SpecialVFX@Cloud - Group 2

Davi G. Valério
ist1108497
Instituto Superior Técnico
Lisboa, Portugal

Francisco R. Nogueira
ist1108469
Instituto Superior Técnico
Lisboa, Portugal

Chonghe Cui
ist1108077
Instituto Superior Técnico
Lisboa, Portugal

1 ABSTRACT

This project focuses on developing a cloud-based system for a VFX server tasked with image generation and transformation tasks, including raytracing, blurring, and image enhancement. The system objective is to balance cost-efficiency, low latency, and high throughput, balancing the request usage from AWS components such as EC2 instances and Lambda Functions using Load Balancers to apply different loads and Auto Scalers to scale up/down instances. These components tries to ensure a dynamic resource allocation, maintaining performance during peak loads while minimizing costs during low demand periods.

Keywords - Cloud, VFX, AWS

2 INTRODUCTION

The aim of this project is to develop a system for a hyphotetical VFX studio tasked with executing image generation and transformation tasks. The service is designed to support visual effects such as raytracing, blurring and enhancement of pictures. The system must balance cost-efficiency, low latency, and high throughput to meet requirements of the VFX company.

To meet these requirements, the system will be based in an elastic public cloud infrastructure. This includes components such as Elastic Compute Cloud (EC2) instances and Lambda Functions, which are responsible for performing the computations. The architecture also uses Load Balancers (LB) to distribute the incoming load across various servers, and Auto Scalers (AS), which adjusts the number of active servers. This scaling improves latency and throughput during peak loads; and also reduces costs by scaling down resources during periods of lower load. Finally, Data storage will be managed by DynamoDB.

3 ARCHITECTURE DESIGN

In this section, a brief description of the global architecture will be made, then each component will be explained in deeper detail. The design choices and algorithms will be presented.

3.1 Global Architecture

In Figure 1, all the components for the implemented solution can be seen. It is composed by a main controller EC-2 instance, a Metrics Storage System (MSS), the instrumented webserver workers, Lambda Functions and customized metrics on Cloud Watch.

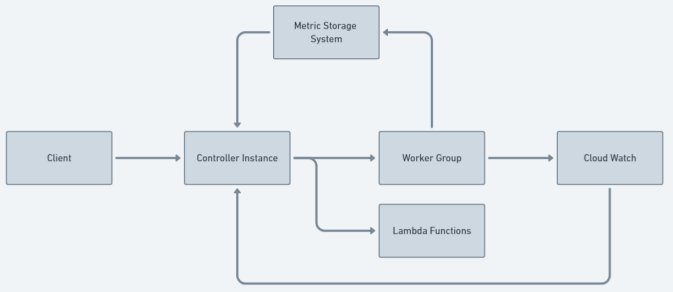

Figure 1: Schematic of the global architecture of the system.

In the controller instance, the Load Balancer receives a request for the VFX server, estimates its complexity and forwards it to a Lambda Function or to a running instance of the worker group; the Auto Scaler periodically fetches the resource usage from the Cloud Watch and may decide to scale up/down the worker group; the Shared Instance Registry manages the interaction between the Load Balancer and the Auto Scaler; and the Metrics Aggregator periodically updates the model for estimating request complexity according to the most recent tasls.

Meanwhile, the Metrics Storage System is a DynamoDB that stores the instrumented characteristics of the requests and the most recent model for calculating the request complexity.

The Webserver workers execute the requests, publishes their characteristics and complexity to the metric storage system and publishes a customized usage metric to Cloud Watch. Finally, the Lambda Functions allow for an alternate method for executing the requests.

3.2 Detailed Architecture and Component Design

The interaction between the detailed components can be summarized in Figure 2. In it, it is possible to see the main responsabilities and data flows.

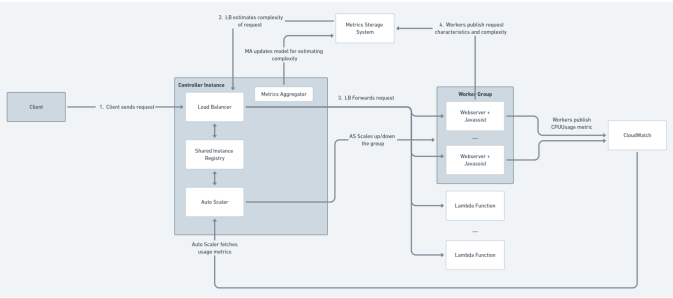

Figure 2: Component of the system.

3.2.1 Load Balancer. When receiving requests from the client, the Load Balancer extracts the type of request and its most important parameters. This will aid in choosing the most adequate way of executing the request. Currently, this component can receive raytracing and two image processing requests (blurring or enhancing an input picture). Raytracing tasks may include the use of textures or not. And experiments showed that the most relevant parameters for estimating the complexity of the request is the specified size of the scene that will be generated. On the other hand, image processing requests' most important parameter is the size of the picture that will be transformed.

After inspecting the type of request, the Load Balancer fetches from the Metric Storage System the latest model for estimating complexity. This model was updated by the Metrics Aggregator and will soon be explained. For any type of request, the fitted model is an univariate linear regression, where the input is the size of the scene in the case of raytracing requests; and the size of the input picture, in case of image processing tasks. After fetching the most updated model, the Load Balancer calculates the expected complexity of this request. One interesting result is the fitted lines for different types of request profiles, which will be inspected in the experiments section.

With the estimate of the request complexity and the type of task, the Load Balancer can now choose how to forward it. To ensure a minimum throughput even in periods of high load, the LB will forward the request to a Lambda Function if the type of task is either blur or enhance. However, to keep costs low, the LB only allows for a maximum of five running Lambda Functions. With this, we aim to ensure that even in periods of high load, simple requests will still be performed with low latency.

In this first implementation of the system, with a limited diversity of requests, the simple requests were defined as the image processing ones. This was done after initial tests which will be described in the experiments section. In future implementations, it will be important to dynamically define which requests are small by evaluating the distribution of complexity of recent requests.

In the case of a raytracing request or if there are five lambda functions executing already, the LB will forward the task to the available instance with the lowest amount of total running complexity. Note that it ignores instances that were marked for termination. After forwarding, the LB updates the Shared Instance Registry with the total amount of complexity and number of requests in the chosen instance. Finally, after the request is complete, it returns the result back to the client and updates the Shared Instance Registry again.

If there are any errors during the executing, the LB returns an error to the client. It is important to note that due to the design choices in the Auto Scaler architecture, the Load Balancer does not need to retry failed requests, since the implementation ensures against errors due to scaling down procedures.

3.2.2 Auto Scaler. Every 10 seconds, the AS fetches from Cloud Watch the CPU usage of each running instance and calculates their average. If the average CPU usage is higher than 90% and the last scale-up happened more than 60 seconds ago, the AS launches a new instance and adds it to the Shared Instance Registry.

Then, the AS checks between the instances that were previously marked for termination if they have any requests running. If they are not performing any task, the AS terminates them and updates the Shared Instance Registry. If any instance has a CPU usage lower than 15%, the AS marks it for termination, but does not immediately shuts it down. It will instead wait for it to drain down and only choose to terminate when it is not executing any request.

It is also important to note that the AS garantees that there is always a minimum of one instance running. This ensures that the worker group is always able of performing raytracing requests. Also, only one instance can be marked for termination at each time.

The thresholds of CPU usage and the cool-down period were set arbitrarily as a "high" and "low" values. Setting them to different values such as 80% and 20% would yield similar results. In future implementations, these values can be fine tuned by applying predictive techniques in the Auto Scaler algorithm, which was not the focus of this work.

3.2.3 Worker Instances. The main purpose of the workers is to execute the tasks and to return the output to the user (through the LB). However, to aid the LB in load distribution, the instance instruments the requests characteristics and tracks their execution. The goal of this instrumentation is to provide the LB with a method for predicting the complexity of incoming requests.

Specifically, the worker instances extract from the task: the type of request (between raytrace, blur or enhance), if it uses anti-alias or texture maps, the size of the specified scene for raytracing tasks and the size of the image in the case of image processing requests. In addition, it measures the number of executed instructions and basic blocks and the time it took to finish the task. All these parameters are published to a table in the MSS. They will be used by the Metrics Aggregator to build a model for estimating the complexity of incoming requests.

Also, every ten seconds, the worker instances pulishes their CPU usage to cloud watch as a custom metric. This was used to achieve a higher frequency of updates for the resource consumption, rather than the default frequency provided by Cloud Watch. With this, the Auto Scaler reacts quicker to an increase in load.

3.2.4 Metrics Aggregator. The purpose of the metrics aggregator is to update the model used in calculating the expected complexity of new requests. To do so, it fetches the most recent requests and calculates a weighted complexity value by doing $0.7 \cdot \text{numInstructions} + 0.2 \cdot \text{numBasicBlocks} + 0.1 \cdot \text{processingTime}$. Then it fits a univariate linear regression model for each different type of request. In the case of raytracing tasks, it uses the number of pixels of the specified scene size as input and the complexity measure as output. For image processing requests, it uses the size of the payload picture as input and the same output as before. It is important to note that, although one different line was fitted for each type of input, the same normalization was used globally. This was made to ensure that the models were compatible between each other. Finally, the MA publishes the fitted line for each type of request in a table in DynamoDB.

3.2.5 Metrics Storage System. The MSS has two tables: one that stores the instrumentation results for each type of request and one that stores the fitted models for each type of request.

4 EXPERIMENT RESULTS

To test the system, the following request profile was prepared. For the image processing tasks, two JPEG images were used, a simpler and a more complex one. The simpler has 658, 560 pixels, but a low amount of detail (16kB). The more complex is smaller, has 615, 360 pixels, but a high amount of detail (80kB). For the raytracing tasks a similar approach was taken. One simpler scene with three spheres and no backround or reflection will be named the ”simple” scene; while another with three spheres, reflection, depth, perspective and the use of textures will be named the ”complex” scene.

In this experiment, a load of 1000 tasks was applied to the system using a specific combination of simple and complex requests. Their complexity and parameters will be recorded and the fitted line for estimating complexity will be calculated. In addition, the time to finish each request will be measured and the CPU usage of the instances will be assessed. We will also aim to see when the system scales up and down, to analyse its reactiveness.

Finally, it must be pointed out that a t3.micro EC2 instance was used to build the images of the workers.

4.1 Instrumented parameters and complexity

In Tables 1 and 2, the average values of the parameters for each type of request can be seen.

Table 1: Instrumented Metrics

Type	Basic Blocks	Instructions
BLUR	7	303
ENHANCE	7	295
RAYTRACER	129711.10	4422540

Table 2: Instrumented Metrics (cont.)

MetricType	Image/Scene size
BLUR	658560
ENHANCE	615360
RAYTRACER	120000

In Table 3, the time to complete each type of request, broken down by simple and complex, is presented.

Table 3: Time to complete by request type.

Type	Interval
simple blur	7.30
complex enhance	10.2
simple raytracer	137.31
complex raytracer	133.20

The fitted line for raytracing requests was $y = 0.0058x + 94$; for enhancing, $y = -0.0027x + 0.0039$; and for blurring, $y = -0.0071x + 0.0086$. These lines can be seen in Figure 3.

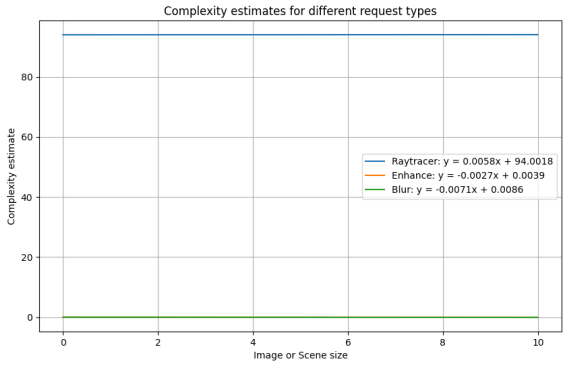

Figure 3: Fitted complexity estimate models.

4.2 Latency Study

In Figure 4, the CPU Usage of each running instance can be viewed. For this test, the system did not decide to scale up, and the CPU usage of one instance never went above 80%.

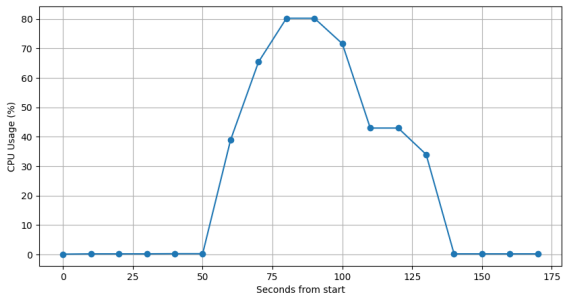

Figure 4: CPU Usage over time.

In Figure 5, the plot the plot shows on the horizontal axis the time of start of the request; and on the vertical axis the time it took to be completed.

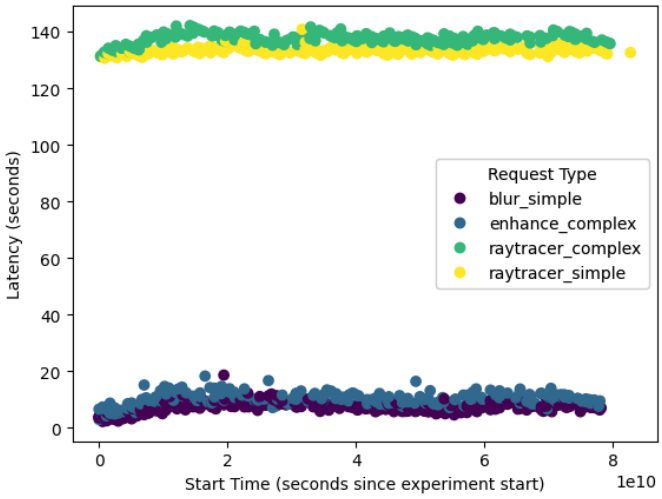

Figure 5: Time to finish execution by request type

In Figure 6, the plot shows on the horizontal axis the time of start of the request; and on the vertical axis the time when it ended. It has the same legends as the previous chart.

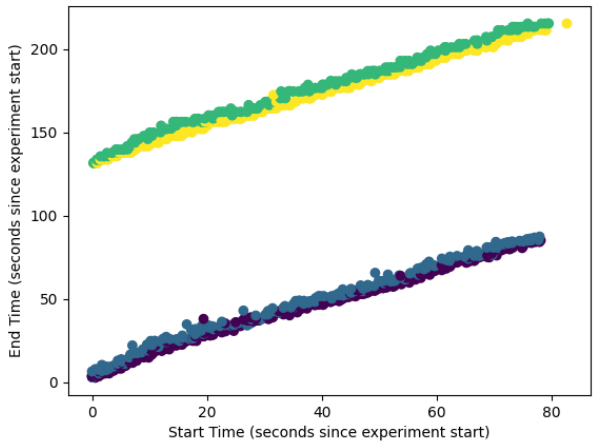

Figure 6: Time of start and end by request type

5 DISCUSSION

First, it can be seen in the instrumented metrics, Table 1, the number of basic blocks and of instructions of image processing tasks is much lower than for raytracing requests. In addition, the instrumented processing time is also lower. This happens even if the size of the image is almost six times bigger for image processing than for raytracing, as it is the case.

As a result, it can be viewed in the fitted model that the image size was not a good predictor for request complexity. That is, even with an increase in image size, the complexity of raytracing tasks is much bigger than that of image processing tasks. Thus, the best predictor of this measure was the request type.

It can be argued, however, that the weighted complexity metric $0.7 \cdot \text{numInstructions} + 0.2 \cdot \text{numBasicBlocks} + 0.1 \cdot \text{processingTime}$ may not adequate. This is because it does not represent well the latency of the tasks. In other words, the complexity metric does not represent well the time to process for each type of request.

It must be noted that the authors did not choose to directly use processing time as the request complexity measure because this is variable with the available compute and other factors that do not depend only of the nature of requests. The previous comparison of latency can only be done with large samples, as it is the case of this experiment.

On the other hand, the strategy of prioritizing Lambda Functions for small requests seem to have maintained the instance usage low. Since the CPU Usage did not go above 80% during the experiments, it can be noted that the method helped in keeping the execution of the image processing requests at a very low latency. In other words, the demand of higher complexity tasks did not impact the performance of the system on the simpler tasks.

The previous conclusion can also be verified in Figures 6 and 5, where even in periods of high load, the simple requests were completed with low latency.

6 CONCLUSION

This project tackled the development of a scalable and cost efficient cloud based system, capable of handling various image generation and transformation tasks such as raytracing, blurring, and enhancement. Utilizing AWS cloud infrastructure, including EC2 instances, Lambda Functions, Load Balancers, and Auto Scalers, the system dynamically adjusts resources to maintain high performance and low latency. The integration of the Metrics Storage System and Metrics Aggregator allowed for continuous optimization of the request complexity model, ensuring accurate load predictions and efficient resource utilization.