03
Instructions	3.32E+07	7.07E+07	1.21E+08	1.75E+08	3.66E+08	6.23E+08	2.54E+09	5.98E+09	1.08E+10

Progressive									
Dataset	512x512			1024x1024			2048x2048		
Scan Range	256x256	384x384	512x512	512x512	768x768	1022x1022	1024x1024	1536x1536	2046x2046
Routine Calls	1.04E+05	5.19E+04	5.19E+04	1.23E+05	1.23E+05	1.23E+05	1.63E+06	1.63E+06	1.63E+06
Stack Depth	5.00E+00	5.00E+00	5.00E+00	5.00E+00	5.00E+00	5.00E+00	5.00E+00	5.00E+00	5.00E+00
Basic Blocks	2.00E+05	2.00E+05	2.00E+05	5.21E+05	5.21E+05	5.21E+05	7.35E+06	7.35E+06	7.35E+06
Stores	1.03E+06	1.03E+06	1.03E+06	3.07E+06	3.07E+06	3.07E+06	4.61E+07	4.61E+07	4.61E+07
Loads	4.83E+06	4.83E+06	4.83E+06	1.65E+07	1.65E+07	1.65E+07	1.43E+08	1.43E+08	1.43E+08
News	9.70E+03	9.70E+03	9.70E+03	1.91E+04	1.91E+04	1.91E+04	2.19E+05	2.19E+05	2.19E+05
Instructions	5.90E+05	5.90E+05	5.90E+05	1.60E+06	1.60E+06	1.60E+06	2.31E+07	2.31E+07	2.31E+07

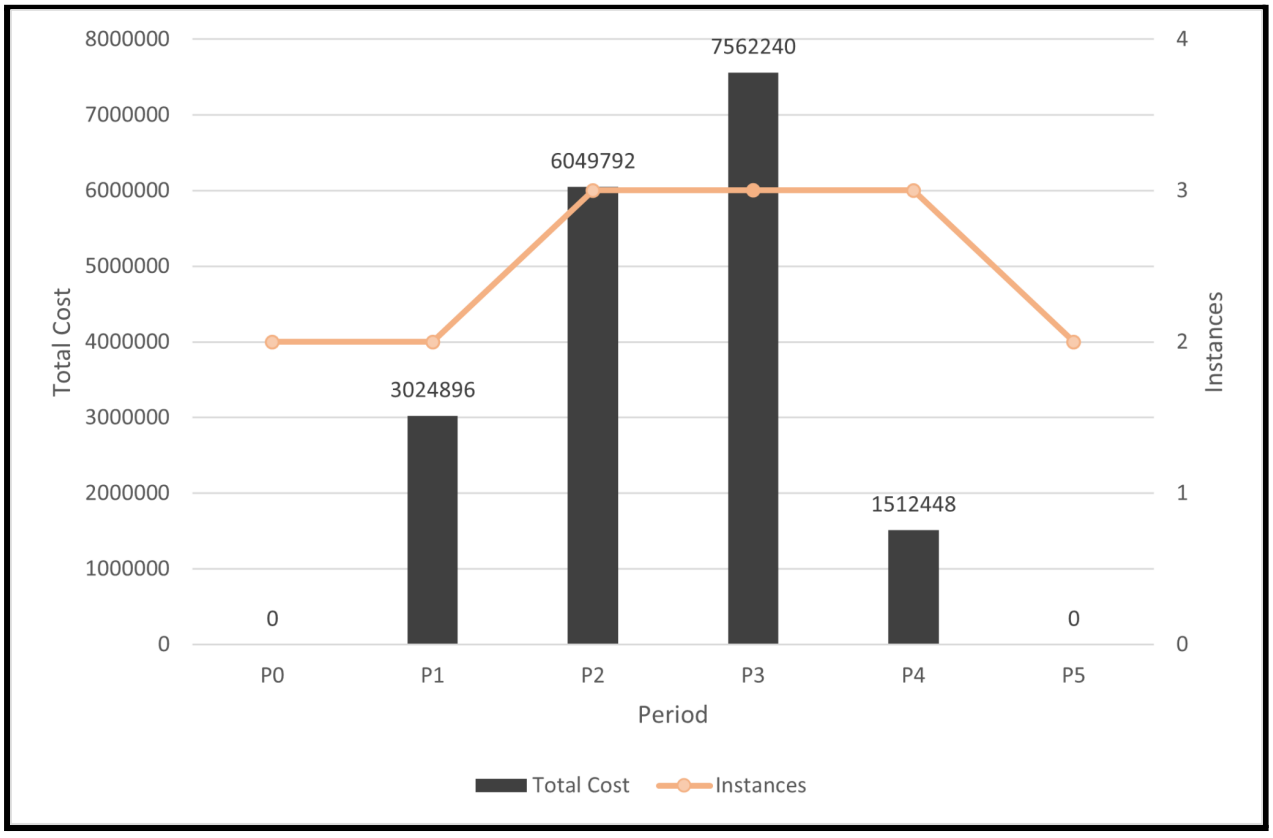
Greedy									
Dataset	512x512			1024x1024			2048x2048		
Scan Range	256x256	384x384	512x512	512x512	768x768	1022x1022	1024x1024	1536x1536	2046x2046
Routine Calls	4.57E+04	4.57E+04	4.57E+04	1.07E+05	1.07E+05	1.07E+05	1.51E+06	1.51E+06	1.51E+06
Stack Depth	5.00E+00	5.00E+00	5.00E+00	5.00E+00	5.00E+00	5.00E+00	5.00E+00	5.00E+00	5.00E+00
Basic Blocks	1.68E+05	1.69E+05	1.69E+05	4.40E+05	4.39E+05	4.39E+05	6.75E+06	6.75E+06	6.72E+06
Stores	8.25E+05	8.27E+05	8.28E+05	2.52E+06	2.51E+06	2.51E+06	4.20E+07	4.20E+07	4.18E+07
Loads	4.35E+06	4.36E+06	4.36E+06	1.53E+07	1.53E+07	1.53E+07	1.33E+08	1.33E+08	1.33E+08
News	9.02E+03	9.09E+03	9.15E+03	1.68E+04	1.68E+04	1.69E+04	2.00E+05	2.01E+05	1.99E+05
Instructions	4.90E+05	4.91E+05	4.91E+05	1.34E+06	1.34E+06	1.34E+06	2.12E+07	2.12E+07	2.11E+07

Correlation	**Routine Calls**	**Stack Depth**	**Basic Blocks**	**Stores**	**Loads**	**News**
	0.99996	NaN	0.99999	0.99188	0.99904	-0.19305

Annex 2: CPU Utilization and measured cost when making requests of different sizes in an instance

CPU	Measured Cost
1.2	45692
2.2	91384
4.2	182768
7.8	365536
19.8	736302
29.8	1104453
42.2	1512448
81.5	3024896
100	6049792

Annex 3A: Graph showing automatic scaling



Annex 3.B: Table showing load balancing

Period	Instance Id	Total Requests
P0	i-008dc0ab847460701 (**New**)	0 Requests
	i-06e4c8aa894a0c4c2 (**New**)	0 Requests
P1	i-008dc0ab847460701	1 Request
	i-06e4c8aa894a0c4c2	1 Request
P2	i-008dc0ab847460701	2 Requests
	i-06e4c8aa894a0c4c2	2 Requests
	i-00bac03a9c7eb1d00 (**New**)	0 Requests
P3	i-008dc0ab847460701	2 Requests
	i-06e4c8aa894a0c4c2	2 Requests
	i-00bac03a9c7eb1d00	1 Request
P4	i-008dc0ab847460701	0 Requests
	i-06e4c8aa894a0c4c2	0 Requests
	i-00bac03a9c7eb1d00	1 Request
P5	008dc0ab847460701 (**Removed**)	___
	i-06e4c8aa894a0c4c2	0 Requests
	i-00bac03a9c7eb1d00	0 Requests

$$Cost\ of\ a\ Request\ =\ 1512448$$