Utilising Scalable Cloud Architecture for Render Farms

Paolo Mura

University of Bristol, Bristol BS8 1UB, UK 19763@bristol.ac.uk

Abstract. This paper introduces an application that serves as a render farm in the Cloud, predominantly making use of AWS infrastructure. The central component of the application architecture is a Kubernetes cluster on which a Docker image is deployed. Other components in the system support the cluster to provide interactions with an end user as well as data storage systems. Horizontal autoscaling enables the work to be dynamically scaled up and down to meet demand. Fault tolerant components enable the whole job to be completed, even after instance failure.

Keywords: Cloud Computing · Render Farm. · Autoscaling · Fault Tolerance

1 Introduction

3D graphics rendering is a compute-intensive task that is embarrassingly parallelisable. With multiple objects in the scene for light to reflect or refract off of, together with multiple light sources and high-resolution images, these factors accumulate to give a compute-heavy task. Rendering even a moderately sized project can take hours and in industry, CGI film renders take years to compute.

Since each frame of an animation is rendered independently of each other, rendering lends itself naturally to parallelisation. Multiple frames can be rendered in parallel on different cores. To take this a step further, a cluster of machines with potentially several cores each are able to parallelise the task on a much larger scale. Although a single user may only have one render job to submit at a time, multiple users not only increases demand, but adds more variation to demand across time.

Large film studios tend to use their own in-house hardware for rendering. These servers or supercomputers are known as render farms and help cut down time costs thanks to distributing the rendering work across thousands of cores over many separate machines. For individuals, access to such facilities was historically limited. However in recent years, cloud computing has evolved to provide remote access to compute resources with affordable pricing via an on-demand system. Through the IaaS (Infrastructure as a System) model, individuals are able to access virtualised IT infrastructure to perform their rendering tasks, making full use of the service's parallel, distributed nature. The scalability of cloud compute resources gives flexibility to the rendering needs of many individuals.

2 Architecture

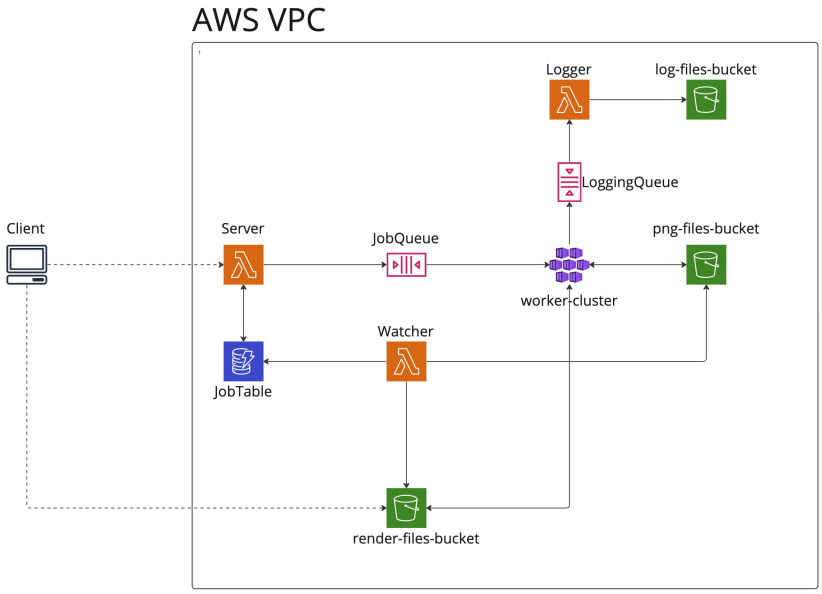

Fig. 1. Architecture diagram for the cloud render farm. Components are coloured as follows: AWS Lambda Functions (orange), SQS (red), Kubernetes cluster (purple), S3 buckets (green) and DynamoDB table (blue).

2.1 Components

The complete architecture is structured as illustrated in Fig. 1. The flow of a given render job through the application is then outlined in Fig. 2. "Jobs" (or "batches") are sent through the system as small JSON objects in the following format:

```
{
    "type": "render" | "sequence",
    "file": "<filename>.blend",
    "start": "<int>",
    "end": "<int>"
}
```

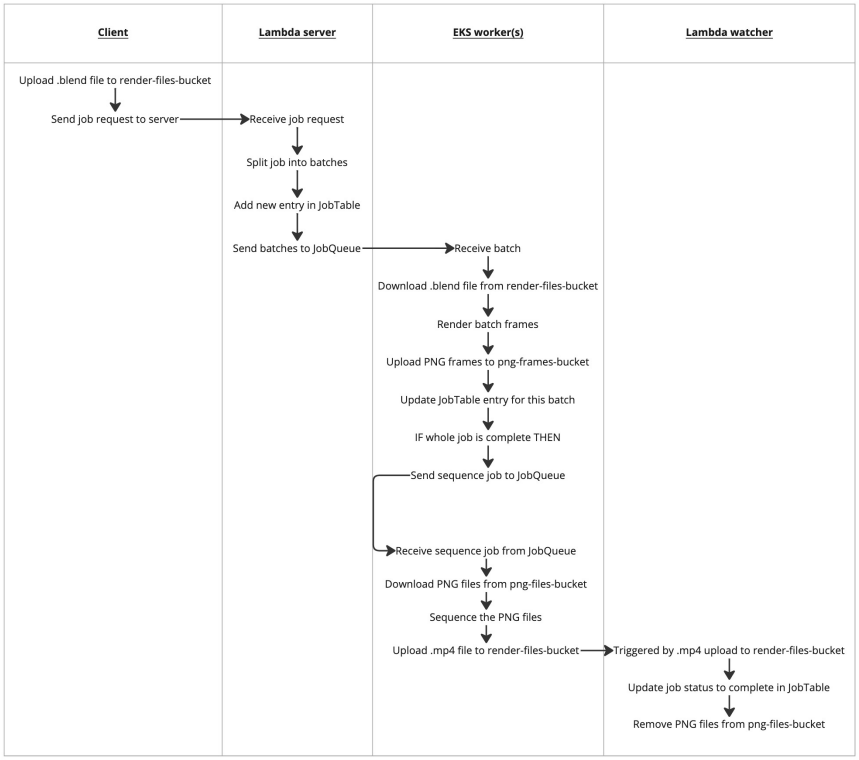

Fig. 2. Sequence diagram for the render farm. Only Lambda functions and EKS workers are included. The diagram is a high level of abstraction that only focuses on the work and its flow through the system.

S3 Buckets I used S3 buckets for persistent storage of the input .blend files and final output .mp4 files as well as the intermediate .png files generated between rendering and sequencing.

DynamoDB Table DynamoDB is a NoSQL database that provides fast access to items via a key-value method. The render farm uses its document database functionality to store the status of each job as well as the status of each of its batches. As with many of the other components, it made sense to use an AWS service rather than alternative like MongoDB [1] since DynamoDB could easily interact with other components via boto3 and AWS CLI.

SQS Queues Two SQS queues are used to send messages to and from the EKS worker cluster. One of the main features is the ability to include a message visibility timeout which prevents other consumers from being able to access a message once it has been read until the timeout period has passed and the message has not yet been deleted. This is a key aspect of fault tolerance as discussed in section 2.4. The scalability and low cost discussed in [2] were also factors for choosing SQS over alternatives like Kafka.

Lambda Functions Lambda functions provide a method for running simple code when triggered by an event. This is a much cheaper and more efficient method than running a full-time server with [3] highlighting Lambda's benefit over EC2 for these use-cases. In this render farm, I have made use of Lambdas for cases where a small amount of work needs to be performed at irregular times (specifically for handling user requests, complete jobs and worker logs; all of which occur far less often than the actual job processing itself).

EKS Cluster The EKS worker cluster makes up the central data processing component of the render farm. Its nodes run deployments from Docker images (see section 2.2). Unlike a custom EC2 cluster, it has built-in autoscaling and fault tolerance that allows the system designer to focus on application development rather than orchestration.

2.2 Containerisation

I used Docker to create an image for the worker code and then hosted this in an ECR repository for easy integration with the AWS-hosted Kubernetes, EKS. The Docker image made it easy to update versions; by carefully specifying the order of Dockerfile commands, only the items that change often (code and access credentials) got updated thanks to Docker's layered image approach. Another benefit of this is that I was able to set a parent image that already provided the Blender dependencies.

2.3 Autoscaling

I applied horizontal pod autoscaling to the cluster to allow it to deploy more pods across the cluster to handle increased demand. It is implemented as a Kubernetes

resource itself, which periodically fetches information about the CPU usage from the cluster. If above the set threshold (50%), it increases the number of pods until the percent CPU usage reaches the threshold. On the other hand, if the percent CPU usage drops below 50%, the autoscaler will reduce the number of pods down to its minimum, which helps cut costs and reduce redundancy. The formula for this behaviour is

$$replicas = \lceil currentReplicas \cdot (currentCPUusage/thresholdCPUusage) \rceil \quad (1)$$

2.4 Fault Tolerance

EKS also provides fault tolerance through managed node groups. A node group is a virtual collection of instances within the cluster. On creation, I specified the desired number of nodes as well as the minimum and maximum allowed. During the application's lifetime, the cluster aims to maintain the desired number of nodes. On startup, it spins up instances to match this number and if a node develops a fault, the cluster automatically creates a new instance to replace it.

In order to protect jobs from instance failures, I made use of the SQS message visibility timeout. This means that when a job message is read by a worker, it is then hidden from all other consumers until the timeout has elapsed. In ordinary circumstances, the worker reads the job, completes it and then deletes the message from the queue. By setting the visibility timeout to a value greater than the normal job completion time, if the worker node fails during processing and is unable to complete their task, the message will be re-queued after the timeout. This gives other workers the chance to process the job instead.

3 Testing

In preparation for testing, I added simple logging methods to the system. These included a timestamp of the job submission in the response from the server Lambda function as well as logs sent from the worker nodes via an SQS queue to another logging Lambda function, which deposited them as CSV files in the worker-logging-bucket S3 bucket.

To test the performance of the application, I uploaded a test .blend file to the render-files-bucket S3 bucket containing a simple animation of a rolling ball. I then submitted a Lambda Test to the server function containing a job to render the first 12 frames of the animation.

I also ran the same job locally using a Python script that also split the job into batches but performed all rendering sequentially. This is because each worker node sends a message to the LoggingQueue SQS queue after it processes each batch along with a timestamp; a metric that could be used to analyse throughput.

To test the scalability of the cluster, I ran a local Python script that periodically called a kubectl command that returned the status of the pods and CPU usage, then piped this output to a .txt file for later analysis.

In order to test the fault tolerance of the system, I manually used the AWS Web Console to determine an EC2 instance that was running one of the current pods via the EKS page. I then terminated this instance to apply fault injection and simulate a malicious or accidental defect.

4 Analysis

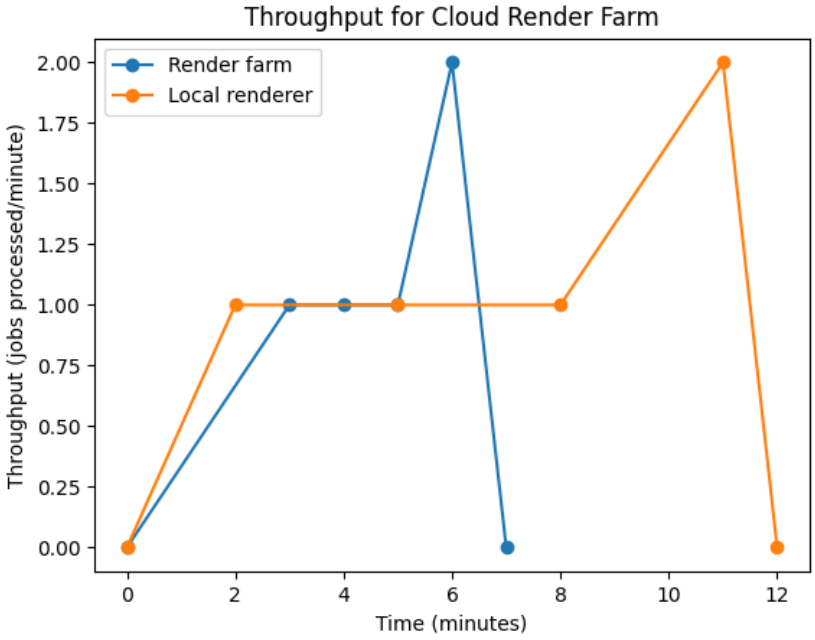

Fig. 3. Comparison of the local render job vs one submitted to the render farm. Both jobs were of the same file and frame range (1-12).

After performing the simulated tests as described in section 3, I downloaded the CSV files generated by the workers. I then passed their filenames together with the start time from the server response as command line arguments to a Python analysis script. This extracted the relevant data from the files and plotted graphs as shown in Fig. 3 and Fig. 4.

Fig. 3 clearly shows that although the render farm took slightly longer to start processing batches, it was able to complete the full job almost twice as fast as the local render. Each batch in the local render took around 3 minutes to process sequentially. Although the render farm processed the batches in parallel,

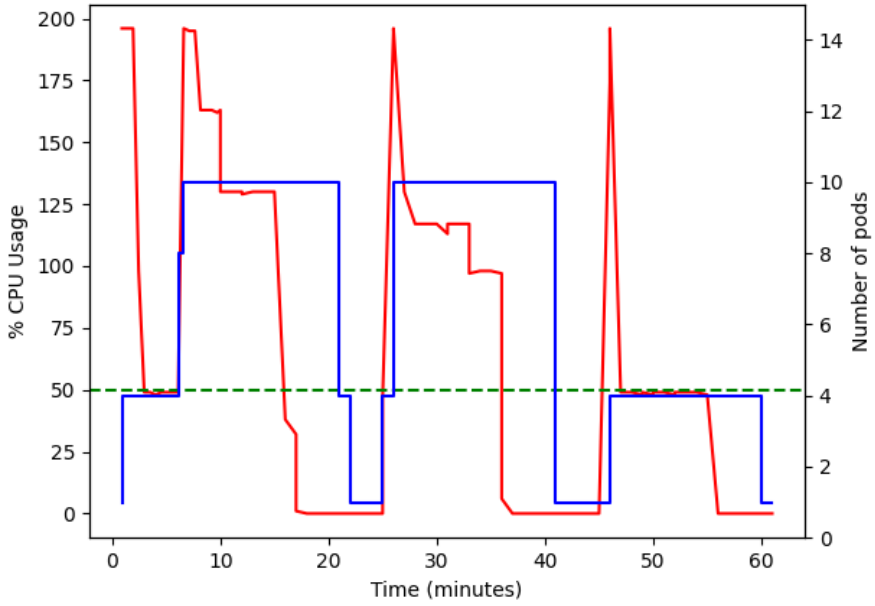

Fig. 4. Measurements of both CPU usage and number of running replicas against time. Two equal jobs were submitted, one at 6 minutes and the second at 25 minutes. An EC2 instance was terminated at 28 minutes. The cluster created a new instance at 30 minutes and it was in a running state by 33 minutes. A dashed green line has been included to mark the CPU threshold of 50%.

the blue marks between 2 and 6 minutes indicate that the workers had a small difference in throughput each of approximately 1 minute.

Fig. 4 demonstrates both the scalability and fault tolerance of the system. After the initial job was submitted (at around 5 minutes), the CPU usage quickly increased to 200%. The number of pods rapidly increased to meet this demand. Once the first job had been processed, CPU usage dropped below the threshold and the number of pods then reduced to the minimum, showing that autoscaling was working for both increases and decreases in demand.

The second test (which was started around 25 minutes) followed a very similar behaviour. However, 3 minutes later an instance was terminated, which is reflected by the drop in CPU usage. 5 minutes after this, another instance was created to replace it. There is another spike in the graph just after 45 minutes, showing that the job that the terminated worker had been processing got successfully accepted by another worker (since the visibility timeout period elapsed). A visual inspection of the graph reveals that this second job took at least 15 minutes longer than the first test that ran without fault injection.

5 Conclusions

I successfully constructed a working render farm using cloud technologies and demonstrated both its autoscaling functionality and fault tolerant worker cluster. There is an initial startup delay when compared with local rendering, but the overall time taken to complete a given task is much faster thanks to the parallelised, distributed approach.

There are several areas for improvement with this application, specifically targeting its efficiency. For example, shared volumes could be made better use of in the EKS nodes so that two pods deployed on the same node could share access to the same .blend file. Following on from this, it would then make sense to adapt the default load balancer to distribute similar work to pods on the same node. The horizontal pod autoscaler could also be further refined. It is likely that it currently deploys more pods than necessary and that some of them don't actually run due to insufficient resources. This could be amended by using a more sophisticated autoscaler algorithm than comparing CPU usage paired with a vertical autoscaler that scales up number of nodes when resource allocation is insufficient.

References

1. BMC, MongoDB vs DynamoDB https://www.bmc.com/blogs/mongodb-vs-dynamodb/. Last accessed 11 Dec 2022
2. CloudInfrastructureServices, Kafka vs SQS - What's the Difference? (Pros and Cons) https://cloudinfrastructureservices.co.uk/kafka-vs-sqs-whats-the-difference/. Last accessed 11 Dec 2022
3. Medium, AWS Lambda Vs AWS EC2: How to choose one over another? https://promatics.medium.com/aws-lambda-vs-aws-ec2-how-to-choose-one-over-another-77e6380ac8fc. Last accessed 11 Dec 2022

6 Appendix

Link to repository with source code: https://github.com/ccdb-uob/CW22-55