Performance comparison of ASG and EKS based infrastructure in AWS cloud.

A thesis presented in partial fulfilment of the requirements for the degree of

Master of Information Technology

At Whitireia Polytechnic, Porirua, New Zealand

Pavel Grigoryev

2022

ABSTRACT

With the move of web applications to more distributed service-oriented architecture the drawbacks of VM-based infrastructure such as the lack of consistent application packaging standards and big virtualization overhead has accelerated the evolution of container-based solutions. The benefits of container-based infrastructure such as reduced virtualisation and storage overhead and faster start-up times are obvious in bare-metal solutions. However, when implemented in the cloud, container orchestration tools like Kubernetes must be run on top of VM-based infrastructure which partially negates their performance benefits as compared to running applications on pure VM-based infrastructure.

The purpose of this study is to compare the performance and scaling capabilities of container-based infrastructure (AWS Elastic Kubernetes Service) and VM-based infrastructure (AWS Elastic Compute Cloud) in AWS cloud and give recommendations on which infrastructure platform offers better performance. This study employs experimental methodology to assess the performance, where a set of test applications is deployed to each infrastructure platform and their performance is measured via a load testing tool.

The results of the experiment show that neither of the infrastructure platforms has definitive advantage over the other as the performance profiles vary a lot between applications. The Apache application has better performance and efficiency on ASG infrastructure. Both Node.js applications (Taewa and Riwai) show marginally better performance on EKS infrastructure, however with much lower efficiency. And Python application (Raupi) shows better performance and efficiency on EKS infrastructure.

An attempt was made to isolate the reason for these variations by examining the contribution of factors such as programming language, web framework, concurrency, difficulty, and pod density. The results suggest that the programming language is the determining factor of the application performance profile while other factors do not seem to have a consistent effect on performance.

ACKNOWLEDGEMENTS

I am extremely grateful to my supervisors Dr Marta Vos and Dr Scott Morton for taking me through this journey and motivating me. Marta's patient feedback and guidance helped me navigate the turns that happened on the way and made this thesis possible.

I am also grateful to all staff of Whitireia who made my life easier and allowed me to concentrate on my study.

Lastly, I would like to acknowledge my family and especially my wife Yile. Thanks for being by my side and encouraging me to start this journey!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

Application Load Balancer	ALB
Auto Scaling Group	ASG
Availability Zone	AZ
Command-Line Interface	CLI
Elastic Load Balancing	ELB
Elastic Kubernetes Service	EKS
Horizontal Pod Autoscaler	HPA
Information Technology	IT
Infrastructure as Code	IaC
JavaScript Object Notation	JSON
Kubernetes	K8s
Launch Template	LT
Load Balancer	LB
Operating System	OS
Security Group	SG
Virtual Machine	VM
Yet Another Markup Language	YAML

CHAPTER 1

INTRODUCTION

1.1 BACKGROUND

As modern distributed applications become highly scalable and require a more elaborate software development lifecycle, the limitations of Virtual Machine (VM) based infrastructure become more apparent and companies move to container-based solutions such as Docker and Kubernetes.

VMs have been used for abstracting the hardware and isolating applications since 1960's (Brawn & Gustavson, 1968). As VMs provide an environment closest to the developer's workstation they are a convenient choice for quickly building an application prototype, but when it comes to establishing a software development lifecycle that includes multiple environments (development, test, staging, production) and running applications at scale they have several drawbacks. Besides the lack of consistent application packaging and deployment standards. Slow start-up times affect scaling of the infrastructure and big virtualization overhead results in degraded performance (Luksa, 2017).

These drawbacks especially affect distributed applications, e.g., ones designed with microservice architecture style which has become widely adopted in the industry (Vayghan et al., 2019). In microservice architecture each microservice represents a business function, is self-contained and scaled and deployed independently. Since this architecture style requires running a large number of lightweight isolated processes the virtualization (isolation) overhead for each process has a big impact on hardware utilization efficiency and performance (Vayghan et al., 2019).

Containers are a relatively new technology that helps to mitigate some of the drawbacks of traditional (hardware) virtualization. Containers are an example of OS-level virtualization where containerised processes share the same kernel. This offers near-instant start-up times (microseconds as opposed to minutes for VMs) smaller virtualization overhead and smaller storage footprint as only the application itself and the libraries need to be packaged (Rad et al., 2017). Besides performance improvements, containers offer other benefits such as improved security due to reduced attack surface and improved portability (package once – run anywhere). Application portability allows developers to test their applications locally without the need to wait for the deployment to the testing environment and allows to achieve better consistency between the environments (Miell & Sayers, 2019).

However, with containers being transient by design, container lifecycle management can get complicated. A common way to deal with container management complexity is by using container orchestration tools. Kubernetes (also abbreviated as 'k8s') is one of the most prominent container orchestration products (Al Jawarneh et al., 2019). It helps to address the complexity of containerised application infrastructure by introducing an abstraction layer so that instead of dealing with containers directly the developer will mostly interact with entities like services that provide service discovery and load-balancing capabilities; and deployments that help to manage the container lifecycle.

As drawbacks of VM-based infrastructure for distributed applications (discussed above) become more apparent and container orchestration tools become more mature, researchers started to look at migrating an existing application from VM-based infrastructure to Kubernetes based. Imran et al. (2021) described the migration of the services that are used to process data coming from the Compact Muon Solenoid (CMS) detector of the Large Hadron Collider (LHC) from VM based infrastructure to Kubernetes. They discussed the design and implementation strategy of the Kubernetes infrastructure and compared the performance of the original (VM) and target (Kubernetes) infrastructure using different metrics finding that Kubernetes performs better in most of the metrics.

However, the CERN project implements Kubernetes infrastructure on top of their OpenStack-based private cloud which is not a very common infrastructure provider. Therefore, since Amazon Web Services (AWS) is much more prevalent and easier to use than OpenStack (Honig, 2019), this research will look at comparing performance and scaling capabilities of VM-based and Kubernetes-based deployments in AWS cloud.

1.2 RESEARCH PROBLEM

Modern distributed applications require efficient ways to isolate the application processes as well as providing better portability of the applications. Container orchestration tools like Kubernetes can help to satisfy this demand. However, to determine the optimal infrastructure platform (VM-based or Kubernetes -based) the technical decision makers need to consider the performance capabilities of initial and target platforms among other factors. As discussed in the background, some teams have already performed migration from VM-based to Kubernetes-based infrastructure in OpenStack private cloud and showed significant benefits of migrating to Kubernetes. However, this hasn't been proved for AWS cloud.

1.3 RESEARCH QUESTION AND OBJECTIVES

This work addresses the question “Which AWS infrastructure platform VM-based or Kubernetes-based offers better performance and scaling capability for web applications?”. To answer this question two infrastructure platforms AWS Elastic Compute Cloud (EC2) and AWS Elastic Kubernetes Service (EKS) are compared. The type of EC2 resource that is used for managing groups of EC2 instances is called Auto Scaling Group (ASG) (Amazon Web Services, 2021b). ASG is the main EC2 entity that will be analysed in this work, so for convenience terms “ASG platform” and “EC2 platform” will be used interchangeably.

The objectives of this research are as follows:

- design and build a VM-based infrastructure using AWS Auto Scaling Groups (ASG)
- design and build a Kubernetes-based infrastructure using AWS managed Elastic Kubernetes Service (EKS)
- develop and package simple web applications to serve as simulated workload
- design and build an automated testing solution to compare scaling and request throughput performance of each platform
- run a series of tests to compare scaling and request throughput performance of each platform and analyse the results.

1.4 SIGNIFICANCE OF THE STUDY

As there currently is a lack of literature comparing the performance of applications running on AWS ASGs and EKS clusters the findings of this project can help organizations using AWS as their cloud services provider to decide which infrastructure platform they should choose for their web workloads, or whether to migrate their existing VM-based workloads to Kubernetes.

1.5 THESIS ORGANISATION

Chapter 1 provides an overview of this research followed by research significance and the problem statements. This chapter also presents the objectives of the research.

Chapter 2 introduces important concepts and gives an overview of virtualization and containerisation technology as well as containerisation services in AWS. Then, existing research regarding running containers in the cloud is discussed.

Chapter 3 elaborates the research methodology, outlines experimental setup, and describes the dependent and independent variables as well as the controls that were implemented to ensure the accuracy and consistency of the results.

Chapter 4 covers the specifics of the implementation of the experiment. It starts with the overview of the automated testing solution and the infrastructure architecture. It then gives a detailed description of each component including infrastructure modules, deployment automation, test applications, testing procedure and testing automation.

Chapter 5 describes how the data is collected, processed, analysed, and presented.

Chapter 6 presents and discusses the findings of the research.

Chapter 7 summarises the findings and limitations and suggests possible future directions for the research.

CHAPTER 2

LITERATURE REVIEW

This chapter will cover in more detail the types of virtualization technologies and their implementations as well as existing work in this field.

2.1 VIRTUALIZATION

Virtualisation technology is used to make more efficient use of hardware resources. It allows for isolation of processes running on the same physical machine thus improving security and preventing the processes from interfering with each other. It is also widely used in cloud computing to provide multitenancy capability (Bernstein, 2014).

2.1.1 Hypervisors

Traditionally, virtualisation has been achieved via the use of hypervisors – the software that creates and runs virtual machines. In case of hypervisor virtualisation each virtual machine runs a copy of its own "guest" OS, which can be different from the host OS that the hypervisor is running on as shown in Figure 2-1 below.

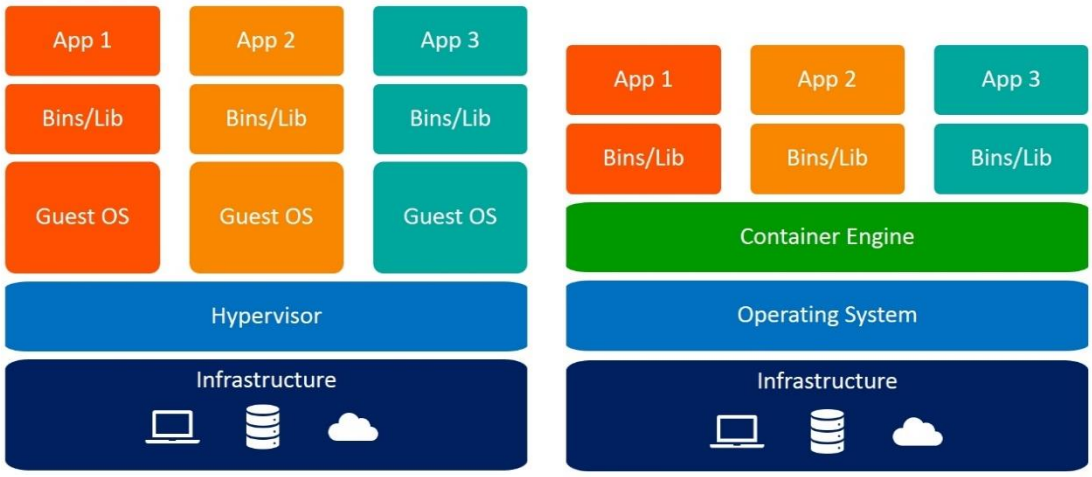


Figure 2-1 Hypervisor-based vs Container-based virtualization

While having the general VM benefits described above, hypervisor-based virtualization has a number of drawbacks such as slow start up times due to time needed to boot guest OS, and inefficient use of

hardware resources as every VM has to have permanently allocated RAM/drive space, occupied even when nothing is running.

2.1.2 Containers

Containerisation on the other hand is an example of so-called OS virtualization, where the processes are sharing the same OS kernel (see Figure 2-1). Since containerised processes are running directly on the host OS kernel, there is only minimal overhead for starting them. Therefore, start-up times in a containerised environment are on the scale of microseconds rather than seconds or even minutes in case of hypervisor-based environments (Rad et al., 2017). This also means that container storage footprint is much smaller because there is no need to package the guest OS in the container. Moreover, resource utilization is much more efficient as the resources are freed up as soon as the containerised process is terminated.

Not every OS will support containers as the OS kernel must have the features that allow it to effectively isolate the application processes. Products like LXC, Docker and Kubernetes use Linux kernel containment features like namespaces and cgroups for this purpose. Although namespaces and cgroups have been around for a long time (cgroups were introduced in 2007 and namespaces in 2002) it took some time for the modern container ecosystem to take shape (Bernstein, 2014). In the early days Docker used LXC as Linux kernel interface while Kubernetes used Docker as its container runtime. But later the Docker removed support for LXC and implemented its own kernel interface (Docker, 2021) while Kubernetes implemented Container Runtime Interface (CRI) to work with variety of container runtimes (The Linux Foundation, 2022a). To make sure container formats are standardised and the same container can be run by different runtimes, the Open Container Initiative (OCI) had been launched (Espe et al., 2020). On the Figure 2-1 below you can see the current state of the components of the containerisation stack and how they are related to each other. For example, it is shown that Kubernetes employs

containerd[1] runtime via Container Runtime Interface API and containerd invokes *runc* to spawn the containerised application processes.

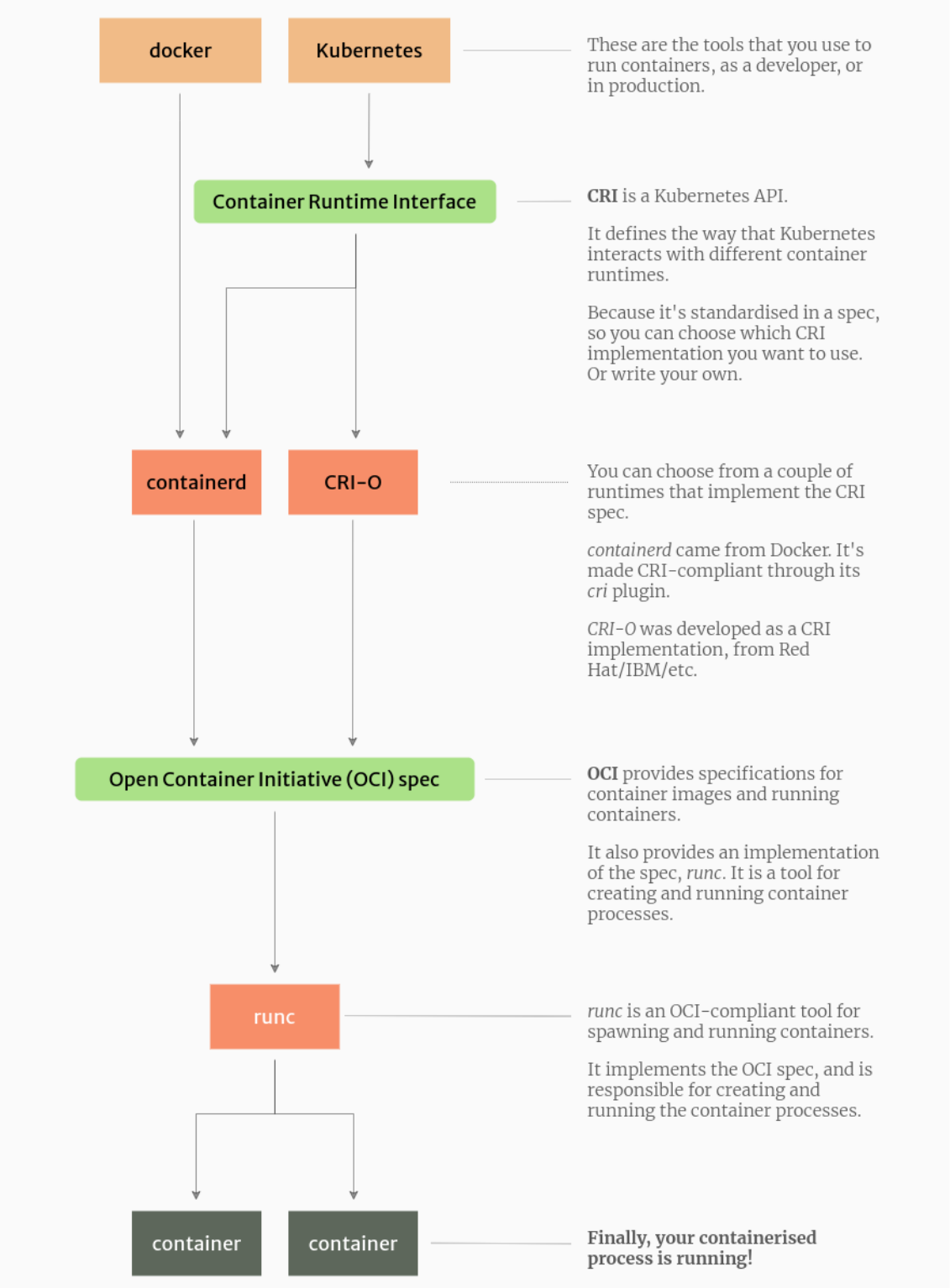


Figure 2-2 Containerisation software stack (Donohue, 2020).

[1] 'contanerd' is the name of a container runtime developed by Docker (Espe et al., 2020)

One of the challenges of containerisation is container lifecycle management complexity. The philosophy of containers is to have one process per container to ensure separation of concerns and straightforward container health detection as well as to simplify log management and scaling (Benevides, 2016; Merkel, 2014). That means containers lack the usual Linux process management tools like systemV or initd. Therefore, the container lifecycle management becomes the responsibility of the container orchestration tools such as Kubernetes.

2.2 KUBERNETES

2.2.1 Kubernetes Basics

As discussed in the introduction, Kubernetes simplifies container management by introducing the abstraction layers. Figure 2-3 Kubernetes objectsFigure 2-3 illustrates the relationships between entities representing these abstractions.

At the lowest level of abstraction are the pods each containing one or more tightly coupled containers while providing networking (IP address) and storage (volumes) capabilities. Pods can be organised into ReplicaSets allowing to maintain a number of pods as specified by a template. However, ReplicaSets are very simple objects and rarely utilised directly. Instead, a more high-level object called "deployment" is usually employed. Besides maintaining the desired number of pods Kubernetes deployments can facilitate rollouts updated versions of the applications or rollbacks to the previous versions as they keep the version history of the pod specifications.

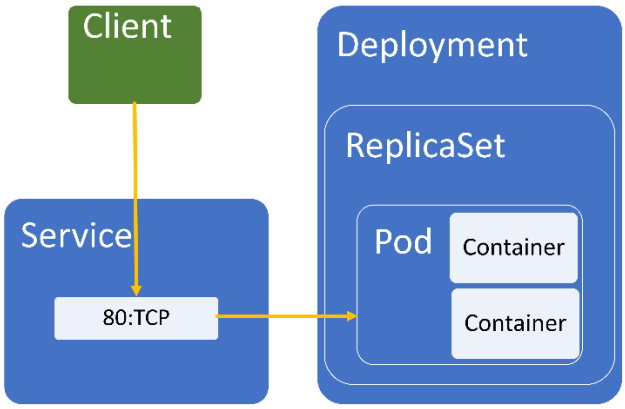


Figure 2-3 Kubernetes objects

Another important Kubernetes object is a Service. Much like conventional load balancers Kubernetes services provide a single endpoint for pods in a deployment. Services are important because pods are

considered to be disposable objects which may fail and be restarted or get transferred to other nodes in the cluster which will cause their IP address to change. Therefore, the applications running on Kubernetes are usually accessed via the service DNS name (rather than the pod IP addresses directly) and the service controller keeps track of which pods are available (Burns et al., 2019; The Linux Foundation, 2022d).

In terms of the Kubernetes architecture, two main components of the Kubernetes cluster are the control plane which consists of Kubernetes system services and the data plane, represented by worker nodes (or simply nodes) that run the containerised applications (see Figure 2-4). The central component of the control plane is the API server which serves a frontend of the Kubernetes cluster. The API server mediates interactions between all other Kubernetes cluster components and serves as an endpoint for management tools (e.g., kubectl).

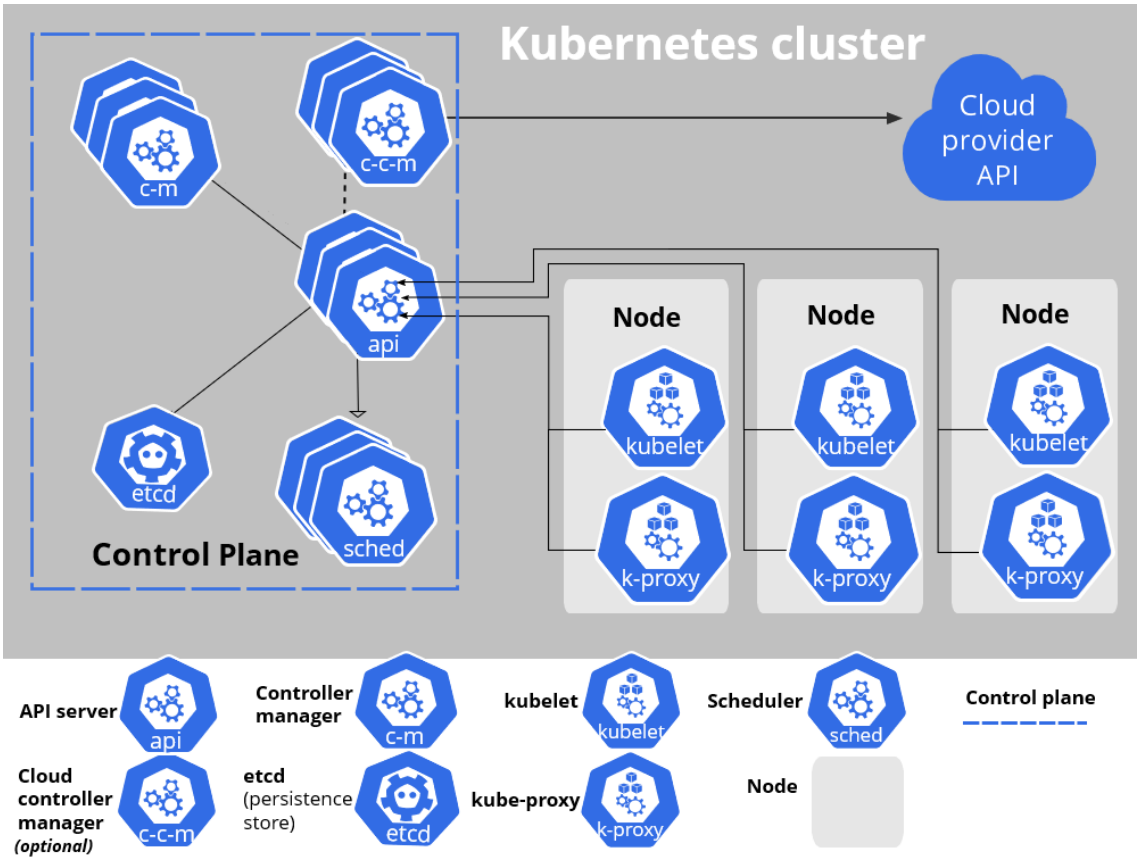


Figure 2-4 Kubernetes component diagram (The Linux Foundation, 2022c)

Other control plane components include etcd which stores the cluster data, scheduler which assigns pods to the nodes and controller manager which is responsible for running Kubernetes controllers (e.g., deployments and services) and node management. Cloud Controller Manager (CCM) is one of the optional Kubernetes components, but it is a vital element of the managed Kubernetes offerings in the cloud. For

example, Elastic Kubernetes Service (EKS), a managed Kubernetes offering by AWS, is built on top of AWS EC2. However, many of the AWS objects comprising the EKS cluster e.g., load balancers are not directly exposed to the user. Instead, they are leveraged by the Kubernetes controllers which are managed by the CCM. This way, when the user creates a new Kubernetes service, Kubernetes makes use of the AWS load balancer controller to create load balancer transparently to the user, and then automatically delete it once the Kubernetes service is destroyed (Amazon Web Services, 2022c).

2.2.2 Kubernetes Research

Kubernetes being the de-facto standard and the most feature-complete tool for container orchestration has caught the attention of both industry and academia in the past few years (Al Jawarneh et al., 2019). According to 2021 CNCF Annual Survey 96% of participating organisations globally are either using or evaluating Kubernetes (Cloud Native Computing Foundation, 2021). Much of the literature is dedicated to tweaking parameters to improve performance and availability and comparing different container platforms. For example, Vayghan et al. (2019) compared the level of service availability of default Kubernetes configuration to other availability management solutions and showed how to improve it by tweaking the redundancy models. Poetra et al. (2020) compared the container start-up times between Docker and Kubernetes and found that Kubernetes is 26 times faster with average container creation time of 7.7 seconds as opposed to 198.7 seconds average container creation time for Docker.

Another important research direction is running Kubernetes in the cloud. Cloud computing offers great benefits such as on demand rapid provisioning of managed resources. However, to achieve that level of service the hardware level of the infrastructure in cloud computing is abstracted away, so the end users are provided compute resources in the form of virtual machines. This means that in the cloud Kubernetes will be run on top of VM-based infrastructure, thus losing some of its performance advantages as compared to bare-metal Kubernetes deployments (Jindal et al., 2017).

The CERN paper by Imran et al. (2021) describes the migration of their data analytics workloads from VM-based to Kubernetes based infrastructure in the OpenStack private cloud. As they compare the performance of VM-based and Kubernetes-based cluster, they also mention that their Kubernetes cluster has extra resources as compared to VM cluster due to the overhead caused by Kubernetes system services such as ingress controller, metrics server, etc. Indeed, if we can see that their Kubernetes Production cluster has 288 CPU cores which is 12.5% more than the VM Production cluster with only 256 cores. If we

keep that in mind it is not so surprising to see that their “New K8s Cluster” performs better than the “VM Production Cluster” across all services in the performance benchmark demonstrated in Figure 2-5. However, as will be discussed in chapter 7, the results of my experiment show a slightly different picture.

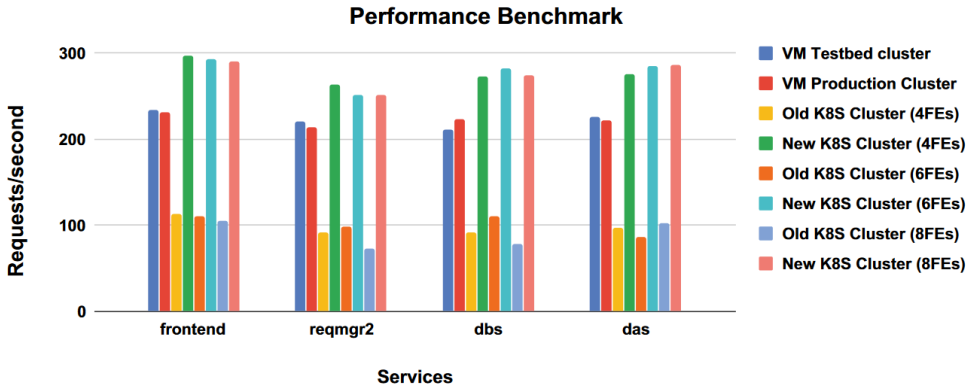


Fig. 10 The performance benchmark results in the form of requests/s (y-axis) of some of the commonly used CMSWEB services (x-axis) for various clusters (i.e. VM production cluster, VM testbed cluster, old Kubernetes cluster, and new Kubernetes cluster using a different number of replicas). A total of 100 tests were performed for each configuration and results show average values from these tests

Figure 2-5 Performance benchmark of the whole cluster (Imran et al., 2021)

When the performance benchmark conducted with the same number of nodes is considered (Figure 2-6[2]) most of the services after migration to the Kubernetes cluster show significant increases in performance as compared to the Production VM cluster, with crabcache service showing a more than 4-fold increase. However, some services like reqmgr2 show reductions in performance. Imran et al. (2021) explain the worse performance of some of the services on the Kubernetes clusters as being due to the “old architecture” which was addressed by means of “migration from Python2 to Python3, re-writing some services from Python in Go language to improve their scalability” (Imran et al., 2021, p. 9).

[2] Here UKC refers to the upstream-keepalive-connections parameter in the Nginx configuration (Imran et al., 2021, p. 11).

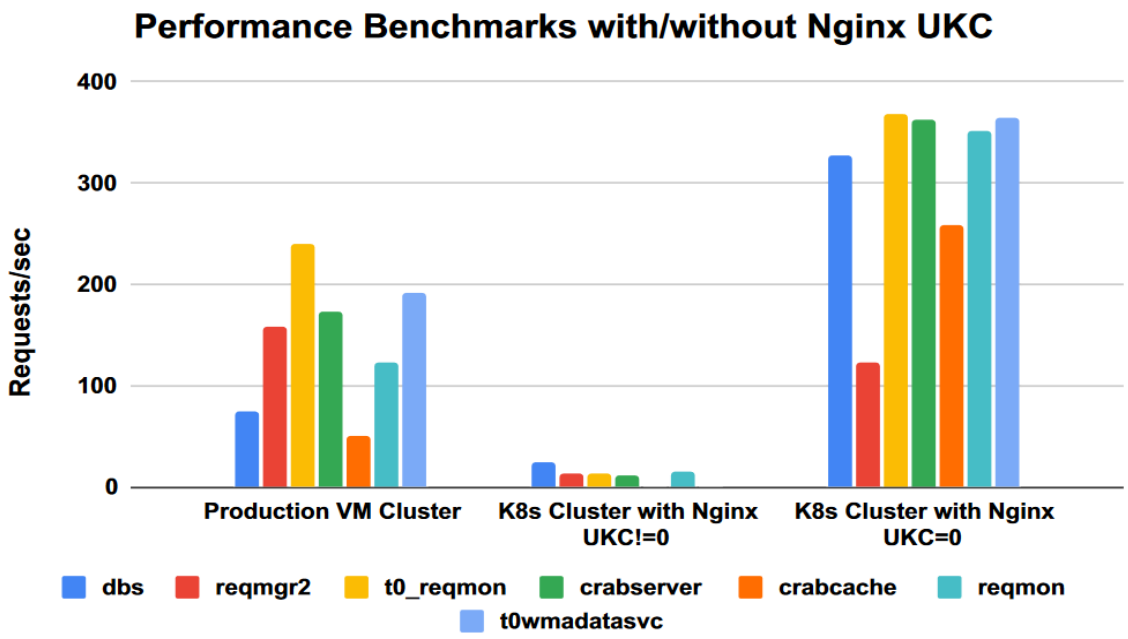


Figure 2-6 Performance benchmark[3] of the equal number of nodes (Imran et al., 2021)

This variability of performance profiles of different applications is also something that became apparent in the course of the this research and as pointed out by Imran et al. (2021) the peculiarities of the application architecture seem to be a possible explanation for that.

As they discuss the benefits of migrating to Kubernetes infrastructure Imran et al. (2021) state that besides performance gains the move to Kubernetes "significantly reduces the release upgrade cycle, unifies deployment procedures, and reduces operational cost"(Imran et al., 2021, p. 13). These non-functional improvements which facilitate faster development and, ultimately, better applications and lower operational cost are also a big reason for the popularity of Kubernetes in the cloud. That is why many enterprises consider the move to Kubernetes even though the performance gains cannot be always guaranteed (Burns et al., 2019; Poniszewska-Marańda & Czechowska, 2021).

2.3 CONTAINERS IN AWS

2.3.1 Amazon Elastic Container Service

All considerations discussed in the previous section equally apply to other cloud providers such as AWS. AWS had been chosen for this research as it is currently the leading cloud provider with the biggest

[3] Here UKC refers to the upstream-keepalive-connections parameter in the Nginx configuration (Imran et al., 2021, p. 11).

market share (Gupta et al., 2021). There are multiple ways of running and orchestrating containerised workloads in AWS cloud. Each way offers different functionality and trade-offs between the level control and the level of managed service. At the highest control level and the lowest level of managed service is the option of running Elastic Container Service (ECS) directly on EC2 instances. Here the EC2 layer is directly accessible to the end user which allows for fine-grained control over the ECS cluster properties and, therefore, allows more customised networking and scaling setup. Using ECS with Fargate on the other hand frees the user from cluster management tasks and allows to concentrate on application development (Amazon Web Services, 2022g). However, this comes at a cost of a more generic infrastructure setup which may not be optimal for some applications (Yussupov et al., 2021).

2.3.2 Kubernetes in AWS

Both ECS options discussed above are only available on AWS cloud which has disadvantage of vendor lock-in. Vendor lock-in may prevent the customers from moving to a cloud provider with a better value proposition and enjoying the benefits of multi-cloud deployments (Miller et al., 2021). Conversely, with AWS EKS customers can enjoy the benefits of the leading container orchestration platform while retaining the possibility of moving their workloads to a different cloud provider. With Amazon EKS Anywhere it is even possible use EKS control plane to manage hybrid clusters where part of the nodes may be running on-prem infrastructure or even in other clouds (Amazon Web Services, 2022b). Of course, there are options to run a completely self-managed Kubernetes cluster in AWS by installing and configuring nodes manually (DIY Kubernetes) or use a 3-rd party tool like kOps. However, these options have much bigger management overhead and require deeper knowledge of Kubernetes and AWS to setup (Poniszewska-Marańda & Czechowska, 2021).

According to Poniszewska-Marańda and Czechowska (2021) the benefits of using EKS over self-managed Kubernetes deployments include out-of-the-box integration with AWS services such as security (IAM), networking (VPC) and load balancing as well as a fully managed Kubernetes control plane and worker node groups. This means the user does not have to worry about installing and configuring Kubernetes services or installing OS security updates on the nodes. Moreover, AWS offers EKS infrastructure management tools such as eksctl which simplify the creation of the cluster. Among disadvantages of EKS as compared to Kubernetes DIY and kOps options is higher cost and fewer customization options (Poniszewska-Marańda & Czechowska, 2021).

The main reasons why EKS has been chosen as the way to deploy Kubernetes on AWS in this project is because firstly as a managed service it allows to save time which is a limited resource in this project. Secondly, EKS is recommended by AWS, and therefore, will be more relevant for the real-world applications.

2.4 SUMMARY

This chapter gave an overview of virtualization and containerisation technology and introduced the important concepts. Then existing research regarding running containers in the cloud was discussed. The chapter concludes with the overview of containerisation services in AWS cloud and the reasoning for choosing the EKS platform as Kubernetes deployment option.

CHAPTER 3

METHODOLOGY

This research uses an experimental approach to answer the research question because it is important not only to find relationships between variables (e.g., between application performance and infrastructure platform) but also try to find the explanation of those relationships. This chapter will discuss the experimental setup, define dependent, independent and control variables and outline experiment procedure.

3.1 EXPERIMENT SETUP

The experiment was conducted on an application deployment via the testing tool running on the jump host (Figure 3-1). The testing tool was used to generate load (a series of HTTP queries) on the application deployment. The performance and infrastructure metrics were then collected and used to assess the performance of the infrastructure platform.

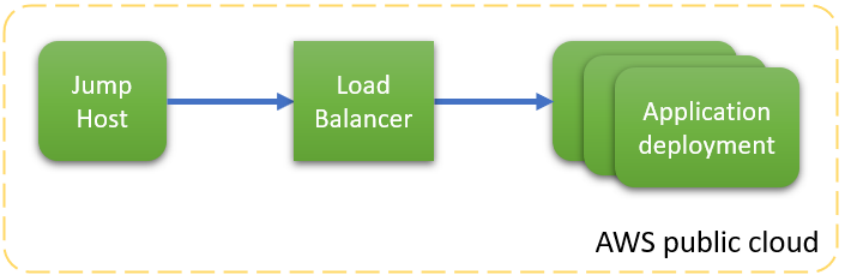


Figure 3-1 Experiment setup

Following the approach of Imran et al. (2021) two benchmarks have been used to assess the performance of the infrastructure: request throughput (how many requests per second can the infrastructure deliver) and scaling duration (how fast can a service scale to maximum number of application instances). These benchmarks correspond to two tests: the scaling test and the performance test. In the scaling test, the application deployment starts with a single application replica, then the time it takes for the deployment to scale to maximum replicas under load is measured. In the performance test, the application deployment is first scaled to maximum, then the average request throughput with maximum load over 5 minutes is measured. In each experiment run both tests are executed 3 times to improve the accuracy of the results.

Each stage of the experiment from infrastructure and application deployment to collecting the data is fully automated. Automation minimizes the risk of human error as well as chances of results being affected by infrastructure configuration drift. Furthermore, since every step of the experiment is cast in code it is easy to audit and reproduce. The complete infrastructure and testing automation is also necessary because the AWS sandbox environment used to carry out the experiment has a time limit of 4 hours, after which the temporary sandbox account is deleted. Therefore, the infrastructure and tooling setup had to be streamlined to be easily re-created on each experiment run.

Figure 3-2 shows a plan of the experiment stages developed during this project (covered in more detail in section 4.2). Experiment procedure starts with deployment of the infrastructure including the application instances as well as the jump host and required tooling. Then the infrastructure is adjusted according to the test specifications and warmed-up in preparation for the performance test. Next three performance test runs are executed followed by three scaling test runs. Finally, CloudWatch metrics are collected over the duration of the warm-up and testing stages and all data is uploaded to the permanent storage.

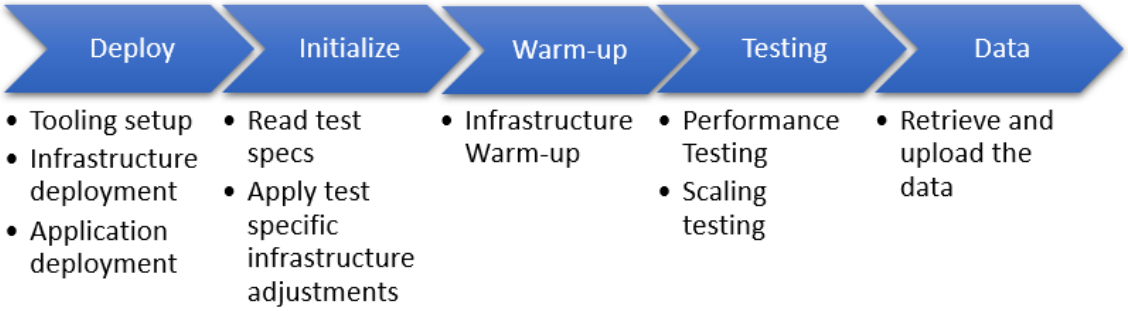


Figure 3-2 Experiment automation stages

Simple test web applications were developed to simulate typical real-world workloads. For a detailed description of test applications please refer to section 4.7. The experiment will help to establish which platform offers the best performance for each type of application and assist in understanding the effects of platform and application parameters on the performance.

3.2 EXPERIMENTAL APPROACH

The essence of the experimental method is to manipulate the independent variables in a controlled manner and measure the change in dependent variables to establish relationship between them (Jackson, 2011). Proper controls of the experimental setup and measurements are important because they help to

confirm cause-and-effect relationships between the dependent and independent variables as well as to factor out alternative explanations (T. P. Ryan & Morgan, 2007).

3.3 DEPENDENT VARIABLES

This research addresses the question of how ASG and EKS platforms compare in terms of performance and scaling. The measurable quantities that represent performance and scaling are the queries per second (qps) and the scaling duration respectively. These are naturally the main dependent variables in this research. There are other dependent variables such as Number of errors and CPU utilization which are not direct indicators of performance but are useful for explaining some of the observations. For the full list of the dependent variables and their functions please refer to Table 3-1.

Variable name		Variable function	Associated metrics
Raw qps	**Raw qps**	A measure of performance capability in queries per second.	qps (Fortio)
	Relative raw qps	Raw qps relative to ASG platform for easier comparison between the platforms.	
Weighted qps	**Weighted qps**	A measure of efficiency	qps (Fortio), cpuUtilization (CloudWatch)
	Relative weighted qps	Weighted qps relative to ASG platform for easier comparison between the platforms.	
CPU utilization		A measure of CPU utilization.	cpuUtilization (CloudWatch)
Scaling duration		A measure of scaling capability.	StartTime (Fortio), groupInServiceCapacity (CloudWatch), readyReplicas (Kubernetes)
Errors		Big number of errors may indicate issues with scaling, infrastructure, or applications.	backendConnectionErrors (CloudWatch)

Table 3-1 Dependent Variables

The relative versions of qps variables have been introduced because of big range of absolute values (1000 to 50000). Not only it is natural to do the comparison in relative terms, but it also makes visual presentation more convenient when the results of multiple applications are displayed on the same chart. Moreover, the absolute values of qps are not compared across the applications since the purpose of the experiment is comparing relative performance of the infrastructure platforms. Therefore, comparing absolute qps values across applications is out of the scope of this project (and was not controlled for).

Raw qps metric works well for measuring performance of the multi-threaded applications (e.g., Apache) that can fully utilize the CPU of the dual-core instances used in this research. However, single-threaded applications like Taewa can utilize only one core, and therefore only 50% of the total instance capacity. This is a problem because ASG platform does not have a built-in mechanism for running multiple instances of an application similar to EKS's pods. Therefore, to make the comparison more meaningful, a weighted qps metric have been introduced. Weighted qps is calculated by dividing the qps by the CPU utilization, therefore it shows us how efficient each platform is at converting CPU usage to useful work (qps). It also gives an indication of cost efficiency if the workloads are run on burstable EC2 instances with limited CPU credits like T2 and T3. The use weighted qps also allows to minimize the impact of CPU steal (see 6.2.1) on the results, since lower raw qps performance due to the stolen CPU will be balanced out by the lower CPU utilization caused by the stolen CPU.

3.4 INDEPENDENT VARIABLES

The factors that are manipulated in an experiment are called the independent variables. This research focuses on comparing the infrastructure platforms and how the choice of the platform affects the performance of the application. Therefore, one group of independent variables is infrastructure related, this includes the types of platforms (ASG or EKS) and platform parameters such as pod density and CPU target. The other group is application related and includes applications themselves and application properties such as difficulty and concurrency. The test applications are discussed in detail in section 4.7. For the full list of the independent variables and their functions please refer to Table 3-2.

Variable type	Variable name	Tested variations	Variable function
Application	**Application**	Apache, Taewa, Riwai, Raupi	To assess how choice of application affects relative platform performance
	Difficulty [only Taewa]	1, 20000, 80000	To test if CPU bound (high difficulty) and network bound (low difficulty) applications show different relative platform performance
	Web framework (library) [only Node.js]	Express, http	To check how choice of web framework affects relative platform performance. Results allow to make more general conclusions (not just about an application but about programming language in general)
	Concurrency	Single-threaded (Taewa, Riwai), Multi-threaded (Apache, Raupi),	To check if there is a difference between single-threaded and multi-threaded application relative platform performance
	Programming language	Node.js (Tewa, Riwai), Python (Raupi), C (Apache)	
Infrastructure	**Infrastructure Platform**	ASG, EKS	To assess which platform offers a better performance for a given application
	Pod Density [only EKS]	Normal (1 pod per core, 2 pods per node), Low (1 pod per 2 cores, 1 pod per node)	To gauge the effect of pod density on performance of a given application. Results also allow to estimate the impact of Kubernetes overhead on performance.
	Number of nodes [only Taewa]	3, 6	To assess how well each platform scales horizontally.

Table 3-2 Independent Variables

3.5 CONTROL VARIABLES

To ensure that the change in the dependent variables is only caused by the change in the independent variables several control variables were required. Control variables are not in the focus of the study, but they need to be kept constant (controlled) because they may affect the results.

The most obvious group of control variables is infrastructure related. Since two different infrastructure platforms are compared there are inevitable differences in the infrastructure setup

between ASG and EKS deployments. However, the effect of those differences was kept to a minimum by ensuring that:

- The type of AWS load-balancers as well as the number and type of EC2 instances are the same between ASG and EKS deployments (see section 4.5)
- Both platforms use the same OS on the worker nodes platforms (see section 4.5.4)
- Both platforms use the same networking and availability zone setup (see section 4.5)
- Package versions of Apache, Node.js and Python are the same across platforms (see section 4.5.4)

Moreover, the versions of tools that are installed on each experiment run like eksctl, Fortio and Kubernetes were "pinned". "Pinning" means that instead of installing the latest version of the software which could change during the experiment, the version that was latest at the start of the experiment had been selected and then set explicitly in the tooling setup scripts. The versions of the 3rd party libraries used in the test applications have been controlled via language specific tools. For Node.js applications it is package-lock.json and requirements.txt for Python (see section 4.7 for more details).

Another group of control variables is related to the experiment procedure itself. As the testing procedure contains many steps and takes over two hours to complete it was important to ensure that it had been run consistently and without any deviations in the timing or order. This was ensured by fully automating the testing procedure as described in section 4.8. Additionally, the load testing script is structured in such a way that the parts responsible for testing are re-used across both infrastructure platforms, the only differing parts are related to the infrastructure configuration. Furthermore, the load script development was frozen after the development deadline date (13 March 2022), that means no changes to the code of the testing procedure[4] were done after that. Therefore, to ensure the consistency of the results all of the test data considered in the comparison is collected after that date. The control variables will be covered in more detail in the Chapter 4.

[4] However, after that date changes still have been made to the parts of the code which do not affect the experiment procedure, e.g., changes in the code for the collection and processing of the data or adding new test specifications

3.6 SUMMARY

This chapter outlined the experimental setup and described the dependent and independent variables as well as the controls that were implemented to ensure the accuracy and consistency of the results. The next chapter will cover the specifics of the experiment implementation and explain the design choices.

CHAPTER 4

DESIGN AND IMPLEMENTATION

4.1 INTRODUCTION

This project started as an effort to compare the performance of infrastructure platforms, so initially most of the efforts were focused on developing the infrastructure and deployment aspects, assuming that testing is just an auxiliary aspect that could be done manually. However, once the prototypes of the infrastructure and application were ready and the first tests had been run, it became clear that it would not be feasible to complete the amount of testing and data collection planned without automating the testing process. So, as the project progressed, testing automation (load script) became the central and most time-consuming part of the development. This code-driven approach to testing resulted in a robust and flexible fully automated testing solution that allowed to run the experiments at a faster pace while mitigating the uncertainties that would be inevitable if the tests were carried out manually by the human operator. Other benefits of the resulting solution are that every test is easily reproducible because the code for infrastructure, deployment, testing, and data processing is stored in a single repository that is version-controlled and can be easily audited.

This chapter gives an account of the testing solution design, architecture, and components as well as the test applications. It starts with an overview of the testing solution, then coves the specifics of ASG and EKS infrastructure implementation. Then the infrastructure deployment automation is discussed as well as the test applications and their build process. Finally, all stages of the load testing script are described in detail.

4.2 SOLUTION OVERVIEW

The solution automates all stages of the experiment – from the initial infrastructure setup to the presentation of the aggregated results and graphs. Figure 4-1 gives a high-level overview of each stage.

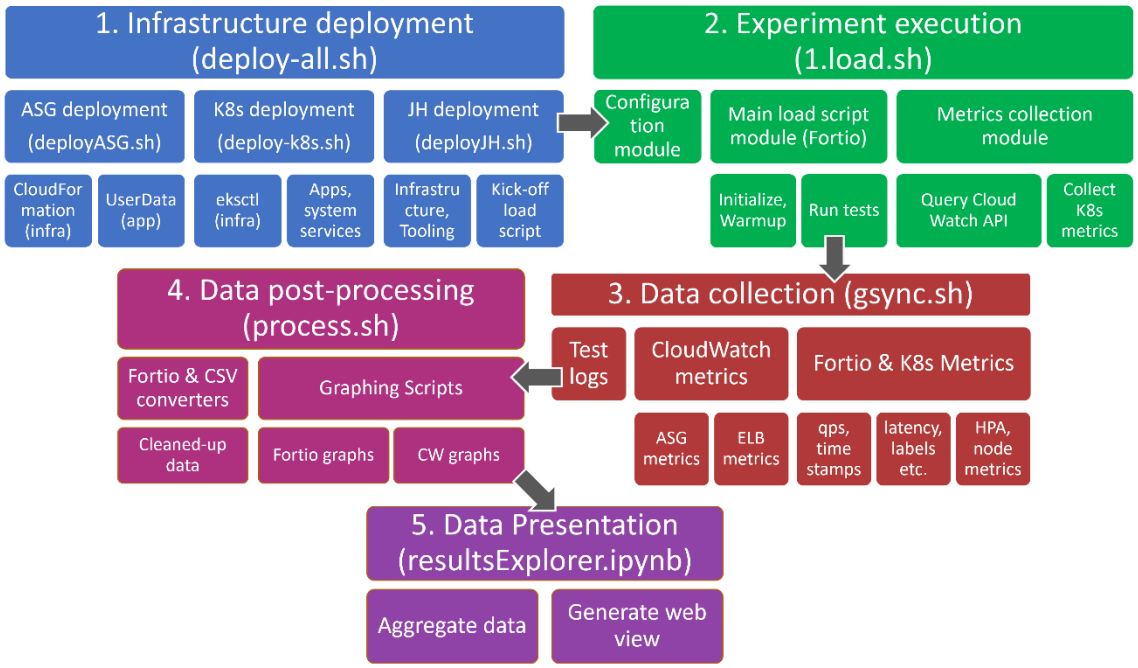


Figure 4-1 Solution overview diagram[5]

Since each experiment starts with a blank AWS sandbox account the function of the infrastructure deployment stage (Figure 4-1) is to create the infrastructure, deploy the application and create the jump host where the testing procedure will be executed. This stage is initiated via the thesis-infra\deploy-all.sh[6] script. Deploy-all.sh script only requires AWS credentials and the name of the test specification as inputs since the name of the test specification already contains all required information including the type of the infrastructure to be deployed, the number of nodes and the application name (section 4.6.1 covers in more detail how the specification names are formed).

Once the application deployment and the jump host initialization are complete the kick-off of the load script marks the beginning of the experiment execution stage. Here, the load script (thesis-infra\data\cli\1.load.sh) executes the experiment procedure from the warmup to the collection of the result data. This stage is completely defined in code and does not need any additional input. The first two stages – preparation of the infrastructure deployment and experiment execution are tightly coupled. They

[5] The Figure 4-1 only covers the infrastructure, experiment and data processing parts of the solution, for information on application build and packaging automation please refer to section 4.7.4.

[6] In this and later chapters all file names are referenced by their full paths including the name of the GitHub repository where they are located. If the file path starts with "thesis-infra" it is located in the infrastructure repository, "thesis-apps" refers to the application repository and "thesis-results" refers to the results repository. Appendix B contains the URLs of the repositories where they can be explored on-line.

are always executed together and take from 2 to 2.5 hours to complete. The resulting raw data is then saved on the intermediate SSH server as the sandbox account is deleted after 4 hours.

In stage three (data collection) the raw data is downloaded into the results repository via the thesis-infra\etc\gsync.sh script. Each experiment run is represented by a folder in the results repository with the name containing the date and the name of the test. Each results folder contains the test log as well as the CloudWatch, Kubernetes and Fortio metrics.

At this point the result data is in a machine-readable format and is not ready for human interpretation. Therefore, the purpose of stage four (data-post-processing) is to consolidate and clean up the raw data as well as to plot the relevant metrics. This is done via a suite of Python and jq scripts (see section 4.3 for overview of the jq tool) which produce the CSV files and the graph images. The resulting processed data is enough to check the results of a particular experiment run e.g., to see if there are any anomalies and the experiment must be re-run. However, at this stage the data processing scope is still limited to a single experiment run and does not provide aggregated metrics.

Stage five (data presentation) uses the data produced in the previous step to calculate more complex metrics (like scaling duration) and aggregates metrics according to the test specification. In the final step it generates the Results Explorer - an interactive web view containing the graphs and aggregated metrics. Stages 4 and 5 are usually executed together via the process.sh script to update the Results Explorer view after the new experiment data has been downloaded.

4.3 TOOLS AND TECHNOLOGIES

This section will cover the tool and technologies used in this project.

AWS sandbox

Two options of AWS sandboxes have been tested – one provided by AWS Academy (www.awsacademy.com), the other provided by A Cloud Guru (ACG) (www.acloudguru.com). The AWS Academy sandbox not only has limitations on the budget, but also on the security permissions and types of AWS services allowed. ACG sandbox on the other hand has no limitations on the budget or AWS services but only persists for 4 hours, after which all resources created during the session are destroyed and the temporary account is deleted.

Manual (via web console) EKS deployments in AWS academy sandbox were successful. However, automated (eksctl) deployment failed with error indicating the permission limitations of the AWS

academy sandbox while eksctl EKS deployment in ACG sandbox was successful. Since eksctl is the preferred method of deploying EKS (see 4.5.2) ACG sand box has been selected for this experiment.

Infrastructure as code

Infrastructure as Code (IaC) is the practice of automating the infrastructure deployment via scripts that describe the desired state of infrastructure. These scripts commonly use data serialization languages like YAML and JSON, but can be written in any programming language (Java, Python, etc) through the use of appropriate libraries (Artac et al., 2017). IaC approach has been used to facilitate fast and consistent infrastructure deployment in each experiment run. Both AWS CloudFormation templates and Kubernetes manifest files used in this project are examples of IaC approach to infrastructure development.

AWS CloudFormation

AWS CloudFormation is an Infrastructure as Code tool that allows to deploy collections of AWS resources called stacks. Cloud Formation stacks are built from the YAML templates with declarative style language. All AWS resources used in this project including the ASG and EKS infrastructure as well as the jump host are built via CloudFormation.

Infrastructure management utilities

Kubectl, eksctl and AWS CLI are the command line utilities for managing Kubernetes resources, EKS deployments and AWS infrastructure respectively. They have been extensively used in both infrastructure deployment and load testing scripts, as well for troubleshooting purposes. As these are the standard command line tools of respective technologies, they are well supported and have good documentation.

Docker and Docker hub

Docker is a standard tool for building and running containers. Docker is used in this project to build the test application images for running on Kubernetes. Docker Hub is a public container image repository, where the images are published to be later retrieved by the Kubernetes nodes during the EKS infrastructure deployment.

Fortio

Fortio is a load testing tool used in this project to measure the performance of a given test setup. In the initial stages of the project the hey tool used by Imran et al. (2021) was tested as well. However, it was found that hey tool does not report the status code distribution correctly for runs with over 1 000 000 total requests (it only considers first 1 000 000 requests), while in this project some test runs contain

close to 20 million requests (e.g., asg_apache_3 performance runs). Moreover, since Fortio was also used in other container performance research papers (Murtaza et al., 2020; Wang & Cai, 2022), it has been chosen for this project instead of the hey tool.

JQ utility

A lot of modern command line utilities either use JSON as the default format or have the option to output in JSON. Although CLI utilities often have built-in JSON filtering functionality, it is usually limited and the syntax is not consistent between different programs, so it requires additional time to learn. Therefore, a standardized way of filtering and processing JSON became essential for this project. JQ is a flexible command-line utility that filled that need. In this project JQ has been used for processing Fortio and Cloud watch metrics, filtering AWS and Kubernetes command-line outputs for monitoring as well as retrieving the values needed for test and infrastructure automation.

Shell scripts

Shell scripts are a common way of automating running command-line commands and OS administration tasks in Unix-like systems. In this project shell scripts are used to automate deployment and testing procedures as well as for batch processing. As infrastructure deployment and batch processing tasks are executed on a Windows laptop, a Windows binary of bash from gitforwindows.org had been installed to run the shell scripts. Using shell scripts instead of windows native interpreters like cmd and PowerShell allows the code to be more portable, so that it can be used across Windows, MacOS and Linux with no modifications. It also allows to easily re-use the code or move functionality between automation modules if needed. Shell scripts are also used to write the AWS EC2 UserData scripts which help to automate the EC2 instance start-up tasks (Amazon Web Services, 2021f).

Python and Node.js

Python and Node.js are the programming languages used for developing the web applications. They have been selected because they are among the most popular programming languages (Statista, 2021; UC Berkeley, 2022) and I already had some previous experience with them. Besides being used for development of the web applications Python is also used for data processing and plotting the graphs.

Git and GitHub

Git is a version-control software used to track the changes in the code. Since all code (infrastructure, deployment and application) has been version controlled from the beginning, it is easy to follow the

evolution of the code and even reproduce intermediate results if needed. GitHub is an online hosting for Git repositories. It is used for storing and sharing the code of this project as well as the results. The GitHub Pages functionality is used for publishing the online version of the Results Explorer. Links to all GitHub repositories containing the code and results of this project can be found in Appendix B.

4.4 INFRASTRUCTURE BUILD

As discussed earlier in this chapter, all infrastructure for the experiment is built via CloudFormation and eksctl inside a temporary sandbox account. There are three main infrastructure components – EKS, ASG and jump host. Each component contains at least one CloudFormation stack. Jump host component contains one stack which deploys the jump host, ASG platform component also contains one (more complex) stack which deploys ASG infrastructure while EKS platform component contains 2 stacks – one for EKS control plane and the other for the EKS node group.

Since EKS stacks are generated by eksctl (see section 4.6.2) the developer has little control over exact infrastructure being created. Therefore, the approach taken in this project is to mirror the EKS infrastructure setup in ASG CloudFormation code as much as possible. However, instead of trying to create a perfect reproduction of EKS infrastructure in ASG, a more pragmatic approach has been taken where industry standard practices were followed for each infrastructure platform. For example, instead of using an identical binary of Apache in both ASG and Kubernetes implementations a pre-packaged RPM package of Apache shipped with OS has been used in ASG implementation and an official Apache docker image had been used in EKS implementation. This way the results of this project can be more relevant for real-life applications.

However, when not contradicting the above principle, measures have been taken to ensure a fair and consistent comparison, e.g., software versions used were as close as possible between the platforms (see the “Packages” section for software versions).

4.5 SOLUTION ARCHITECTURE

From the architecture perspective both EKS and ASG deployments look similar. As shown on Figure 4-2, the application instances (ASG instances and EKS nodes) are distributed across availability zones (AZ) A and B, while the jump host instance is placed in the AZ C. In both cases the application instances are fronted by the load balancers. One of the differences is that in EKS deployments the application instances are put in a separate Cluster VPC. This does not affect the results because VPC is only a logical network

separation while the jump host and application instances are already physically separated into different AZs. The EKS control plane is not displayed on the diagram because it is fully managed by AWS and its components are not accessible to the end user directly.

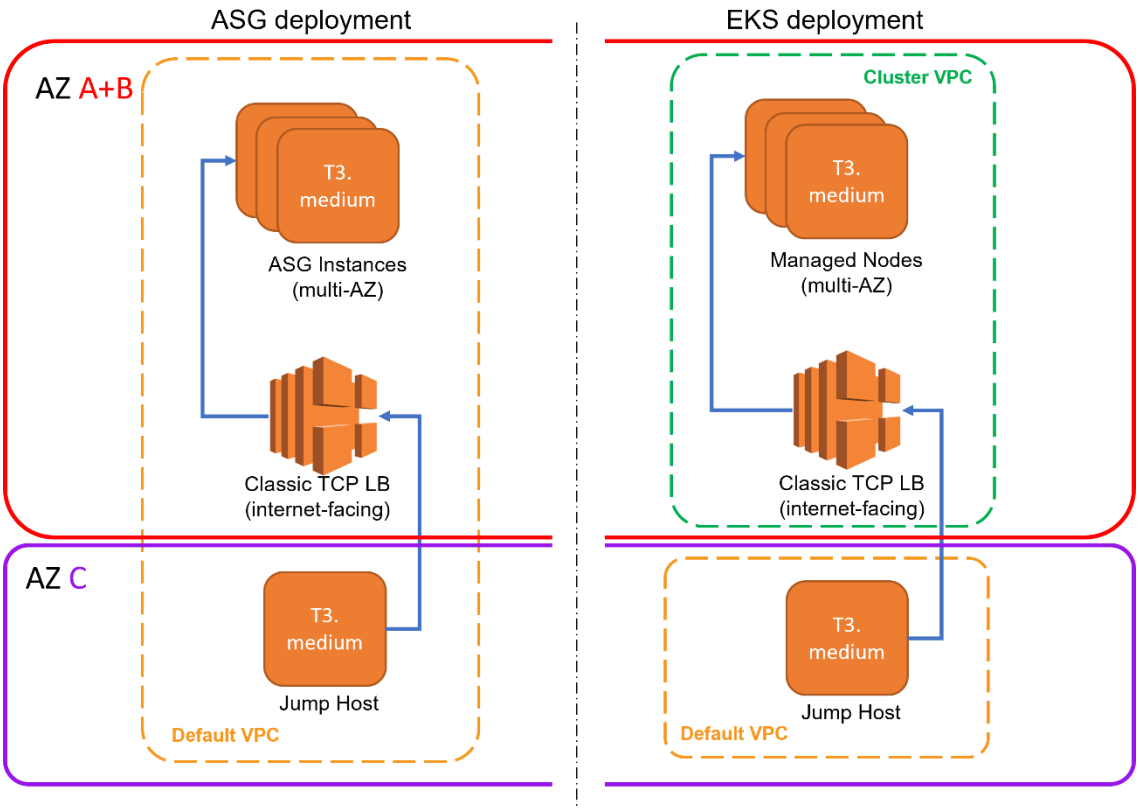


Figure 4-2 Solution architecture diagram.

The blue arrows on Figure 4-2 indicate the path of the requests which are sent by the jump host to the load balancer and then forwarded to the application instances.

4.5.1 ASG deployment

There are several options for deploying ASG infrastructure: manual (using AWS EC2 web-console), semi-automated (using AWS CloudFormation web-console) and fully automated (using AWS CLI). The first two options involving web-console require manual input and cannot be fully automated. Therefore, the fully automated CloudFormation option has been chosen to ensure consistency of the results and to streamline the infrastructure deployment procedure.

A standard pattern of ASG coupled with Load Balancer has been used (see Figure 4-2). Although originally an Application Load Balancer has been used it had been later changed to Classic LB. The main

reason for this change was because Classic LB it is the only option in the EKS platform[7] and it didn't affect the performance. Furthermore, Application LB and Classic LB have different sets of available metrics (Amazon Web Services, 2022d, 2022e). Therefore, to ensure consistent comparison between platforms it was decided to use the Classic LB on both platforms.

Several design approaches were found to be useful when developing CloudFormation templates. Passing secrets such as SSH and email credentials as template parameters instead of hard coding allowed to avoid storing secrets in the version control system (Git) and therefore avoid potential security risks. Passing stack names as parameters of the deployment scripts prevented stack name collisions which allowed to simultaneously run multiple CloudFormation stacks from the same template. However, when running multiple copies of the stack resource name collisions become a problem. Appending stack name to child resource names prevents that, allowing to run multiple instances of the stack and speeding up the development of CloudFormation code.

4.5.2 EKS deployment

For the same reasons as described in the previous section, the manual web-console driven options were not considered for EKS deployment. Instead, several automated EKS deployment options have been tested. The first option considered was a "QuickStart" suite of CloudFormation templates recommended by AWS (Amazon Web Services, 2020a). Upon closer examination it became clear that this solution was too complex for the requirements of this project as in "QuickStart" suite the simplicity of the code had been sacrificed to showcase all features of EKS and allow the user to make choices via GUI. This led to complicated, hard to analyse, code.

So, in search of a more concise CloudFormation code that could be easily adjusted for the purposes of the project the second attempt to create an EKS cluster via CloudFormation was made using the tutorial by Hammond (2019). In this case CloudFormation stack containing EKS Control Plane had been successfully created by following the instructions. However, creating the worker nodes stack especially setup of the corresponding network infrastructure, turned out to be difficult and after several unsuccessful attempts this option was abandoned.

[7] There is an option to use Application LB as Kubernetes ingress resource in EKS. However, the requests will still be mediated by the Classic LB as it represents the Kubernetes service resource (Amazon Web Services, 2021a).

The final attempted approach to EKS deployment was to create the EKS cluster via the eksctl utility. eksctl is a command line utility developed by AWS (Amazon Web Services, 2022h) which simplifies the creation and management of EKS clusters. It automatically generates and executes the CloudFormation code required to create the EKS control plane, worker nodes and network infrastructure as well as necessary security policies associated with them. Testing of this approach was successful, and it was therefore selected for the Kubernetes infrastructure deployment in this project.

As opposed to ASG deployment where low-level AWS resources such as EC2 instances and load balancers are directly specified in the CloudFormation template code, only high-level entities such as EKS clusters, node groups and Kubernetes services can be controlled by the user in the EKS deployment. The underlying CloudFormation stacks containing low-level AWS resources are created and modified automatically.

To get a more complete understanding of the EKS architecture the CloudFormation template code of the stacks generated by eksctl have been saved in the thesis-infra\etc\collateral\cf-generated-cluster.json (cluster template) and thesis-infra\etc\collateral\cf-generated-workers.json (workers template) files of the infrastructure repository. The analysis of the code revealed that all the networking infrastructure including the worker node VPC, subnets and security groups is contained within the cluster template file, while the worker nodes themselves as well as the associated launch template, IAM role and policy are contained within the workers template file.

Regarding the EKS instance sizing, one of the things that needed to be considered was the maximum number of pods that different types of AWS instances can support. According to AWS documentation, there is a limit to how many IP addresses can be assigned to an EC2 instance which in turn limits the number of Kubernetes pods that can be run on that instance. For T3.medium instances this maximum is 17 pods (Amazon Web Services, 2022j), this also includes system pods which brings the maximum number of application pods down to 12-14 which is enough as the maximum number of pods per node used in this research is 12.

4.5.3 Deployment summary

The table 4-1 below summarises the AWS infrastructure setup for both platforms.

Resource	ASG setup	EKS setup
Application instances	ASG composed of T3.medium EC2 instances	Managed EKS node group composed of T3.medium EC2 instances
Load balancer	Internet-facing classic TCP load balancer	Internet-facing classic TCP load balancer
Networking	Multi-AZ, single VPC	Multi-AZ, 2 VPCs
Jump host	T3.medium EC2 instance (shared code)	

Table 4-1 Infrastructure deployment summary

4.5.4 Software and OS controls

Besides achieving similarity on the hardware and networking level it is important that the OS and package versions are controlled as well. Since EKS managed nodes use the latest available EKS optimized Amazon Machine Images (AMI)(Amazon Web Services, 2022i), a decision had been made to use the latest image (as opposed to using a particular build of an AMI image) for ASG instances as well. This is in line with the industry practice of using up-to-date security-patched OS images. Since different tests could potentially have used different images, it might have had an impact on the results. However, the impact should be minimal as stable versions of the images have been used. Please see Appendix E for the list of the AMI versions used in this project.

The software package versions have also been controlled. If a platform has a pre-packaged version of the software, then it is preferred. Otherwise, the official binary is downloaded from the software website. Please refer to Appendix E for the list of the software package versions used in this project.

4.6 INFRASTRUCTURE AND APPLICATION DEPLOYMENT

The infrastructure deployment automation consists of the main deployment script and three deployment modules (Figure 4-3): the jump host deployment module, the EKS deployment module and ASG deployment module. This modular structure evolved gradually over the course of the project. At the start of the project the commands have been issued individually by copying from the text file. Later, the individual parts of deployment have been automated by creating respective deployment modules. Finally, the overarching script (the main deployment script) have been created to run the deployment modules in required sequence. All deployment modules are self-contained which allows them to be run individually if required. Besides, this modular architecture makes the resulting code clearer and allows for easier troubleshooting.

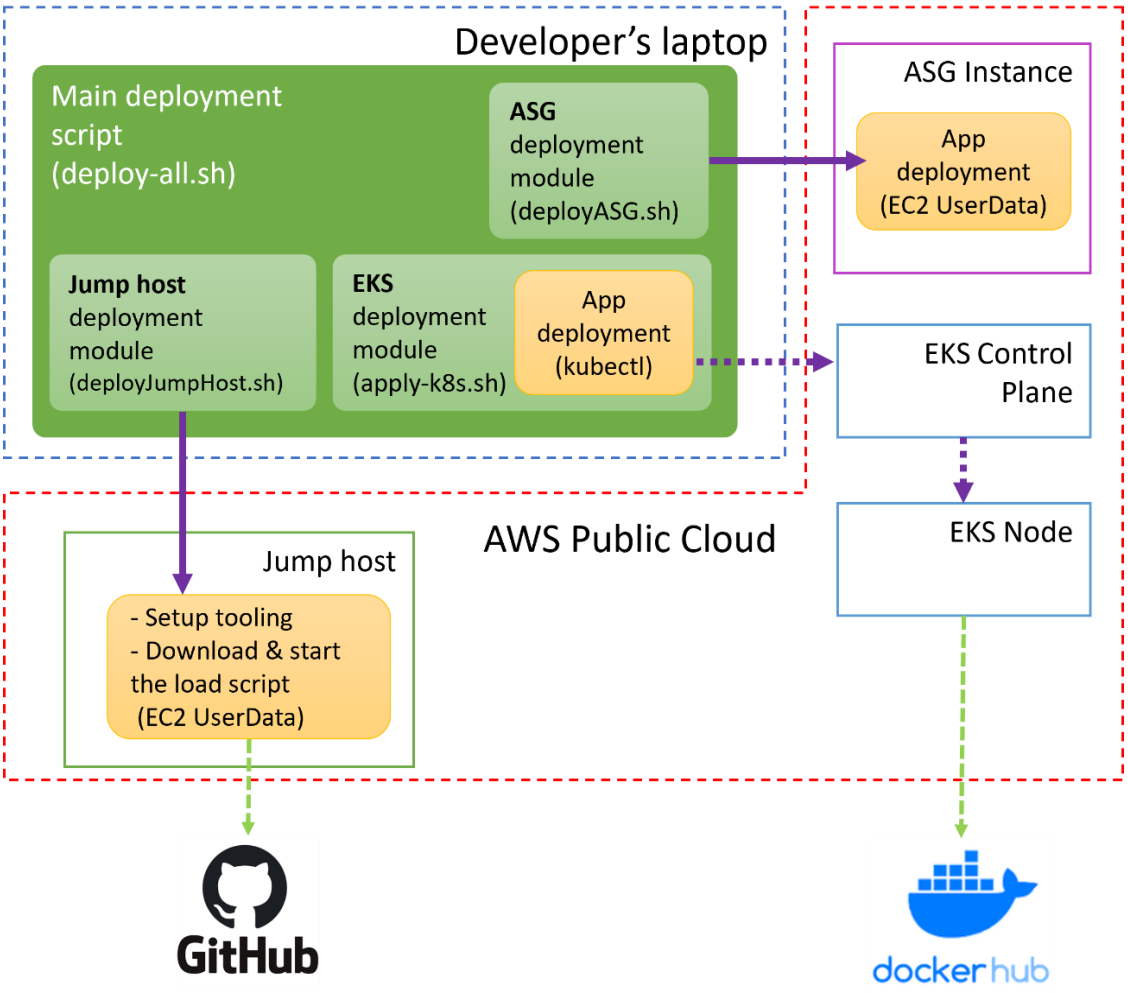


Figure 4-3 Infrastructure deployment automation diagram.

As shown on Figure 4-3 the infrastructure deployment is performed by the main deployment script (green) which is executed on the developer's laptop. First, the jump host deployment module is executed. After the jump host instance is created its UserData[8] initialisation script performs installation of testing and infrastructure tools, then it downloads the load testing script from the GitHub and starts it. Then, depending on the test specification either ASG or EKS infrastructure is created by the respective infrastructure deployment module. ASG deployment module uses CloudFormation to create the ASG instances, while the UserData scripts on ASG instances perform the application install. EKS deployment module, on the other hand, uses eksctl to create the EKS cluster and then initiates application deployment via kubectl. The application containers are then downloaded from the Docker Hub and executed on the EKS nodes.

[8] UserData is a script executed when an EC2 instance first starts. For Linux instances used in this research UserData scripts are written as Linux shell scripts (Amazon Web Services, 2021f).

4.6.1 Main deployment script

The main deployment script (thesis-infra\deploy-all.sh) is executed manually by the operator on the developer's laptop. It has only one parameter which is mandatory - the load test name which determines the deployment modules that will be run for a particular experiment run. The test name consists of three parts separated by underscores. The first part is the platform name, the second part is the name of the application, and the third part is the number of nodes. For example, passing k8s_apache_3 as test name will run the EKS deployment module with three nodes and deploy an Apache application. While passing asg_taewa_3 as test name will run the ASG deployment module with three instances and deploy the Taewa application.

The sequence of the main deployment script is as follows:

1. The SSH keypair that will be used in creating EC2 instances (jump host and/or ASG) is created and saved.

2. The jump host CloudFormation stack is deployed via the jump host deployment module

3. The ASG or EKS platform infrastructure is deployed (depends on the load test parameter).

The script needs to be authenticated with an IAM role with account administrator permissions (e.g. AWS managed "AdministratorAccess" policy) configured in the default profile as per AWS CLI user guide (Amazon Web Services, 2021c). Specifically, the default AWS CLI credentials file must be used as it is parsed by the jump host deployment module to retrieve the AWS credentials used for kubectl authentication (see 4.8.2.1).

4.6.2 EKS deployment module

The EKS Kubernetes cluster is created via eksctl command line utility which is reading the cluster configuration from the thesis-infra\eksctl\Cgenerated.yml file. The YAML file has been initially generated by the eksctl and later customised to fit the requirements of the project and specifies the configuration of the Kubernetes, AWS networking and worker nodes.

After the cluster deployment is complete the auxiliary Kubernetes services such as metrics server and cluster autoscaler are deployed by the thesis-infra\k8s\apply-k8s.sh script via https://github.com/kubernetes-sigs/metrics-server/releases/download/v0.6.1/components.yaml and thesis-infra\k8s\cluster-autoscaler-autodiscover.yaml Kubernetes manifest files respectively. Next, the

test applications are deployed based on the test name parameter that is passed to the EKS deployment module. The test applications are deployed via the manifest files from the thesis-infra\k8s directory. Each application has at least one manifest file with the name matching the application (e.g. apache.yaml, taewa.yaml, etc). The applications that have low density tests have additional manifest files accordingly (e.g. apache2.yaml, taewa2.yaml, etc).

At this point the service for the test application on EKS cluster is ready to accept requests, but the deployment is not yet auto scaled. The HorizontalPodAutoscaler controller for that deployment will be created as part of the load test script (see 4.8.2.3).

4.6.3 ASG infrastructure deployment module

The ASG infrastructure is deployed by the ASG deployment module (thesis-infra\aws-asg\deployASG.sh). The deployASG.sh script first detects the AWS account network configuration such as VPC and subnet IDs and then passes these values to the ASG stack creation CLI command. The ASG CloudFormation template (thesis-infra\aws-asg\asg-template-classicELB.yaml) which is used to create the ASG stack defines the autoscaling group with its initial scaling limits and scaling policy as well as the load balancer and security objects.

The deployment of the test application is performed via UserData scripts of the EC2 instances, which are executed during initialisation of each instance. Like in the EKS deployment, the name of the application to be installed is extracted from the test name parameter passed to the stack.

After the ASG stack is created the test application is ready to accept requests, however, the scaling policy including the maximum number of instances may be adjusted later in the load test script if the scaling policy in the test specification is different from the one specified in the ASG CloudFormation template (see 4.8.2.3).

4.6.4 Jump host deployment module

Similarly, to the ASG infrastructure, jump host is deployed via CloudFormation with dynamic parameters detected by the deployment shell script and passed to the CloudFormation stack. The jump host CloudFormation template (thesis-infra\aws-tools\jumphost.yaml) is parametrised on network settings, AMI image ID and the name of the load test. Other template parameters include AWS credentials for authenticating eksctl (see 4.8.2.1), SSH server address and credentials for storing the results of the test and MailJet credentials used to send the test completion notification.

The UserData script of the jump host instance is used to install the tools used by the load script (kubectl, eksctl, Fortio, git) and to pass the SSH, AWS and MailJet credentials to the load script. The deployment phase of the jump host is finished with the kick-off of the load testing script. For more information on the load script please refer to the section 4.8.

4.6.5 Deployment automation script idempotence

A piece of automation is called idempotent if the final result doesn't depend on how many times it is run (Hummer et al., 2013). Idempotence is what most Infrastructure-as-Code tools are striving to achieve (Amazon Web Services, 2020b).

The scripts that are used to deploy the infrastructure for this project are idempotent. If there was an error during the deployment, the script can be simply re-run to fix the failed operations. This is important and convenient because the deployment of the cluster can take up to 30 minutes, so if only few operations fail, the Kubernetes cluster doesn't have to be re-deployed. Another benefit is that the user of the script doesn't have to track which component is currently in which state – the user simply needs to fix the error and run the script again. If there were any operations dependent on the failed module – they will be executed once the parent module is fixed. This comes from the idempotency of the tools used – AWS CloudFormation and Kubernetes as well as the choices in code of the script itself. E.g., in the thesis-infra\deploy-all.sh line 5, the private key for the SSH connection is only written to disk if the "aws ec2 create-key-pair" command is successful. This prevents overwriting of the key during the subsequent runs of the deployment script, when the key creation will fail due to the key object already existing in that namespace. The script can be still improved by handling the errors better e.g., if CloudFormation "create-stack" command fails due to stack already existing, it can run "update-stack" instead.

4.7 TEST APPLICATIONS

The purpose of the test applications is to emulate different types of real-life workloads. Multiple applications have been used as simulated workload in the infrastructure platform performance comparison including Apache web server, web applications written in Node.js (Taro, Taewa. Riwai) and one application written in Python (Raupi). The code for all developed apps can be found in the application code git repository (see Appendix B). The rest of this section will cover the details of the applications and the reason why they have been selected.

4.7.1 Apache

The first application, Apache web server, has been selected as it was used as the test application in the research by Imran et al. (2021) that inspired this project. It is a popular web server and a common component of web applications. Therefore, Apache is a good example of a real-life workload. A pre-packaged version of Apache shipped with Amazon Linux OS had been used for ASG platform and official Apache image from DockerHub had been used for EKS (see Appendix E for exact versions of the packages).

4.7.2 Node.js (Taro, Taewa. Riwai)

JavaScript (Node.js) and Python are among the most popular programming languages for web development (Statista, 2021; UC Berkeley, 2022). Therefore, they have been selected for development of simple test applications.

First tests had been run against the Taro application which was developed by me as a Node.js and Express (OpenJS Foundation, 2022) web framework learning exercise prior to the start of this project. However, the analysis of the first test results showed that Taro application's ASG performance fluctuated to a great extent even within one series of Fortio runs with results ranging between 1000 to 10000 qps. At the same time CPU utilization and EC2 CPU credit usage seemed stable, so lack of CPU credits or stolen CPU did not seem to be the reason for this behaviour. After running more tests to confirm these results, I concluded that inconsistent results must have been caused by one of the 3rd party libraries (logging, hashing, web framework, etc) used in the application.

Therefore, to avoid the potential effect of the third-party libraries a very simple application that used only Node.js built-in libraries was developed. The consequent testing confirmed that this approach indeed mitigated the Node.js performance inconsistency. The Taewa application developed only makes use of 'http' and 'url' libraries and does not require any third-party components. The application takes an integer 'n' as a parameter and runs a loop of multiplication of random numbers of n times then returns the result trimmed to 44 characters. This ensures that HTTP payload size is constant and can be used to infer the approximate number of requests made from the number of bytes transferred.

However, it was still unclear which third-party library caused issues and if Express framework could offer some performance optimizations over the built-in Node.js http library. To test this a simple app (named Riwai) mirroring Taewa's functionality was built utilizing only the Express framework. The Riwai app showed results very similar to Taewa, this confirmed that current behaviour was not a peculiarity of

a particular web framework but indicative of Node.js performance profile in general. This also showed that Express library was not the cause of inconsistent performance.

4.7.3 Python (Raupi)

A Python app Raupi had been developed to test if different programming language show difference performance profiles. Raupi also mirrors Taewa's functionality and is developed using the Flask web framework and Waitress WSGI server libraries, which are a common choice for Python web applications (Balla et al., 2020; Chan et al., 2019). Raupi is the only app where EKS platform showed superiority over ASG in terms of performance, which was quite unexpected (see 6.1.3).

4.7.4 App packaging and distribution

As shown in the listing of the application repository in Table 4-2 (refer to Appendix B for Application repository URL), each application is stored in a separate directory. Besides the application code each app directory contains a 'package.sh' script that automates app packaging and a 'Dockerfile' containing the code for building the Docker container. The package.sh script first creates a zip file for ASG deployment then it builds the Docker container image of the application and pushes it to the Docker Hub image repository where it is accessed by the Kubernetes nodes.

Application	FIles
Taro	**./app** ./app/package-lock.json ./app/package.json ./app/server.js … ./package.sh ./Dockerfile
Taewa	**./smplapp** ./smplapp/simpleserver.js ./smplapp/package.json ./smplapp/package.sh ./smplapp/Dockerfile
Riwai	**./smplxprss** ./smplxprss/package-lock.json ./smplxprss/package.json ./smplxprss/server.js ./smplxprss/package.sh ./smplxprss/Dockerfile
Raupi	**./smplflask** ./smplflask/app.py ./smplflask/requirements.txt ./smplflask/package.sh ./smplflask/Dockerfile

Table 4-2 Application repository structure

To release a new version of the application a developer needs to simply run the packaging script and it will package and upload the artifacts (ASG package needs to be uploaded manually). The hash of the

latest git commit is used to name application versions. This way it is easier to associate application versions with git commits that contain the code for that version and make sure that ASG and Kubernetes are deploying the same version of the application. Figure 4-4 below summarizes the application packaging and distribution process.

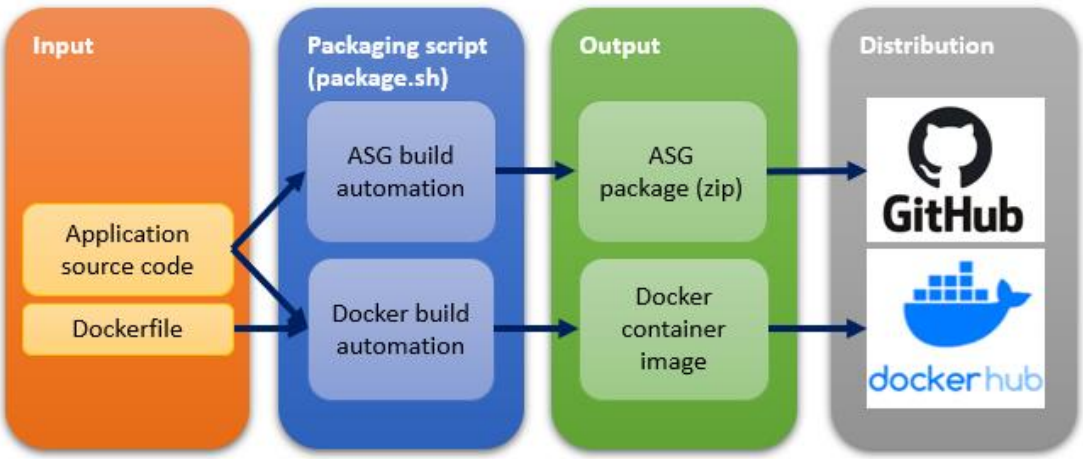


Figure 4-4 Application packaging and distribution

4.8 LOAD TESTING SCRIPT

4.8.1 Overview

Load testing script (or simply load script) was originally thought to be merely a way to automate running a series of Fortio tests. However, as other requirements emerged in the course of the project more functions such as adjusting the infrastructure parameters according to test specifications, retrieving the metrics and uploading the results had been added. Since the script is testing performance of different platforms in the cloud, to eliminate the effect of the internet connection bandwidth and latency limitations, the script is running on the jump host which is deployed in the same AWS account as the infrastructure under testing.

The script consists of three main functional parts – the main module, the configuration module and the metrics collection module. The main module (thesis-infra\data\cli\1.load.sh) is the one that is initialized first. It is responsible for updating the infrastructure and running Fortio series according to the test specs. As it goes through the initialization, warmup, performance, and scaling stages it scales infrastructure, reports its status, and calls other modules.

The configuration module (thesis-infra\data\cli\0.setup.sh) is a collection of test specifications, where each section defines test parameters (such as web endpoint, scaling policy threshold, number of Fortio threads, etc). It started as part of the main module but was separated out to allow for easier change

tracking as it has independent development cycle (adding, removing, and updating the test specifications can be done without changing the testing procedure).

The function of the metrics collection module is to retrieve the CloudWatch and Kubernetes metrics. This is done via the AWS CLI (thesis-infra\data\cli\2.jh-get-data.sh) for CloudWatch metrics and via kubectl (thesis-infra\data\cli\k8s-metrics.sh) for Kubernetes metrics (refer to section 5.1 for details).

<table>
<tr><th>Stage</th><th>ASG</th><th>Kubernetes</th></tr>
<tr><td rowspan="7">Initialization</td><td colspan="2">1) Redirect console output to log file.
2) Authenticate AWS CLI admin user.
3) Read test specifications (configuration module)
4) Define check_stats function</td></tr>
<tr><td>5) Check if ASG is created</td><td>6) Check if managed node ASG is created</td></tr>
<tr><td colspan="2">7) Record CloudWatch metrics start time</td></tr>
<tr><td></td><td>8) Enable EKS worker EC2 metrics collection</td></tr>
<tr><td colspan="2">9) Wait for infrastructure stack to stabilise</td></tr>
<tr><td>11) Configure ASG scaling policy (and scale to max)
12) Update LB health check.</td><td>13) Authenticate K8s.
14) Scale EKS nodes to max.
15) Configure HPA
16) Start K8s metrics collection</td></tr>
<tr><td colspan="2"></td></tr>
<tr><td rowspan="3">Warmup</td><td colspan="2">17) Warmup series: (x10)
a) Wait for 60 sec
b) Perform n-second Fortio run</td></tr>
<tr><td colspan="2">18) Save full specifications of LBs, ASGs and instances</td></tr>
<tr><td colspan="2">19) Control series: (x5)
a) Perform 60-second Fortio run</td></tr>
<tr><td>Performance test</td><td colspan="2">20) Performance series: (x3)
a) Wait for 60 sec
b) Perform 5-minute Fortio run</td></tr>
<tr><td rowspan="3">Scaling test</td><td>21) Flush ASG instances (scale to 0)</td><td></td></tr>
<tr><td colspan="2">22) Scaling series: (x3)</td></tr>
<tr><td>a1) set 99% policy (prevent scale out)
b1) scale ASG to min
c1) wait for scale
d1) back to initial scaling policy</td><td>a2) delete HPA (prevent scale out)
b2) scale pods to min
c2) scale nodes to min
d2) wait for scale
e2) re-create HPA</td></tr>
<tr><td colspan="2">f) Scaling Fortio series: (xN)
f1) Perform 60-second Fortio run</td></tr>
<tr><td>Metrics collection</td><td colspan="2">23) Wait for CloudWatch logs to catch up
24) Record CloudWatch metrics end time
25) Execute metrics collection module
26) Upload the results
27) Send email notifying of test completion</td></tr>
</table>

Table 4-3 Load testing script procedure. Red represents ASG-only steps, Green – Kubernetes-only, Blue – common for both platforms.

Table 4-3 above gives an overview of the testing procedure. Each of these stages will be described in full in the following sections.

4.8.2 Initialization stage

The function of the initialisation stage is to setup logging, authentication and prepare the infrastructure according to the test specifications.

4.8.2.1 Authentication

The initialisation stage starts with authenticating the AWS CLI tool with the credentials of the user who created the platform infrastructure stack (see step 2 in script procedure Table 4-3). This step is necessary for later EKS infrastructure adjustments because not only does the load script need to have EKS permissions for AWS API, but it also needs Kubernetes Role-based access control (RBAC) permissions for the Kubernetes API. This is done automatically in the process of cluster creation for the user who initiated it:

> "Initially, only the creator of the Amazon EKS cluster has system:masters permissions to configure the cluster. To extend system:masters permissions to other users and roles, you must add the aws-auth ConfigMap to the configuration of the Amazon EKS cluster. The ConfigMap allows other IAM entities, such as users and roles, to access the Amazon EKS cluster." (Amazon Web Services, 2022l)

Therefore, custom RBAC config maps must be created for any users besides the one that created the cluster. This was attempted, however aws-auth ConfigMaps configuration turned out to be quite complex and was not successful. The workaround solution was to simply pass the credentials used by the infrastructure deployment script (running on the laptop) to the load script (running on the jump host) instead of setting up proper Kubernetes RBAC configuration for another user. This way the load script was no longer using the EC2 role permissions assigned to the jump host instance, instead it was using full admin permissions of the infrastructure deployment user. This is not a secure configuration, but for the purposes of this project it suffices. The security risk had been partially mitigated by enabling the "NoEcho" property of the CloudFormation parameters to mask sensitive values held there.

4.8.2.2 Test Specifications

The next important step in the initialisation stage is reading the test specifications. The testing parameters for each variation of the experiment are grouped into sections called test specifications and are stored in the configuration module of the load script (thesis-infra\data\cli\0.setup.sh). The name of each test specification is composed of the name of the platform (asg or k8s), application (e.g., taewa) and

the number of nodes. Figure 4-5 shows an excerpt from the configuration module containing k8s_taewa_3 and asg_taewa_3 specifications.

```
if [ "$test" == "k8s_taewa_3" ]; then
warmup_url='3000/?n=20000'
testing_url='3000/?n=20000'
hpa_perc=70
warmup_min_threads=15
warmup_max_threads=25
warmup_cycle_sec=90
scaling_minutes=14
performance_sec=300
cluster_name="C888"
max_pods=6
max_nodes=3
fortio_options="-a -qps -1 -r 0.01 -loglevel Error -allow-initial-errors"
fi

if [ "$test" == "asg_taewa_3" ]; then
warmup_url='3000/?n=20000'
testing_url='3000/?n=20000'
cpu_perc=35
warmup_min_threads=15
warmup_max_threads=25
warmup_cycle_sec=90
scaling_minutes=14
performance_sec=300
max_capacity=3
fortio_options="-a -qps -1 -r 0.01 -loglevel Error -allow-initial-errors"
fi
```

Figure 4-5 Example of test specifications

Table 4-4 below gives an overview of testing parameter values for all specifications. Note that Pod requests and limits are not specified in the configuration module directly, but rather in the application manifest files (see 4.6.2).

Test name	CPU target, %	Frotio thread count	Warm-up duration, sec	Scaling duration, min	Perform-ance duration, sec	Max node count	Max pod count	Pod Requests (and limits)	N (difficulty) numer
k8s_apache_3	70	60-70	130	14	300	3	6	0.7 (1.0)	-
asg_apache_3	70	60-70	130	14	300	3	-	-	-
k8s_apache2_3	70	60-70	130	14	300	3	3	1.1 (2.0)	-
k8s_taewa_3	70	15-25	90	14	300	3	6	0.7 (1.0)	20000
asg_taewa_3	35	15-25	90	14	300	3	-	-	20000
k8s_taewa2_3	70	15-25	90	14	300	3	6	1.1 (1.1)	20000
k8s_taewa_3lite	70	60-70	90	14	300	3	6	0.7 (1.0)	1
asg_taewa_3lite	35	60-70	90	14	300	3	-	-	1
k8s_taewa_6	70	15-25	90	25	300	6	12	0.7 (1.0)	20000
asg_taewa_6	35	15-25	90	25	300	6	-	-	20000
k8s_taewa_3extra	70	5-10	90	14	300	3	6	0.7 (1.0)	80000
asg_taewa_3extra	35	5-10	90	14	300	3	-	-	80000
k8s_riwai_3	70	15-25	90	14	300	3	6	0.7 (1.0)	-
asg_riwai_3	35	15-25	90	14	300	3	-	-	-
k8s_raupi_3	70	15-25	90	14	300	3	6	0.7 (1.0)	-
k8s_raupi2_3	70	15-25	90	14	300	3	3	1.1 (2.0)	-
asg_raupi_3*	35	15-25	90	14	300	3	-	-	-
asg_raupi_3_70	70	15-25	90	14	300	3	-	-	-

Table 4-4 Test specification summary

* results renamed to asg_raupi_3x

The rest of this section contains the overview of the function of different testing parameters and the explanation why the specific values have been chosen.

1. **CPU target**

CPU target is the parameter representing the CPU utilization threshold as a percentage. When the CPU utilization is sustained above the threshold for a certain period, the scaling policy will increase the number of instances running the workloads until the instance limit is reached. In the configuration module the CPU target parameter is represented by hpa_perc variable for Kubernetes platform and cpu_perc variable for ASG platform.

In the ASG platform CPU target is a property of the 'AWS::AutoScaling::ScalingPolicy' resource. It is set in the scaling test stage (see step 22-d1 in the Table 4-3) via the put-scaling-policy command and is controlled by the cpu_perc variable in the load script.

In EKS platform CPU target is a property of the Horizontal Pod Autoscaler (HPA) Kubernetes resource. It is set in the scaling test stage (see steps 15 and 22-e2 in the Table 4-3) via the "kubectl autoscale deployment" command and is controlled by the hpa_perc variable in the load script.

Due to the fact that the t3.medium instances used for testing are dual-core the single-threaded (Node.js) test applications are able to utilize only half of the CPU, so 70% CPU target will never be reached on the ASG instances while Kubernetes nodes will have full CPU utilization due to their ability to run a separate pod for each CPU core.

An attempt had been made to mitigate this difference between the platforms and make a more meaningful scaling comparison. Therefore, the CPU target for ASG tests with single-threaded applications is set to half of that of the Kubernetes tests. E.g., if k8s_taewa_3 (2 active cores) has 70% CPU target, and the asg_taewa_3 (1 active core) has 35% CPU target (see Table 4-4). The hope was that this way the CPU utilization per active CPU core would be the same between platforms and the scaling duration comparison would be more meaningful.

However, upon later investigation of the results it became clear that ASG and HPA CPU targets are quite different quantities. The ASG CPU target is based on the ASGAverageCPUUtilization metric (see AppServerSP resource in aws-asg\asg-template-classicELB.yaml) which represents the average of the overall CPU utilization (including all CPU cores) of the instances (Amazon Web Services, 2022k). However, the HPA CPU target is based on targetCPUUtilizationPercentage metric which is calculated as a percentage of the CPU requests which are specified in CPU units and 1 CPU unit is equal to 1 CPU core (The Linux Foundation, 2022b, 2022e). Therefore, to calculate the Kubernetes CPU target equivalent to

ASGAverageCPUUtilization metric the calculation will have to include the number of cores, number of pods per node, container requests and potentially the number of cores the application can utilize (weather it is single- or multi-threaded). Due to time constraints finding the exact formula was not possible. It was attempted, however the formulas derived could explain only a subset of the experimental data. Therefore, the scaling comparison across platforms was abandoned. Instead, the scaling comparison will be made across applications within one platform as CPU target within each platform is consistent.

2. Fortio thread count

Fortio thread count specifies how many simultaneous threads Fortio uses when running a load test. It is set with "-c" parameter of the Fortio CLI command and is controlled by warmup_max_threads variable. The optimal number of threads for a particular test specification should be big enough so that it does not become a bottleneck but should not be so big that network processing takes too much of the CPU time. To find the maximum performance per thread several Fortio runs with different numbers of threads were executed. This established that maximum to be approximately 1100 qps on t3.medium instance used for the jump host. Therefore, if an app deployment can reach x qps, the minimum number for threads required will be x/1100. To account for jump host and platform performance variability the minimum number is then multiplied by a factor of 1.5 and rounded to the next round number to obtain the final value used in the specification.

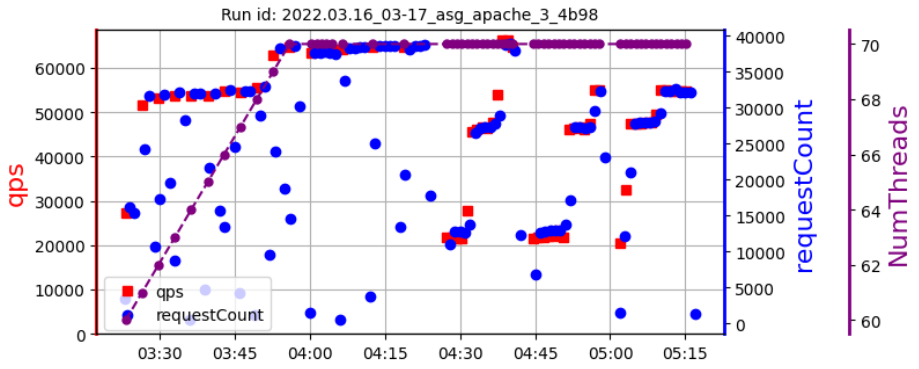


Figure 4-6 Example graph of thread count over time

Figure 4-6 shows that thread count (purple NumThreads metric) does not show correspondence with the request count (blue). If the thread count value was the bottleneck, we expect performance (requestCount metric) to be sensitive to changes in thread count value. The lack of this correspondence in the experiment data proves that the thread count value defined in the test specification is sufficient.

3. **Warm-up duration**

This parameter specifies the length of the Fortio run in the warm-up stage (step 17b of Table 4-3) in seconds. It is set with "-t" parameter of the Fortio CLI command and is controlled by warmup_cycle_sec variable. The deployments of applications that can reach higher qps and need longer warmup cycles. After running several tests to test how long it takes for application deployment to reach maximum qps, the optimal values have been found to be between 90 and 130 seconds.

4. **Scaling test duration**

This parameter specifies the total length of the Fortio runs in each iteration of the scaling test stage (step 22f of Table 4-3) in minutes and is controlled by the scaling_minutes variable. Unlike the warm-up cycle duration parameter this parameter does not control the duration of an individual Fortio run. Instead, it controls the number of iterations of the scaling Fortio series (step 22f) where each run is fixed at 60 seconds (see 4.8.5).

During the development stage it has been found that 14 minutes is enough for all 3 node deployments. The only tests that have a different scaling duration are the 6 node tests (k8s_taewa_6 and asg_taewa_6) where it is set to 25 minutes (see Table 4-4) to account for increased scaling duration due to the increased node number.

5. **Performance test duration**

Performance duration is the duration of the performance Fortio run in seconds (step 20b of Table 4-3) and is controlled by the performance_sec variable. Similarly, to warm-up duration, it is set with "-t" parameter of the Fortio CLI command. To ensure the consistency of performance comparison across different tests the performance duration parameter is constant and has been set to 300 seconds.

6. **Difficulty number**

Difficulty number controls the number of simulated workload cycles that are run for each HTTP query. It is passed to the application via the GET HTTP parameter "n" and is controlled by the testing_url variable. Difficulty numbers have been chosen such that they represent CPU bound (e.g., Taewa extra), network bound (e.g., Taewa lite), and balanced (Taewa) applications.

7. **Max pod and node count**

Max node count controls the number of EC2 instances that run the applications. It is set to 3 in most test due to the sandbox limitations. Several tests with 6 nodes have been conducted. However, after

receiving a warning from sandbox support, the decision was made to keep the node count within the sandbox limit to avoid the sandbox account suspension.

The pod count in EKS deployments is set to one pod per CPU core for normal density deployments (6 pods for 3-node deployment) and one pod per node for low density deployments (3 pods for 3-node deployment).

4.8.2.3 Infrastructure adjustment

After reading the test specifications the load script waits for the infrastructure readiness, by checking if the ASG instances (or EKS nodes) are created (steps 5 and 6). Once this check is successful the experiment run is considered started and the CloudWatch metric collections start time is recorded (step 7). Then the infrastructure is adjusted according to the test specification. If it is an ASG deployment, then the initial ASG scaling policy is configured (step 11), the ASG is scaled to maximum for the warmup and the Load Balancer health check is updated according to the application being tested. In case of EKS deployment first kubectl is authenticated against the EKS cluster (step 13), then the EKS node group is scaled to maximum for the warmup (step 14) and the Kubernetes Horizontal Pod Autoscaler is configured (step 15).

4.8.2.4 Metrics

Since ASG infrastructure detailed (1-minute interval) CloudWatch metrics are enabled in the CloudFormation template, no additional steps are required for ASG deployments. However, EKS node groups only have low resolution (5-minute intervals) EC2 metrics enabled by default (Amazon Web Services, 2020d). To fix this, detailed monitoring is enabled via EC2 API call for each instance (step 8). This is done inside the check_stats function as it runs regularly (see section **Error! Reference source not found.**).

The Kubernetes metrics collection is started in the step 16 of initialization stage. The collection is done via a custom script which runs in the background on the jump host (see section 5.1.3 for details).

4.8.2.5 Logging

Another important function of the initialisation stage is setting up logging. At the start of the load script all console output is redirected to the main log file (step 1) thesis-results/{test_id}/load-k8s.log for EKS platform and thesis-results/{test_id}/load-asg.log for the ASG platform (see Table 5-1). Besides the output and errors of the infrastructure adjustment and testing commands it contains the JSON

descriptions of created AWS resources such as Launch Templates and Load Balancers (step 18), so it is easier to analyse the causes of the failed or anomalous runs. The check_stats function which is used to check the status of the infrastructure between the Fortio runs is also defined in the initialisation stage.

4.8.3 Warmup stage

Once the infrastructure and logging are setup in the initialization stage the warmup stage commences. The warmup stage consists of a series of 60-second Fortio runs separated by 60 second intervals (step 17 a and b). The warmup is needed because the load balancer (LB) that serves as the entry point for the requests (see Figure 4-2) needs some time to scale to support high qps that are required by some of the tests. This is especially obvious in asg_apache_3 tests where qps reach 50000 almost immediately but need another 20-30 minutes to go beyond that. In Figure 4-7 illustrating this behaviour there is a jump in qps just after 1:30, indicating the LB scale out event even though the number of ASG instances (groupInServiceCapacity) stays constant from the beginning.

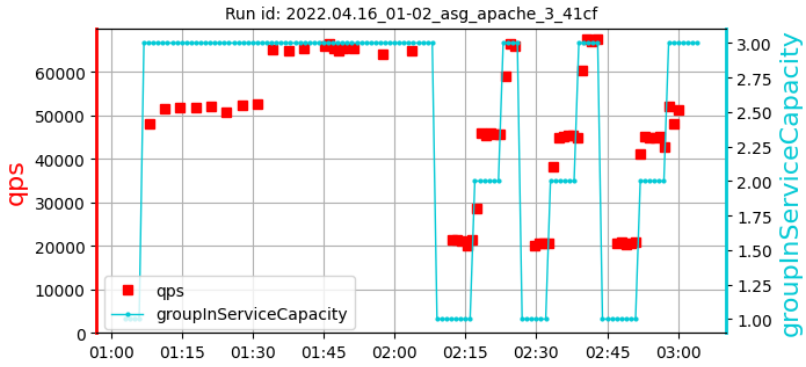


Figure 4-7 Example of asg_apache_3 run groupInServiceCapacity metric

The need for LB warmup is also confirmed in the article "Best Practices in Evaluating Elastic Load Balancing" by Amazon Web Services (2012) "We recommend that you increase the load at a rate of no more than 50 percent every five minutes". However, according to asg_apache_3 test results (Figure 4-7) it takes about 20 minutes of warmup before LB scale-up occurs, which is much longer than what is stated in the AWS article.

Another reason to have the warmup stage is that even some low qps applications need time to warm up on Kubernetes after the cluster creation. On the example of k8s_taewa_3extra run below (Figure 4-8) you can see that it takes about 20 minutes for the qps to reach maximum.

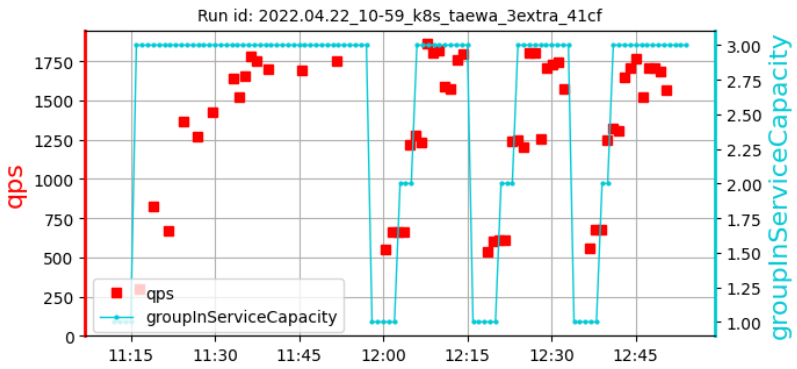


Figure 4-8 Example of k8s_taewa_3extra run groupInServiceCapacity metric

During the early development stages the warmup series didn't have pauses between Fortio runs, which was not enough to scale up to the maximum number of instances for some of the high-performance tests, especially given that some of the early test configurations had higher instance counts (up to 6). The 60 second pauses have been added to the warmup cycle to ensure that the LB had enough time to scale (step 17 a and b).

The warmup stage concludes with a control series (step 19). Besides warming up the load balancer, the purpose of the control series is to see weather HTTP keep-alive affects the performance tests by comparing it to the standard (continuous) performance runs.

4.8.4 Performance test stage

Performance stage is where the performance of the infrastructure is evaluated. It is represented by the performance series (step 20) which consists of three 5-minute Fortio runs separated by 60-second intervals. The test parameters such as number of threads and test duration are loaded from the test specifications in the initialization stage (see 4.8.2.2). The results include the Fortio, CloudWatch and Kubernetes metrics which are covered in section 5.1.

4.8.5 Scaling test stage

Scaling test stage is one of the more complex stages. It consists of two nested loops. The outer loop is the scaling series (step 22) which is repeated 3 times for every experiment run. Here the infrastructure is prepared for each scaling Fortio series. First the application deployment is scaled to minimum (steps 22 b1, b2, c1, c2). It was found that simple reset of scaling settings to defaults does not prevent immediate scale-out on the historical data from the previous scaling test, so additional step has been added to keep the minimum instance number (steps 22 a1 and a2). Then, depending on the platform being tested it either resets the ASG scaling policy (ASG platform) or the Horizontal Pod Autoscaler (EKS platform) (steps

22 d1, e1). Finally, once the infrastructure is prepared for the next scaling run, the scaling Fortio series is executed (step 22f).

HTTP keep-alive effects had to be considered when designing scaling tests as they have profound effect on scaling behaviour. HTTP keep-alive (or HTTP persistent connection) is a technique where HTTP client sends multiple HTTP requests within one TCP connection instead of opening a TCP connection for each HTTP request. This technique is widely used by modern web applications to improve performance and reduce TCP connection management overhead (Abdelsalam et al., 2017; Gourley et al., 2002).

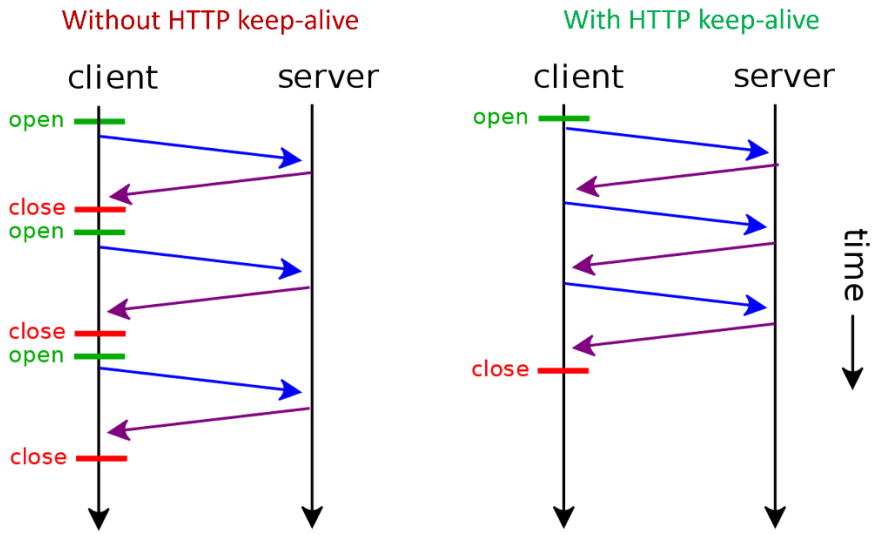


Figure 4-9 Opening and closing TCP connections with and without HTTP keep-alive

HTTP keep-alive poses a challenge in designing the scaling test because as you can see from the requestCount graph on Figure 5-2 all connections are distributed at the start of the test. This prevents the client from accessing the newly scaled backend instances and, therefore, prevents further scaling. Furthermore, as pointed out by Elamin and Paardekooper (2021) long-lived connections caused by HTTP keep-alive cause issues with scaling Kubernetes deployments for the same reason. This has been observed during the development of the early Taewa scaling tests which were originally implemented as single 14-minute Fortio runs. During those early scaling tests, it was found that connections are not distributed evenly across the nodes. This uneven connection distribution manifested as uneven CPU load on the nodes (e.g., node1: 98%, node2: 60%, node3: 0%) which would persist within the full length of a single Fortio run.

The solution that allowed to overcome these unwanted effects was to use a series of 1-minute Fortio runs in the scaling test instead of a single continuous Fortio run. This way the connections were

redistributed every minute allowing the client to take advantage of the newly scaled instances. After the implementation of this change the scaling test started to perform without issues. These effects, however, do not seem to affect the performance Fortio runs, which is confirmed by the results of the control series (see 4.8.3). This is likely because the infrastructure is already warmed-up and scaled to maximum at the time when the performance testing starts.

4.8.6 Metrics collection stage

Metrics collection is the last stage of the load script. While some of the metrics (Fortio and Kubernetes) are already in the results folder, CloudWatch metrics first need to be extracted from the cloud storage (CloudWatch metrics collection is described in detail in section 5.1.1). To ensure that no data is missing from the results a pause had been added after the scaling test stage (step 23) as it takes up to 5 minutes for the metrics to be available after collection by the CloudWatch agent. Then the experiment end time is recorded (step 24) to be later used in the CloudWatch queries (step 25).

After all metrics and logs are collected, the results are uploaded to the intermediate storage (step 26) as the sandbox AWS account is deleted after 4 hours. An example of the results directory can be seen on the Table 5-1. Before the script finishes it notifies the operator that the experiment is complete via JetMail API (step 27). After that the operator can download the results from the intermediate storage and start processing the data.

4.9 SUMMARY

This chapter covered the specifics of the implementation of the experiment. It started with the overview of the automated testing solution and the infrastructure architecture. It then gave a detailed description of each component including infrastructure modules, deployment automation and test applications. Finally, the test specifications and each stage of the load testing script were described in detail. This concludes the discussion of the implementation of the experiment. The next chapter will focus on how the data collected during the experiment is processed and analysed.

CHAPTER 5

DATA PROCESSING

5.1 DATA SOURCES

The main data sources that are used to evaluate the artifacts are AWS CloudWatch metrics, Fortio metrics and Kubernetes metrics. Data for each experiment run is stored in the directory with the naming pattern which is referred to as the "test id". Test id consists of date and time of the execution (e.g., 2022.04.04_22-12), test name (e.g., k8s_taewa_3) and first 4 characters of the git commit hash (e.g., 41cf). The test id corresponding to the values described above will be 2022.04.04_22-12_k8s_taewa_3_41cf. Since the test id contains hash of the git commit it is easy to identify the version of deployment, application and infrastructure code that has been used for that experiment run. Table 5-1 shows what a results folder looks like on the example of test id 2022.04.04_22-12_k8s_taewa_3_41cf[9].

File	Description
./0.setup.sh ./1.load.sh ./2.jh-get-data.sh	load testing script modules (copied from the infrastructure rpository)
./metrics_vars.txt ./mytest	files storing the variables used in the test
./load-k8s.log	the main log file
./ 2022-04-04-233127_k8s_taewa_3 _*.json … …	about 60 json files (depending on the length of the scaling cycle) containing the results of the Fortio runs
./alb-query-template.json ./asg-query-template.json	CloudWatch query templates (copied from the infrastructure rpository)
./alb_data.json ./asg_data.json ./alb-query.json ./asg-query.json	Generated quieries and the data of the CloudWatch metrics
./k8s-deploy-metrics.json ./k8s-hpa-metrics.json ./k8s-metrics.sh	Kubernetes metrics collection script and data (copied from the infrastructure rpository)
./csv	processed data and graphs

Table 5-1 Results folder example (test id: 2022.04.04_22-12_k8s_taewa_3_41cf)

As the results are first copied to the intermediate storage (see section 4.8.6) they first need to be downloaded to the local copy of the results repository for further processing. The thesis-

[9] Raw results of this particular run are available on the following URL on GitHub: https://github.com/taro-ball/thesis-results/tree/master/2022.04.04_22-12_k8s_taewa_3_41cf

infra/etc/gsync.sh[10] script had been developed to simplify this process. It compares the listings of the local and remote results directories and only copies the ones that haven't been synchronised already. All experiment results are available online in the GitHub results repository (see Appendix B).

5.1.1 AWS CloudWatch Metrics

CloudWatch metrics are collected by the metrics collection module of the load script (thesis-infra /data/cli/2.jh-get-data.sh). First, the CW JSON queries are generated using the CW query templates (thesis-infra/data/cli/alb-query-template.json and thesis-infra/data/cli/asg-query-template.json). Then "aws cloudwatch get-metric-data" command is used to run the queries and save the resulting JSON files. For a full list of CW metrics please refer to Table 5-3.

Figure 5-1 below shows an example of EstimatedProcessedBytes metric query from the LB query template (thesis-infra\data\cli\alb-query-template.json). You can see that some of the parameters are preconfigured in the template (Namespace, MetricName), and some are inserted at the run time (lb_name, t_start).

```
{
"MetricDataQueries": [
…
…
{
"Id": "estimatedProcessedBytes",
"MetricStat": {
"Metric": {
"Namespace": "AWS/ELB",
"MetricName": "EstimatedProcessedBytes",
"Dimensions": [
{
"Name": "LoadBalancerName",
"Value": "${lb_name}"k
}
]
},
"Period": 60,
"Stat": "Sum"
},
"Label": "myRequestLabel",
"ReturnData": true
}
],
"StartTime": "${t_start}",
"EndTime": "${t_end}",
"ScanBy": "TimestampAscending"
}
```

Figure 5-1 A Sample CloudWatch query template (from alb-query-template.json)

Table 5-2 below gives a summary of the template parameters and describes their functions.

[10] Here the path relative to the infrastructure repository is given, but usually the script is run from the results repository for convenience

Parameter Name	Configuration	Description
StartTime	run-time	Start of the metrics reporting period (time of the first successful infrastructure readiness check).
EndTime	run-time	End of the metrics reporting period (time 4 minutes after the end of the last Fortio run).
LoadBalancerName/ AutoScalingGroupName	run-time	Name of the AWS resource to filter on.
Stat	Template preconfigured	Statistical aggregation function used on the raw data. For the full list of CW statistics functions refer to the CW documentation (Amazon Web Services, 2022f)
Period	Template preconfigured	Aggregation period for statistics, set to 60 seconds
MetricName/Namespace	Template preconfigured	Name and namespace of the metric

Table 5-2 CloudWatch query parameters

The smallest possible aggregation period for CloudWatch metrics for the namespaces of the collected metrics ("AWS/ELB", "AWS/EC2" and "AWS/AutoScaling") is 60 seconds, so that is the time resolution set for the stat functions in all queries.

The description of **requestCount** metric (see Table 5-3) may make it seem like a reasonable metric for tracking the number of requests, and this is the case when used with Application Load Balancer. However, after the switch to the classic TCP load balancer (see section 4.5.1) it became apparent that TCP listener only registers individual TCP connections rather than HTTP requests thus working on a lower layer of the OSI model. This is a problem since most of the web applications make use of HTTP keep-alive technique to reuse TCP connections for consequent HTTP requests to improve performance (Chang & Mac Vittie, 2008). This means that the numbers reported by the requestCount metric when used with TCP load balancer don't represent the number of HTTP requests but rather several TCP sessions each containing multiple HTTP requests. We can see it clearly if we look at the Figure 5-2 below where the requestCount metric (blue) reports exactly the same values (ranging from 15 to 25) as the NumThreads metric (purple). From this we can infer that by the end of the warmup series we end up with 25 Fortio threads and each Fortio worker thread is establishing only one TCP connection. However, just by looking at these two metrics we can't tell how many requests each worker preformed. We can use the qps (red) metric to see the actual number of HTTP requests ranging from 2000 to 7000 requests per second in this case (see Table 5-4 for Fortio metrics definitions). Therefore, only the **qps** metric reported by Fortio is used for measuring performance.

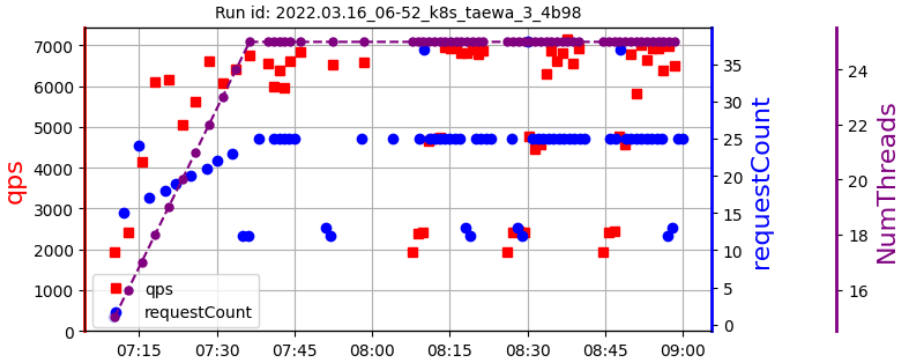


Figure 5-2 Example of requestCount plotted against qps and NumThreads

However, as an additional check from the load balancer side the CW **estimatedProcessedBytes** metric is used to cross-check the qps reported by Fortio. Figure 5-3 below demonstrates that estimatedProcessedBytes shows very close correspondence with qps across different apps and platforms and, therefore, adds assurance to the validity of the qps values.

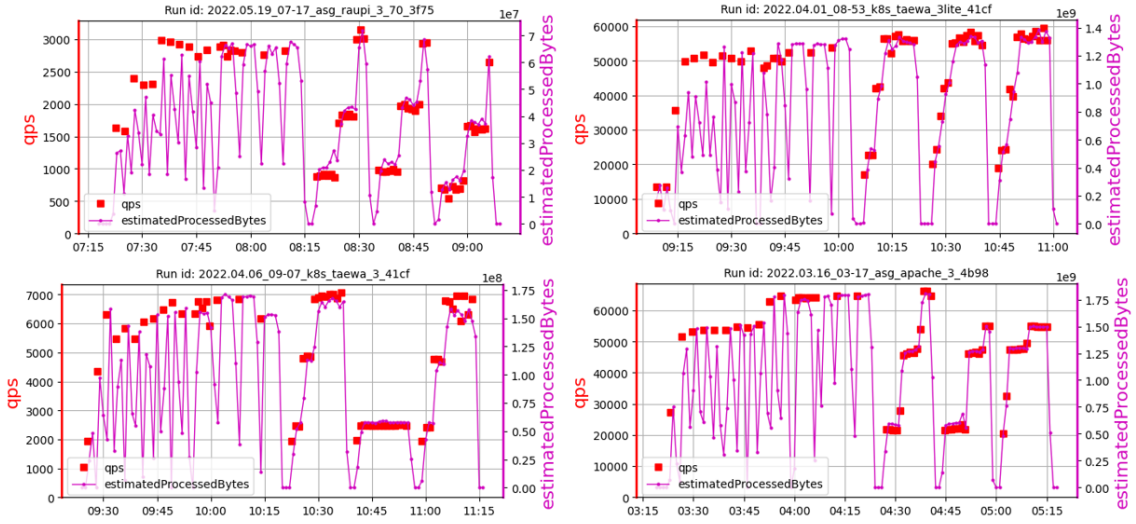


Figure 5-3 estimatedProcessedBytes plotted against qps across different applications and platforms

Table 5-3 below presents all CloudWatch metrics collected with descriptions quoted from official AWS documentation (Amazon Web Services, 2021e, 2022e). Most of them are from AWS/ELB and AWS/AutoScaling namespaces representing Classic LB metrics and ASG metrics respectively. The only

metric from AWS/EC2 namespace is cpuUtilization since the CPU utilisation is a property of an EC2 instance.

CloudWatch Metric Name	Metric description	Stats function	Stats aggre-gation period	Usage
cpuUtilization Unit: Percent Namespace: AWS/EC2	"The percentage of allocated EC2 compute units that are currently in use on the instance. This metric identifies the processing power required to run an application on a selected instance. Depending on the instance type, tools in your operating system can show a different percentage than CloudWatch when the instance is not allocated a full processor core."	99th percentile (useful in analysing "spiky" data)	60s	used to monitor CPU usage of the instances
estimatedProcessedBytes Unit: Bytes Namespace: AWS/ELB	"The estimated number of bytes processed by an Application Load Balancer."	Sum (total network traffic)	60s	Plotted over QPS to confirm performance over time from AWS side
backendConnectionErrors Unit: Count Namespace: AWS/ELB	A number of failed connections from LB to the service running on the EC2 instance "The number of connections that were not successfully established between the load balancer and the registered instances. Because the load balancer retries the connection when there are errors, this count can exceed the request rate. Note that this count also includes any connection errors related to health checks."	Sum (total number of errors)	60s	used to monitor backend errors
groupInServiceCapacity Unit: Count Namespace: AWS/AutoScaling	"The number of instances that are running as part of the Auto Scaling group. This metric does not include instances that are pending or terminating."	Maximum (max number of instances reached)	60s	used to calculate scaling duration for ASG platform
estimatedALBConsumedLCUs Unit: LCU Namespace: AWS/ELB	"The estimated number of load balancer capacity units (LCU) used by an Application Load Balancer."	Maximum (max LB load)	60s	used to monitor LB scaling
Latency Unit: Seconds Namespace: AWS/ELB	"The total time elapsed, in seconds, for the load balancer to successfully establish a connection to a registered instance."	Average	60s	not used as it is reporting per connection latency instead of per request for TCP LBs
requestCount Unit: Count Namespace: AWS/ELB	"The number of […] connections made [to the registered instances] during the specified interval"	Sum	60s	not used as it is reporting connections instead of requests for TCP LBs (see keep alive explanation above)
groupDesiredCapacity Unit: Count Namespace: AWS/AutoScaling	"The number of instances that the Auto Scaling group attempts to maintain."	Maximum (max number of instances requested)	60s	not used
estimatedALBActiveConnectionCount Unit: Count Namespace: AWS/ELB	"The estimated number of concurrent TCP connections active from clients to the load balancer and from the load balancer to targets."	Average (connections per minute)	60s	not used

Table 5-3, List of collected CloudWatch metrics

5.1.2 Fortio Metrics

Fortio metrics are saved in JSON format automatically after each Fortio run (Fortio package docs). In the post-processing, only useful fields are selected and aggregated into a single time-series corresponding to a particular experiment run. The original Fortio JSON output file contains a lot of fields that are not relevant to this research, so below are described only those used in this research:

Field Name	Description	Stats function	Usage
Labels	Labels to distinguish between Fortio runs	N/A (non-numeric)	Used to filter data related to particular Fortio runs (e.g. performance runs or scaling runs)
StartTime	Marks the start time of the Fortio run	N/A (index)	Used as index field
ActualQPS	Queries per second	Average over the duration of the run	Used to evaluate the performance of the infrastructure
NumThreads	Number of worker threads	N/A (constant)	Used to analyse and tune the load test parameters
URL	The URL of the web endpoint being tested	N/A (non-numeric)	Has information on number of cycles (difficulty) of application requests

Table 5-4, List of collected Fortio metrics

5.1.3 Kubernetes metrics

Although Kubernetes metrics can be collected via Container Insights service, it requires additional Kubernetes services such as CloudWatch agent and FuentD to be installed on the EKS nodes (Amazon Web Services, 2022a). This means each Kubernetes node will have to run additional system pods which affect cluster performance. This performance impact was quite noticeable as the Kubernetes nodes used in this project are fairly small due to the sandbox limitations. Therefore, instead of using AWS Container Insights, a custom Kubernetes metrics collection script (data\cli\k8s-metrics.sh) was developed. The script runs on the jump host in the background and periodically (every 60 seconds) saves the deployment and HPA metrics in the JSON format. The resulting Kubernetes metrics contain information about the requested and in-service replicas of the deployment and CPU utilization as well as CPU target for the HPA. Currently the only Kubernetes metrics used in calculating the results is readyReplicas. Therefore, only the deployment metrics are presented in the Table 5-5 below.

Kubernetes Metric Name	Metric description	Stats function	Stats aggre-gation period	Usage
readyReplicas Unit: Count	Number of replicas of the Kubernetes deployment that have been successfully started and reporting healthy.	Maximum (max number of replicas reached)	60s	used to calculate scaling duration for EKS platform
Replicas Unit: Count	Number of desired replicas of the Kubernetes deployment.	Maximum (max number of replicas reached)	60s	Not used

Table 5-5 Collected Kubernetes deployment metrics

5.1.4 Data sources summary

To sum it up. At the end of the experiment all data is saved into the results directory. Fortio metrics are saved automatically after each Fortio run, the CloudWatch metrics are extracted via the AWS CLI and the Kubernetes metrics are collected via a custom script that runs in the background on the jump host. This data is then automatically uploaded to the intermediate storage. After that it can be downloaded to the results repository via the gsync.sh script and is ready for post-processing.

5.2 DATA POST-PROCESSING

Once the data is collected, the post-processing stage begins. Here, the raw data is cleaned and presented in a more convenient format. The raw JSON files are processed via jq utility, and the output files are saved in the csv subfolder.

5.2.1 CloudWatch

Raw CloudWatch metrics for each experiment run are presented as two JSON files - one for ASG metrics and one for ALB metrics, each containing an array of MetricDataResult objects (Amazon Web Services, 2021d). To make plotting and analytics of CloudWatch metrics easier thesis-infra\data\processing\csv_gen.sh script has been developed. The script uses jq utility to convert each MetricDataResult object into a CSV timeseries. This way each resulting CSV file represents one CloudWatch metric for a particular experiment run.

5.2.2 Fortio

Raw Fortio output is not presented as a time-series but rather as a collection of JSON files each containing an object describing a particular Fortio run. To convert this data to a timeseries thesis-infra\data\processing\fortio_prc.sh script has been developed. The script selects useful fields from each

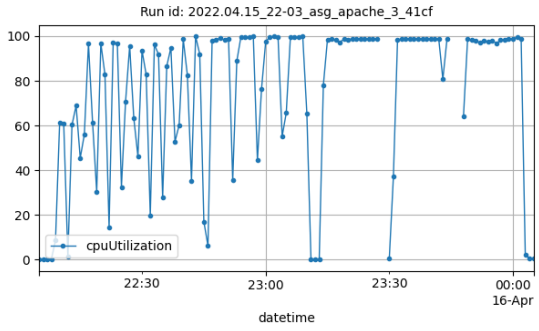
JSON object and appends the resulting timeseries row to the output file. This way the resulting file (e.g., thesis-results\2022.03.16_03-17_asg_apache_3_4b98\csv\2022.03.16_0317_asg_apache_3_4b98.json) contains Fortio metrics for a particular experiment run and each line in the file contains the data of one Fortio run.

5.3 ANALYTICS AND PRESENTATION

Once the data is converted to timeseries files it can be plotted and further aggregated to provide more insights into infrastructure performance and behaviour.

5.3.1 Plotting

Plots allow a researcher to better spot trends and anomalies in the data as compared to looking at raw numerical data. In this project plotting is done via python with libraries including Pandas, NumPy and Matplotlib. Plotting of AWS metrics is done thesis-infra\data\processing\simplegraph.py script which takes the path to the CSV file containing CloudWatch metric timeseries as input, and creates a PNG image of the plot as output. The script uses Pandas' library "plot" function to plot the metric as a function of time. CloudWatch plots are used to see if there were any anomalies during the run and as feedback during the development of load test and apps. Since the time resolution of CloudWatch metrics is quite coarse (only one datapoint per minute) and are only provided on a best effort basis (there may be gaps in the data), markers are used to emphasize the actual data on the plot so that the viewer is not deceived by a seeming continuity of the plot line.



To get better insights into the data it is useful to plot Fortio and CloudWatch metrics against each other. In this project this kind of advanced plotting is done via the thesis-infra\data\processing\foldergraph.py script. Since this script needs to plot multiple metrics, it takes the path to the load test results folder and the name of the CloudWatch metric as parameters and plots the selected CloudWatch metric against Fortio QPS metric. Since Fortio QPS metric is reported per run (not

per unit of time) it is treated as discreet and QPS values are displayed as markers which are not connected by lines, while the CloudWatch metric presentation is like the one in the simplegraph.py. These combined graphs are extremely useful as they clearly show the connection between the performance of the infrastructure (qps metric) and the state of AWS resources (e.g., EC2 instance CPU utilization or number of ASG in-service instances). For example, on the Figure 5-4 you can see a distinct association between queries per second (qps) and the number of instances (groupInServiceCapacity metric).

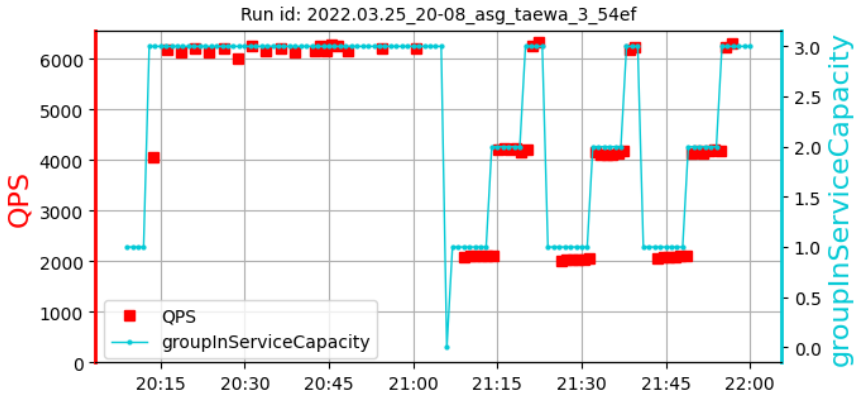


Figure 5-4 Example of asg_taewa_3 groupInServiceCapacity metric

Another advantage of graphs is that it is easy to identify the stages on the graphs and if something went wrong it is easy to understand where it happened. The Figure 5-5 below shows how load script stages can be identified on the graphs (for information on load script stages please refer to section 4.8).

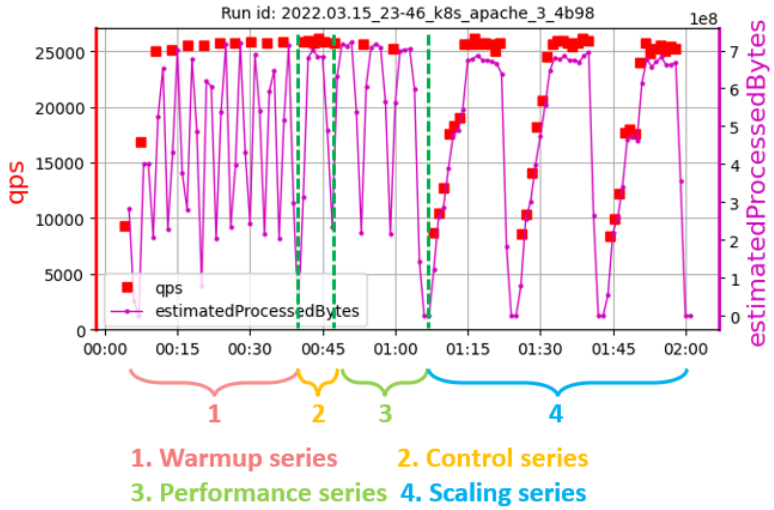


Figure 5-5 Load testing script stages highlighted on a graph

5.3.2 Results Explorer

Looking at the separate graphs is useful during the initial phases of development and testing. However, as the number of tests grows it becomes important to present the data in a more systematic way. To achieve this Results Explorer (thesis-infra\data\processing\resultsExplorer.ipynb) Jupyter notebook has been developed. The notebook analyses the data and generates a group of HTML files each representing a particular test specification. Each HTML file contains a table, where each row contains graphs and aggregated metrics of a particular experiment run.

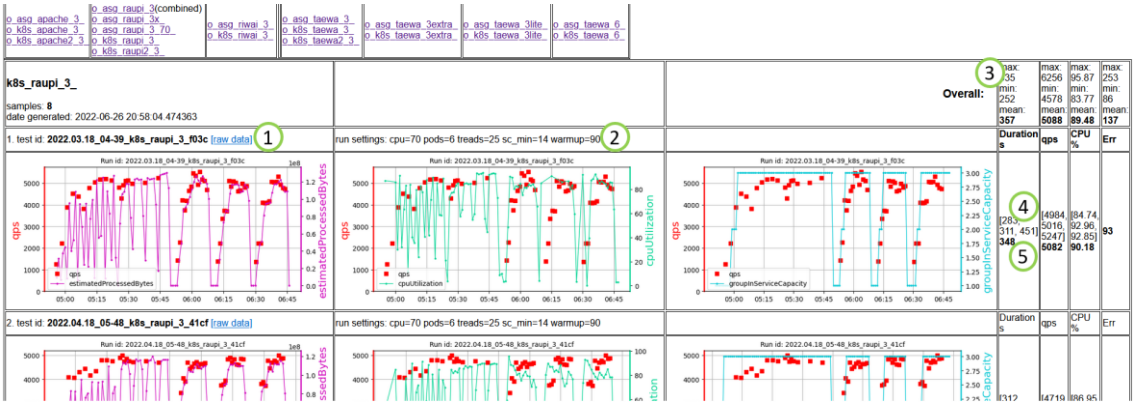


Figure 5-6, Example of Results Explorer table

The Figure 5-6 shows an example of a Results Explorer view. The first row contains the name of the test and the overall metrics averaged across all runs in the table (3). Below that each experiment data is presented in two rows. The first experiment row starts with the test id, the link to the folder with raw data (1) and the test settings (2). The test settings are read from the actual test log. This way if the test spec changes (which should not normally happen) it will be visible in the Results Explorer. The second experiment row starts with three graphs showing qps plotted against LB traffic, ASG CPU utilization and number of active instances respectively. The values in the square brackets (4) show values for each individual test iteration in an experiment run while the bold numbers below them (5) show the mean value for the whole run. The script also generates a CSV file with the overall values for further analysis in Excel.

The Results Explorer is hosted on the GitHub Pages and is accessible via https://taro-ball.github.io/thesis-results/ URL (see Appendix B for details, Appendix E for a sample view).

5.3.3 Data processing algorithm

This section describes how the values of the dependent variables presented in Table 6-10 and Appendix D are calculated. The dependent variable value calculation is done in the Results Explorer Jupyter notebook introduced in the previous section. Results Explorer notebook employs Pandas and NumPy Python libraries which are commonly used in scientific data analysis (Fuhrer et al., 2021; McKinney, 2011) to ensure consistency and accuracy of the results.

5.3.3.1 Data processing algorithm architecture

As discussed in section 3.3 the dependent variables used in this research are qps, scaling duration, CPU utilization and the number of errors. The source data files that are used to calculate the values of dependent variables are represented by green rectangles on Figure 5-7. Table 5-6 below presents a summary of the source data files.

Shorthand	Path relative to the results repository	Dependent variable
Fortio JSON	{test_id}\csv\{test_id}.json	Raw qps
Pods JSON	{test_id}\k8s-deploy-metrics.json	Scaling duration (EKS)
Instances CSV	{test_id}\csv\metric_groupInServiceCapacity.csv	Scaling duration (ASG)
CPU CSV	{test_id}\csv\metric_cpuUtilization.csv	CPU utilization
Error CSV	{test_id}\csv\metric_backendConnectionErrors.csv	Errors

Table 5-6 Source data files

These source data files are fed to the calculation algorithm which produces the individual experiment run results as well as overall aggregated results across all runs. The calculation algorithm consists of 3 nested loops (see Figure 5-7): the test loop (1), the run loop (2) and the spec loop (3).

The run loop iterates through each results directory matching the spec name. Here the source data files are processed to calculate the dependent variable values for each Fortio run in scaling and performance series (turquoise rectangles). The CPU utilization and scaling duration values need additional processing, which is done in the test loop (1). Since there are three Fortio tests in each series, to calculate the results for one experiment run, a mean value across the three Fortio tests is derived from the run lists (purple rectangles). These intermediate values are displayed in the experiment results rows of the Results Explorer view. Then all values from the run lists are appended to the overall lists to be further processed in the spec loop.

.

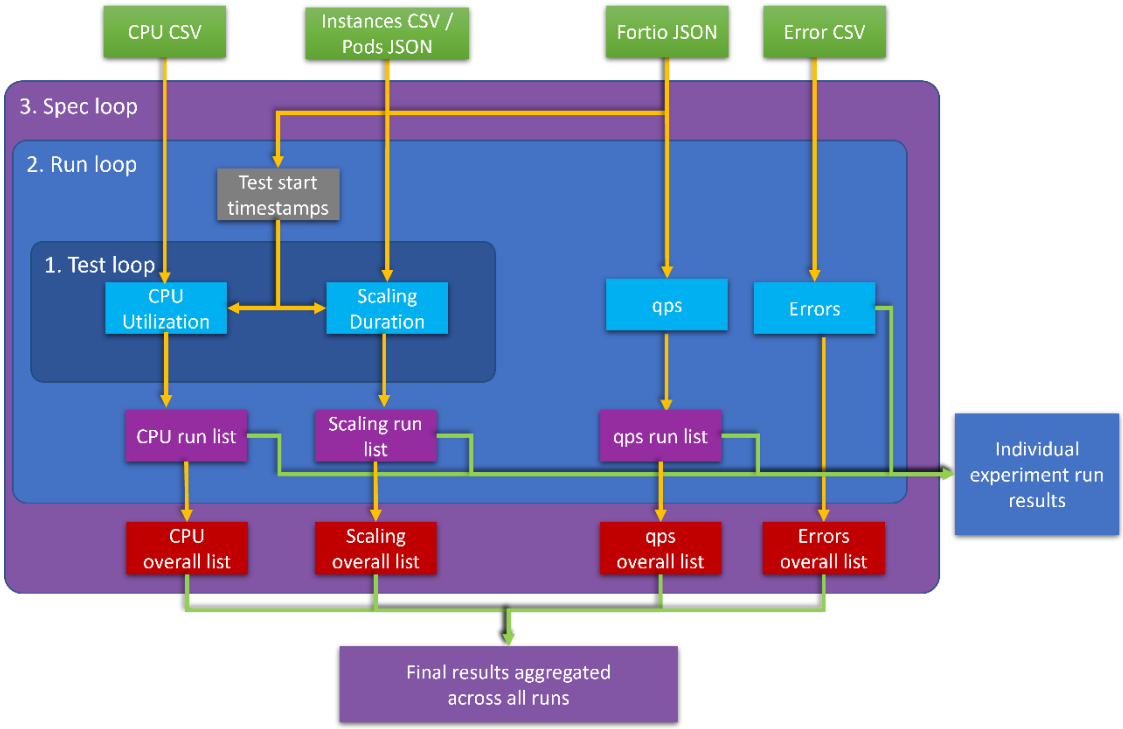


Figure 5-7 Experiment data analytics diagram

The spec loop iterates through the test specifications. This is where the overall minimum, maximum and mean values of each metric are calculated from the overall lists (red) after all nested loops are completed. These overall values are the ones that can be seen in the experiment result Table 6-10, Appendix D and the Results Explorer view.

The overall qps, Error number and scaling duration values are rounded to whole numbers. CPU utilization, on the other hand, is rounded to 2 decimal places. This rounding is done only during the final output operation, all intermediate calculations are done with full precision.

5.3.3.2 Dependent variable calculation

1. Raw qps

The values of raw qps dependent variable are calculated from Fortio JSON file (see Table 5-6). Fortio metrics data (Fortio JSON) is central to the calculation of the results as it not only contains the qps data, but also the timestamp of the test starts needed for calculating the CPU and scaling duration values. First, the performance tests values for all 3 iterations from the performance series (see Table 4-3) are filtered from the Fortio JSON into a performance data frame[11] using the "performance" labels. Then the qps values are extracted from it and saved as qps run list. Qps run list is used to

[11] Data frame is a data structure similar to a table used in Pandas data analytics library (Pandas, 2022).

calculate qps run mean value, then the qps run list is appended to the qps overall list. Once all runs have been iterated, the minimum, maximum and mean of the qps overall list are taken and recorded as the raw qps results.

2. CPU utilization

The CPU utilization dependent variable value is calculated from the CPU CSV timeseries. In the test loop Fortio timestamps from the performance data frame are used to find the datapoints that belong to a particular iteration of the performance series. Only three middle datapoints are taken from 5-minute performance test period to calculate the mean CPU value for the test because the values for the first and the last minute are unreliable. This is caused by the timestamp mismatch as CloudWatch timestamps always start at zero seconds (e.g., 10:01:00, 10:02:00, etc.) while Fortio run can start any time. This means the first or the last minute of the performance test almost always shows lower CPU values. This is because the test started somewhere in between the minutes so the first part of the minute where the instance has been idle affected the mean value of the whole minute. Then, similarly to the qps calculation, the test values are gathered in the run list to calculate the run mean value; and each run list is appended to the overall list which in turn is used to calculate the minimum, maximum and mean CPU utilization values.

3. Errors

Unlike the previous calculations, error values extracted from the Error CSV file are not averaged across the tests in the experiment run as there is no added value at looking at errors per Fortio run and it easier to gauge the state of the experiment by looking at only one number. Instead, the sum of the number of errors from the start of the performance series until the end of the scaling series is computed and added directly to the overall list from which the overall values are calculated. If there is an unusually big number of errors the specific stage where it happened can be easily identified on the error graphs.

4. Scaling duration

As discussed above, CPU utilization, qps and errors are calculated directly from the corresponding metrics. Scaling duration on the other hand must be calculated based on the number of in-service instances in ASG experiments or the number of healthy deployment replicas in EKS experiments. Also, this is the only calculation done on the scaling series data (all calculations described above

were based on performance series data). The reason why the in-service instance data cannot be used for computing scaling duration in EKS experiments is because it takes some time for the deployment replicas to start on the new node. Therefore, the node will be reporting healthy while the application is not running yet. This will result in scaling duration value to be calculated shorter than it should be.

The scaling duration calculation begins with determining the start timestamps of each scaling series iteration, which is done in a similar fashion as for the CPU utilization described above. The only difference is that this time Fortio JSON is filtered on the "scaling" labels. Then the timestamp of the first datapoint in the scaling timeseries (Pods JSON or Instances CSV) where the number of instances reaches maximum is subtracted from the test start timestamp to obtain the scaling duration value. Similarly, to CPU calculation, this is done in the test loop for every iteration of the scaling series. Then in the run loop the run mean value is calculated, and the overall mean value is calculated in the spec loop.

5.3.4 Batch processing

As mentioned above, several shell and python scripts has been developed to process and plot the data. However, even with this automation, running all these tasks manually turned out to be tedious and error prone, especially when bug-fixes and new features had been added to the data processing scripts and all data had to be repeatedly re-processed. So, an overarching batch-processing script has been developed to combine all these tasks into one tool (process.sh). Process.sh takes a test id pattern as parameter and then runs all data processing scripts over the subset of tests that match the pattern. It first executes post-processing scripts (csv_gen.sh and Fortio_prc.sh) to generate the cleaned-up data. Then, it executes the plotting scripts (simplegraph.py and foldergraph.py) to plot the metrics. And finally executes the resultsExplorer.ipynb to update the result explorer. The process of integrating a new test result into total statistic (including plotting the graphs) takes under a minute, so it is easy to refine aggregated statistics as new results arriving.

5.4 SUMMARY

This chapter discussed the collection and processing of the metrics used in infrastructure comparison. The process starts with copying the raw data files from the intermediate storage into the results repository. The raw data first goes through the post-processing stage where the raw CloudWatch

and Fortio metrics are cleaned up and converted into the time series for easier plotting and analysis via jq scripts. Then the images containing the graphs for all metrics are generated via the Python scripts. Next, the Results Explorer notebook utilises the graphs and cleaned-up timeseries data to generate the interactive web-view for exploring the results. Here the dependent variable values for each experiment are calculated and aggregated to give the mean values for each test specification. These results will be discussed in detail in the next chapter.

CHAPTER 6

FINDINGS AND DISCUSSION

In this chapter the findings of the experiment will be presented and discussed. The first section will cover the results per application and discuss the effects of infrastructure platform, pod density, difficulty, and node number on the performance. Then the results across applications will be discussed. Finally, some of the interesting effects encountered during the testing will be reviewed.

6.1 RESULTS ANALYSIS

The analysis of results for each application in this chapter below includes the infrastructure platform and pod density comparisons. However, due to time constraints only the Taewa application was examined for additional variables such as difficulty, web framework and number of nodes. The first row of each result table in this section (e.g., Table 6-1) contains the name of the test (e.g., k8s_apache_3) with number of samples given in the brackets. Since each experiment run contains three performance tests, the number of samples equals to the number of experiments multiplied by 3. The platform result tables (e.g., Table 6-1) also display the performance profile, which is a summary of performance and efficiency comparison for a particular application.

6.1.1 Apache

1. Platform

As show in Table 6-1 below Apache EKS deployment shows significantly lower performance of only 45.4% in weighted qps and 44.6% in raw qps as compared to ASG deployment. Therefore, ASG would be the recommended platform for Apache deployments.

Test name (number of samples)		k8s_apache_3 (13x3)	asg_apache_3 (11x3)
Independent variable	**Platform**	**EKS**	**ASG**
Dependent variables	**Relative raw qps**	0.446 (25515)	1 (57233)
	Relative weighted qps	0.454 (260.9)	1 (574.6)
	Scaling duration	442	618
	Performance profile	EKS shows worse performance (44.6% of ASG) and worse efficiency (45.4% of ASG)	

Table 6-1 Apache platform results[12]

It is expected that EKS would show lower performance due to the Kubernetes overhead, however this does not explain such a big difference. One potential explanation could be the more complex network setup of EKS which imposes a limit on what the raw qps can reach. However if we look at the Taewa lite results we can see that EKS network infrastructure is capable of reaching as high as 55000 raw qps (see 2022.04.01_21-08_k8s_taewa_3lite_41cf results in Results Explorer). Another explanation could be the varying default settings of the packages across platforms. However, the packages are expected to be optimized for the platform. An alternative explanation is that this observation has something to do with the architecture of the application (implementation of the algorithm that the application uses) as hypothesized by Imran et al. (2021).

2. Pod density

Test name (number of samples)		k8s_apache_3 (13x3)	k8s_apache2_3 (9x3)
Independent variable	**Pod density**	**Normal** (6pod/3node)	**Low** (3pod/3node)
Dependent variables	**Relative raw qps**	0.446 (25515)	0.457 (26141)
	Relative weighted qps	0.454 (260.9)	0.465 (266.9)
	Scaling duration	442	329

Table 6-2 Apache pod density results

The results of the test with low and normal pod density (Table 6-2) show very similar numbers. Which is not surprising since Apache is a multi-threaded application and therefore can utilize all available cores with just one application process. This shows that the overhead of running an extra Apache pod per

[12] Here and later normal pod density results are used unless otherwise stated

node is minimal. The scaling duration is slightly shorter in low density EKS deployments which may be attributed to lower overhead for running fewer pods.

6.1.2 Taewa

1. **Platform**

Test name (number of samples)		k8s_taewa_3 (12x3)	asg_taewa_3 (11x3)
Independent variable	**Platform**	**EKS**	**ASG**
Dependent variables	**Relative raw qps**	1.077 (6428)	1 (5968)
	Relative weighted qps	0.576 (66.1)	1 (114.7)
	Scaling duration	388	656
	Performance profile	EKS shows marginally better performance (107.7% of ASG) and worse efficiency (57.6% of ASG)	

Table 6-3 Taewa platform results

Similarly, to Apache (45%), Taewa EKS deployment shows results only 57% of ASG performance in weighted qps. However, since Node.js is a single-thread language, EKS can offer improved performance with 2 pods running on each node fully utilizing the CPU cores. Although even with this advantage EKS deployment shows only marginally better results of 107% in raw qps as compared to ASG while using almost twice as much CPU (97% EKS vs 52% ASG). Therefore, the platform recommendation would depend on the type of the instances utilised. For burstable EC2 instances (e.g., class T) ASG platform is recommended as it will allow for reducing cost by spending less CPU credits (Amazon Web Services, 2020c). For other general-purpose instances (e.g., class M) on the other hand, both platforms offer similar raw qps performance per node, so the choice will depend on factors other than performance.

2. **Pod density**

Test name (number of samples)		k8s_taewa_3 (12x3)	k8s_taewa2_3 (13x3)	asg_taewa_3 (11x3)
Independent variable	**Pod density**	**Normal** (6pod/3node)	**Low** (3pod/3node)	**N/A**
Dependent variables	**Relative raw qps**	1.077	0.874	1
	Relative weighted qps	0.576	0.807	1
	Scaling duration	388	363	656
	CPU utilization	97.32	56.36	52.03

Table 6-4 Taewa pod density results

The lower pod density (1 pod per node instead of one pod per core) test for Taewa EKS deployment showed 80% of ASG's weighted qps which is (unlike lower density results for Apache) much better than the normal density results of 57%. The likely explanation for this difference is that the OS had moved all the Kubernetes system processes to the free CPU core, this way the application process didn't have to share the core with them and thus showed a 23% (80%-57%) increase in efficiency. However, this 20% increase in efficiency is accompanied by only 4.3% increase in CPU utilization. This indicates that there are other factors contributing to this performance improvement rather than only the CPU performance. Overall, this shows that the effect of Kubernetes overhead is quite significant for small size nodes such as the ones used in this experiment, which was also pointed out by (Tamiru et al., 2020). This means that in real world EKS deployments larger sized nodes may show increased performance as compared to ASG deployments of the same size.

3. Difficulty

For the Taewa application independent variables other than the platform and node density have been tested this includes the number of nodes, web framework and difficulty settings. Taewa lite and Taewa extra are the variations of the Taewa app with different difficulty settings where Taewa extra represents a CPU-bound workload and Taewa lite represents network-bound workload. If we look at the test results in the order of increasing difficulty: Taewa lite (n=1) –> Taewa (n=20000) –> Taewa Extra (n=80000) there is no apparent trend in the dependent variables (see Table 6-5)

Test name (number of samples)		**k8s_taewa_3lite (12x3)**	**k8s_taewa_3 (12x3)**	**k8s_taewa_3extra (8x3)**
Independent variable	**N (difficulty)**	1	20000	80000
Dependent variables	**Relative qps**	0.951	1.077	1.006
	Relative w. qps	0.595	0.576	0.558
	Scaling duration	385	387	388

Table 6-5 Taewa difficulty results

From these results we can conclude that difficulty does not seem to affect performance and scaling duration. Therefore, the architecture of the application seems to be the main factor that determines the performance profile.

4. **Web Framework**

Test name (number of samples)		**k8s_riwai_3 (5x3)**	**k8s_taewa_3 (12x3)**
Independent variable	**Web framework/ library**	**Express**	**http (Node.js native)**
Dependent variables	**Relative qps**	1.129	1.077
	Relative w. qps	0.598	0.576
	Scaling duration	354	388

Table 6-6 Taewa web framework results

Two applications written in Node.js have been tested (Taewa and Riwai) and both show similar performance profiles even though they employ different web frameworks. Therefore, these results are not a peculiarity of a particular web framework but indicative of Node.js performance profile in general. Therefore, these results may be extrapolated to other Node.js web applications as well (see section 4.7.2 for more information on Node.js applications).

5. **Number of nodes**

Test name (number of samples)		**K8s_taewa_3 (12x3)**	**k8s_taewa_6 (2x3*)**	**asg_taewa_3 (11x3)**	**asg_taewa_6 (2x3*)**
Independent variable	**Number of nodes**	3	6	3	6
Dependent variables	**Raw qps**	6428	9945	5968	9774
	Relative raw qps	**1.077**	**1.017**	**1**	**1**
	Weighted qps	66.1	108.3	114.7	182.3
	Relative weighted qps	**0.576**	**0.594**	**1**	**1**
	Scaling duration	656	1289	656	1289

Table 6-7 Taewa number of nodes results

* Low sample count for 6-node deployments is due to sandbox limitations (see 4.8.2.2)

If we look at the number of nodes as independent variable as demonstrated in Table 6-7, we can see that the EKS performance results in both relative raw and relative weighted qps don't show significant change. Both EKS and ASG platforms only show about 55% increase in performance when the number of nodes doubles. The scaling duration doubled together with the number of nodes as expected.

6.1.3 Raupi

6. Platform

Test name (number of samples)		k8s_raupi_3 (8x3)	asg_raupi_3_70 (12x3)
Independent variable	**platform**	**EKS**	**ASG**
Dependent variables	**Relative raw qps**	1.731	1
	Relative weighted qps	1.602	1
	Scaling duration	357	650
	Performance profile	EKS shows better performance (173.1% of ASG) and better efficiency (160.2% of ASG)	

Table 6-8 Raupi platform results

Unlike the previous applications, Raupi EKS deployment shows better performance figures then ASG deployment in both raw (73% better) and weighted (60% better) qps (see Table 6-8). Raupi is the only application where EKS platform surpassed ASG in the weighted qps metric by a reasonable margin. This is especially surprising because Apache (the other multi-threaded application in the experiment) shows exactly the opposite trend, with EKS weighted qps being the lowest (55% lower than ASG) relative to ASG among all applications. The pod density analysis below reveals that this is likely caused by the difference in the Python binaries used in the platforms.

7. Pod density

Test name (number of samples)		k8s_raupi_3 (8x3)	k8s_raupi2_3 (6x3)	asg_raupi_3_70 (12x3)
Independent variable	**Node density**	**Normal** (6pod/ 3node)	**Low** (3pod/ 3node)	**-**
Dependent variables	**Raw qps**	5088	3947	2939
	Relative raw qps	**1.731**	**1.343**	**1**
	Weighted qps	59.2	56.9	35.5
	Relative weighted qps	**1.668**	**1.602**	**1**
	Scaling duration	344	357	650

Table 6-9 Raupi pod density results

Similarly, to Apache, the relative weighted qps performance of the EKS Raupi deployment does not seem to depend on the node density (see Table 6-9) with both normal and low density deployments showing values around 160% of ASG. However, if we look at the raw qps results as compared to ASG

deployment (which also runs only one application instance per node) we can see that even in this case, where ASG platform has an advantage due to lower overhead, EKS shows better performance with 5088 raw qps against 3947 of ASG. The possible explanation for this result is that it is caused by the difference between the Python packages of ASG and EKS platforms. Although the proper controls have been in place for this test (both platforms use official packages of the same version of python, (see Appendix E for package details), it is still possible that this result is caused by differences in binaries or the default package settings. To confirm this hypothesis, more tests need to be conducted using other Python application or using different OS. For example, using Ubuntu ASG nodes and the official Python Ubuntu container image. Therefore, for now this remains only a hypothesis.

6.1.4 Cross-application comparison

1. Weighted qps

As presented in Figure 6-1, ASG platform shows better efficiency (weighted qps) than EKS, with Taewa and Apache showing only half the efficiency on the EKS platform. This is expected since EKS is built on top of ASG and has additional overhead which is also pointed out by Jindal et al. (2017). In this regard the Raupi application is an outlier, showing over 60% efficiency improvement on EKS platform as compared to ASG platform.

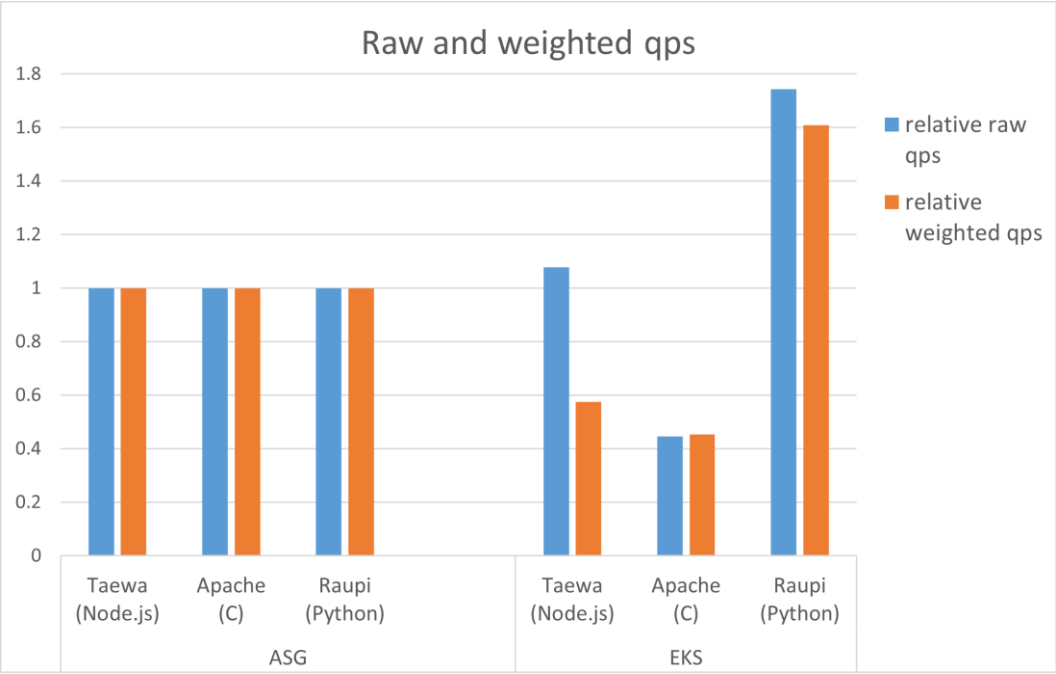


Figure 6-1 Performance Results

2. Raw qps

The performance (raw qps) comparison shows varying results, as each application exhibits a distinct performance profile with Raupi showing better EKS performance (173.1% of ASG), Taewa showing marginally better EKS performance (107.7% of ASG) and Apache showing worse EKS performance (44.6% of ASG). While the Taewa result is expected, both the unusually low Apache EKS performance and unusually high Raupi EKS performance need further investigation.

Improved performance of Kubernetes over ASG in Taewa and Raupi deployments can be explained by higher workload density and more efficient resource management (Kubernetes frees resources as needed and can utilize partially used nodes). Therefore, relatively higher performance gains in the paper by Imran et al. (2021) could be due to the fact that they used much bigger nodes (8-32 cores, 15-60Gb RAM) and therefore had relatively lower overhead as compared to the t3.medium instances (2 cores, 4 Gb RAM) used in the current research.

3. Scaling duration

Scaling duration results (Figure 6-2) are quite consistent across applications, suggesting that CPU based scaling is not affected by application differences.

4. CPU utilisation

CPU utilisation results (Figure 6-2) show that overall EKS exhibits higher CPU utilization due to higher density of application deployment. However, with lower EKS efficiency, the higher CPU utilization does not always produce proportionately better performance. For example, in Taewa experiments although EKS deployments showed almost double CPU utilisation the performance increased only by 7%.

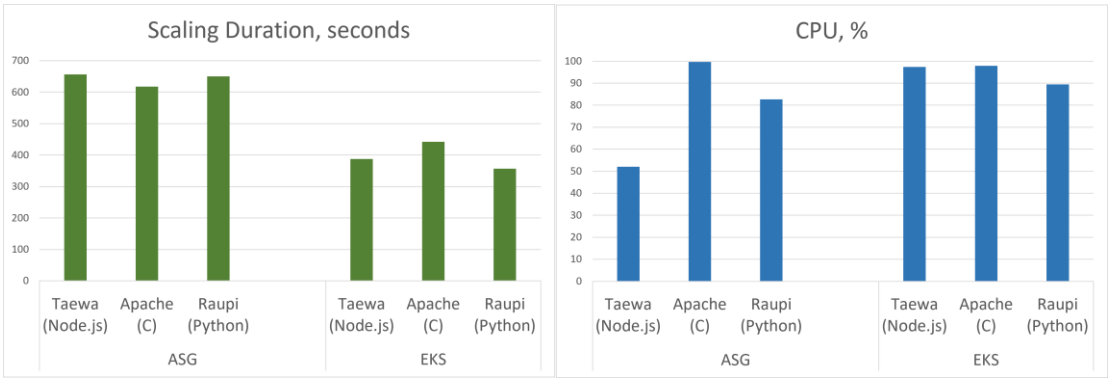


Figure 6-2 Scaling and CPU results

6.1.5 Result summary table

The detailed experiment summary result table produced by the Results Explorer notebook is presented in Appendix D, it contains the full information for each test including the spec name and minimum and maximum values of dependent variables. Table 6-10 below presents a more systematic view where the dependent variable values are organised according to the independent variables and contains only mean values of the dependent variables for ease of reading. The first column displays the names of the dependent variables, the second column displays the infrastructure platform. The next six columns represent the applications and their variations and are marked by different colours. They are subdivided into smaller columns based on the pod density (the number of pods relative to the number of nodes) ratio. Since ASG deployments do not have pods, the ASG results are put into the normal pod density (6 pods/3 nodes) column.

Dependent variables	Indepen-dent variables	Apache [C]		Taewa (n=20000) [Node.js/http]			Taewa lite (n=1) [Node.js/http]		Taewa extra (n=80000) [Node.js/http]		Riwai [Node.js/Express]		Raupi [Python/Flask]	
	Platform	3 pods/ 3 nodes	6 pods/ 3 nodes	3 pods/ 3 nodes	6 pods/ 3 nodes	12 pods/ 6 nodes	3 pods/ 3 nodes	6 pods/ 3 nodes	3 pods/ 3 nodes	6 pods/ 3 nodes	3 pods/ 3 nodes	6 pods/ 3 nodes	3 pods/ 3 nodes	6 pods/ 3 nodes
Raw qps	EKS	26141	**25515**	5218	**6428**	9945	NT	**49757**	NT	**1623**	NT	**5302**	3947	**5088**
	ASG	NA	**57233**	NA	**5968**	9774	NA	**52333**	NA	**1625**	NA	**4696**	NA	**2939**
Relative raw qps	EKS	0.457	**0.446**	0.874	**1.077**	1.017	NT	**0.951**	NT	**0.999**	NT	**1.129**	1.343	**1.731**
	ASG	NA	**1.000**	NA	**1.000**	1.000	NA	**1.000**	NA	**1.000**	NA	**1.000**	NA	**1.000**
Weighted qps	EKS	266.9	**260.9**	92.6	**66.1**	108.3	NT	**514.9**	NT	**17.6**	NT	**53.9**	59.2	**56.9**
	ASG	NA	**574.6**	NA	**114.7**	182.3	NA	**864.7**	NA	**31.9**	NA	**90.2**	NA	**35.5**
Relative weighted qps	EKS	0.465	**0.454**	0.807	**0.576**	0.594	NT	**0.595**	NT	**0.553**	NT	**0.598**	1.668	**1.602**
	ASG	NA	**1.000**	NA	**1.000**	1.000	NA	**1.000**	NA	**1.000**	NA	**1.000**	NA	**1.000**
CPU utilization	EKS	97.93	**97.81**	56.36	**97.32**	91.86	NT	**96.64**	NT	**92.02**	NT	**98.31**	66.68	**89.48**
	ASG	NA	**99.60**	NA	**52.03**	53.62	NA	**60.52**	NA	**50.99**	NA	**52.05**	NA	**82.81**
Scaling duration	EKS	329	**442**	363	**388**	743	NT	**385**	NT	**387**	NT	**354**	344	**357**
	ASG	NA	**618**	NA	**656**	1289	NA	**607**	NA	**611**	NA	**619**	NA	**650**

Table 6-10 Experiment results

NT – Not tested
NA – Not Applicable

6.2 CHALLENGES AND LESSONS LEARNED

A number of challenges have been encountered during the development and execution of the experiment. Some of them related to running applications in the cloud, others were due to the architecture of modern applications and infrastructure deployment. This section will discuss some of them.

6.2.1 Noisy neighbours and stolen CPU

"Noisy neighbour" and "stolen CPU" are the effects of other users of the AWS public cloud. Both of these phenomena are caused by a situation where one of the virtual instances on the same physical host is using a lot of resources, starving other instances.

As defined by M. Ryan and Lucifredi (2018) "Stolen CPU time represents the share of time the instance's virtual CPU has been waiting for a real CPU while the hypervisor is using it to service another virtual processor" while "noisy neighbour" is a similar effect affecting network bandwidth and disk I/O.

The unusually low raw qps values and longer scaling durations in performance tests often correspond to the lower CPU utilization numbers which can be attributed to both stolen CPU and noisy neighbour effects. The qps of the affected runs (e.g. 2022.03.26_08-14_k8s_taewa_3lite_9d2d, 2022.04.05_19-47_k8s_taewa_3lite_41cf, etc) can be up to 10% lower than the mean value for the test specification which affects both ASG and EKS results, although only small percentage of the total number of tests seems to be affected.

6.2.2 AWS API issues

Another issue that was encountered was slow or failed AWS API calls. For example, slow response of the EC2 instance creation API caused some of the tests to scale slower or fail to scale to maximum as in example in Figure 6-3 below.

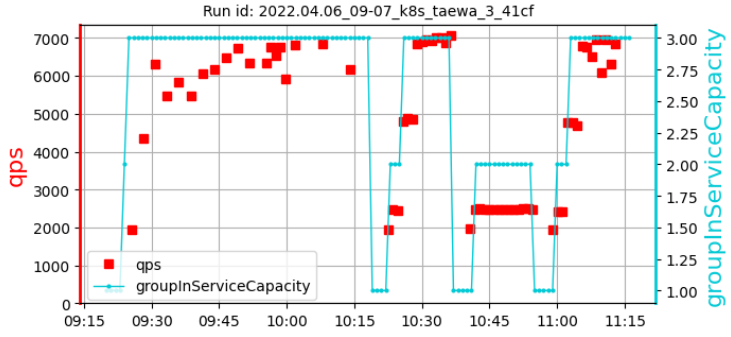


Figure 6-3 Example of failed scaling

Another issue that caused the EC2 instance creation API to fail was the lack of instances of t3.medium size in the AWS availability zone. This happened multiple times in the jump host deployments which caused them to fail, and the experiment had to be re-started. Although this may be a limitation of the sandbox and not of AWS in general.

6.2.3 HTTP keep alive effects

As discussed in 4.8.5 and 5.1.1 there are two major consequences of HTTP keep-alive in this project. Firstly, it affects TCP LB requestCount metric making it unreliable, which can go unnoticed for applications with relatively short keep-alive sessions like Apache. Therefore, alternative sources of request throughput performance metric must be considered. Secondly, it causes uneven connection distribution in scaling tests, since all connections are distributed at the beginning of the test and new instances simply can't receive new requests. This has been addressed by splitting the scaling test into 1-minute runs.

6.2.4 EKS deployments

Although it is possible to deploy EKS via CloudFormation directly, the lesson learnt was that in practice this is a quite complex task. Therefore, recommendation for beginners is to use eksctl instead as it can generate functional EKS stacks without the need to understand the intricacies of the EKS setup.

6.2.5 Data processing

The default data structures of some data sources are not convenient for processing with data analytics tools like Pandas. And since all the data in this experiment came in JSON format it was important to have a convenient tool for processing JSON files like jq.

6.3 SUMMARY

In this chapter the findings of the experiment were presented and discussed. First each application's performance was discussed in the context of independent variables, such as platform, pod density, etc. Then the dependent variable trends across all applications were analysed. Finally, the challenges and lessons learnt were reviewed. The next chapter will summarise the results and discuss the limitations and potential future directions for the research.

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 CONCLUSION

This research was undertaken to answer the question whether AWS ASG or AWS EKS platform offers better scaling and performance capabilities. To answer the question the performance and scaling benchmarks were selected based on similar research in the field, the experimental setup was devised, and a robust automated testing solution was developed to execute the experiment in a controlled and consistent way. The tests were conducted on test applications which were developed and packaged for this experiment. Over 160 runs of the experiment were executed to test different combinations of independent variables which amounts to over 380 hours of execution time.

The results of the experiment show that neither of the platforms shows a clear advantage as the performance profiles vary a lot between applications. The Apache application has better performance and efficiency on ASG infrastructure. Both Node.js applications (Taewa and Riwai) show marginally better performance on EKS infrastructure, although with much lower efficiency. The Python application (Raupi) shows better performance and efficiency on EKS infrastructure.

An attempt was made to isolate the reason for performance variations by examining the contribution of factors such as programming language, web framework, concurrency, difficulty, and pod density. The results show that the web framework and difficulty do not seem to have a significant effect on the performance, while the programming language seems to be the determining factor of the application performance profile. The concurrency does not seem to be a performance determining factor either as the two examined multi-threaded applications show opposite results with Apache performing better on ASG infrastructure and Raupi performing better on EKS infrastructure. The EKS pod density comparison shows that the Kubernetes overhead seems to have a considerable effect on the performance. This means that EKS deployments with bigger node sizes may show better results in comparison to ASG deployments.

The scaling comparison turned out to be hard to perform between the platforms because of the difference in the way scaling is implemented on ASG and EKS. Therefore, the comparison was only made

between applications within one platform. The results showed that scaling seems to depend only on the infrastructure platform with scaling duration being very similar between all tested applications.

7.2 LIMITATIONS

The experiment was conducted in the sandbox environment, therefore, some of the observed issues could be specific to the sandbox and not AWS in general. Another limitation of the sandbox is the amount and sizing of EC2 instances (maximum four t3.medium instances including the jump host). This means the system overhead may be bigger than in many of the real-world deployments which use bigger node sizes. Moreover, because of sandbox EC2 instance number limitations it was not possible to run both EKS and ASG experiment in the same AWS account at the same time, which means it is likely that EKS and ASG experiments were running in different AZs, since AZ IDs are mapped independently for each AWS account (Amazon Web Services, 2020e).

Due to limited time the number of samples used for the comparisons is not optimal, which may affect the accuracy of the results. Simple web applications have been used for testing purposes this means results for more complex real-world applications may differ.

CloudWatch metrics have low resolution and are provided on best-effort basis. Although generally reliable, they still have some gaps which may affect the results.

In addition, the cloud computing industry is moving extremely fast, and hardware and software gets constantly improved, so the test results are most likely just a moment in time result as EKS and EC2 are constantly being evolved by AWS.

7.3 RECOMMENDATIONS AND FUTURE WORK

Since the results are inconclusive on what causes the observed differences in application performance profiles, more work can be done to investigate the causes of these differences. For example, research can be done to see if containerised versions of Apache, Node.js and Python have different default settings from packages shipped with the OS of ASG instances. Also, more applications written in different programming languages can be tested to confirm if the programming language plays a defining role in determining application performance profile.

As mentioned in the limitations, only T3 instances have been tested, so the 'M' type instances will be great candidates for future research as these provide consistent performance and don't incur the overhead of CPU credits and bursting on a best-effort basis.

More advanced scaling setups can be tested that use parameters for scaling other than CPU target, such as custom Prometheus metrics used in the paper by Imran et al. (2021). Besides, this research only considers scaling-out performance, so future research may compare the whole scaling cycle efficiency including cooldown policies.

This thesis considered only the performance aspect when comparing VM's and containers, so more research can be conducted in the context of business issues, developer productivity, AWS consumption costs and carbon footprint consumed. Carbon footprint analysis can be especially interesting when applied to running single-threaded workloads on dual-core VMs.

Another potential direction is to consider AWS running on Firecracker microVMs which may offer improved performance due to lower overhead(Amazon Web Services, 2018).

REFERENCES

Abdelsalam, A., Luglio, M., Roseti, C., & Zampognaro, F. (2017). Evaluation of TCP wave performance applied to real HTTP traffic. *2017 International Symposium on Networks, Computers and Communications (ISNCC)*, 1–6.

Al Jawarneh, I. M., Bellavista, P., Bosi, F., Foschini, L., Martuscelli, G., Montanari, R., & Palopoli, A. (2019). Container orchestration engines: A thorough functional and performance comparison. *ICC 2019-2019 IEEE International Conference on Communications (ICC)*, 1–6.

Amazon Web Services. (2012). *Best Practices in Evaluating Elastic Load Balancing*. Amazon Web Services, Inc. https://aws.amazon.com/articles/best-practices-in-evaluating-elastic-load-balancing/

Amazon Web Services. (2018, November 26). *Firecracker – Lightweight Virtualization for Serverless Computing | AWS News Blog*. https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing/

Amazon Web Services. (2020a). *Amazon EKS Quick Start*. https://aws-quickstart.github.io/quickstart-amazon-eks/

Amazon Web Services. (2020b). *Ensuring idempotency—Amazon Elastic Compute Cloud*. https://docs.aws.amazon.com/AWSEC2/latest/APIReference/Run_Instance_Idempotency.html

Amazon Web Services. (2020c). *General purpose instances—Amazon Elastic Compute Cloud*. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/general-purpose-instances.html

Amazon Web Services. (2020d). *Monitor your instances using CloudWatch—Amazon Elastic Compute Cloud*. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-cloudwatch.html

Amazon Web Services. (2020e). *Regions and Zones—Amazon Elastic Compute Cloud*. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html#az-ids

Amazon Web Services. (2021a). *Application load balancing on Amazon EKS - Amazon EKS*. https://docs.aws.amazon.com/eks/latest/userguide/alb-ingress.html

Amazon Web Services. (2021b). *Auto Scaling groups—Amazon EC2 Auto Scaling*. https://docs.aws.amazon.com/autoscaling/ec2/userguide/auto-scaling-groups.html

Amazon Web Services. (2021c). *Configuration and credential file settings—AWS Command Line Interface*. https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-files.html

Amazon Web Services. (2021d). *MetricDataResult—Amazon CloudWatch*. https://docs.aws.amazon.com/AmazonCloudWatch/latest/APIReference/API_MetricDataResult.html

Amazon Web Services. (2021e). *Monitor CloudWatch metrics for your Auto Scaling groups and instances—Amazon EC2 Auto Scaling*. https://docs.aws.amazon.com/autoscaling/ec2/userguide/ec2-auto-scaling-cloudwatch-monitoring.html

Amazon Web Services. (2021f). *Run commands on your Linux instance at launch—Amazon Elastic Compute Cloud*. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/user-data.html

Amazon Web Services. (2022a). *Amazon EKS and Kubernetes Container Insights metrics—Amazon CloudWatch*. https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/Container-Insights-metrics-EKS.html

Amazon Web Services. (2022b). *Amazon EKS Anywhere – Amazon Web Services*. Amazon Web Services, Inc. https://aws.amazon.com/eks/eks-anywhere/

Amazon Web Services. (2022c). *AWS Load Balancer Controller add-on—Amazon EKS*. https://docs.aws.amazon.com/eks/latest/userguide/aws-load-balancer-controller.html

Amazon Web Services. (2022d). *CloudWatch metrics for your Application Load Balancer—Elastic Load Balancing*. https://docs.aws.amazon.com/elasticloadbalancing/latest/application/load-balancer-cloudwatch-metrics.html

Amazon Web Services. (2022e). *CloudWatch metrics for your Classic Load Balancer—Elastic Load Balancing*. https://docs.aws.amazon.com/elasticloadbalancing/latest/classic/elb-cloudwatch-metrics.html

Amazon Web Services. (2022f). *CloudWatch statistics definitions*. https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/Statistics-definitions.html

Amazon Web Services. (2022g). *Container Orchestration & Management on AWS*. Amazon Web Services, Inc. https://aws.amazon.com/containers/

Amazon Web Services. (2022h). *Getting started with Amazon EKS – eksctl*. https://docs.aws.amazon.com/eks/latest/userguide/getting-started-eksctl.html

Amazon Web Services. (2022i). *Managed node groups—Amazon EKS*. https://docs.aws.amazon.com/eks/latest/userguide/managed-node-groups.html

Amazon Web Services. (2022j). *Pod networking in Amazon EKS using the Amazon VPC CNI plugin for Kubernetes—Amazon EKS*. https://docs.aws.amazon.com/eks/latest/userguide/pod-networking.html

Amazon Web Services. (2022k). *PredefinedMetricSpecification—Amazon EC2 Auto Scaling*. https://docs.aws.amazon.com/autoscaling/ec2/APIReference/API_PredefinedMetricSpecification.html

Amazon Web Services. (2022l). *Provide access to other IAM users and roles after cluster creation in Amazon EKS*. Amazon Web Services, Inc. https://aws.amazon.com/premiumsupport/knowledge-center/amazon-eks-cluster-access/

Artac, M., Borovssak, T., Di Nitto, E., Guerriero, M., & Tamburri, D. A. (2017). Devops: Introducing infrastructure-as-code. *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, 497–498.

Balla, D., Maliosz, M., & Simon, C. (2020). Open source faas performance aspects. *2020 43rd International Conference on Telecommunications and Signal Processing (TSP)*, 358–364.

Benevides. (2016, February 24). *10 things to avoid in docker containers*. Red Hat Developer. https://developers.redhat.com/blog/2016/02/24/10-things-to-avoid-in-docker-containers

Bernstein, D. (2014). Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, *1*(3), 81–84. https://doi.org/10.1109/MCC.2014.51

Brawn, B. S., & Gustavson, F. G. (1968). Program behavior in a paging environment. *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part II*, 1019–1032.

Burns, B., Beda, J., & Hightower, K. (2019). *Kubernetes: Up and running: dive into the future of infrastructure*. O'Reilly Media.

Chan, J., Chung, R., & Huang, J. (2019). *Python API Development Fundamentals: Develop a full-stack web application with Python and Flask*. Packt Publishing Ltd.

Chang, P., & Mac Vittie, L. (2008). The Fundamentals of HTTP. *F5 White Paper*. https://wtit.com/wp-content/uploads/2015/07/f5-white-paper-fundamentals-of-http-.pdf

Cloud Native Computing Foundation. (2021). *CNCF Annual Survey 2021*. https://www.cncf.io/reports/cncf-annual-survey-2021/

Docker. (2021, November 2). *Docker Engine release notes*. Docker Documentation. https://docs.docker.com/engine/release-notes/prior-releases/

Donohue, T. (2020, December 3). *The differences between Docker, containerd, CRI-O and runc*. Tutorial Works. https://www.tutorialworks.com/difference-docker-containerd-runc-crio-oci/

Elamin, M., & Paardekooper, P. (2021). *Scaling of containerized network functions*. University of Amsterdam. https://rp.os3.nl/2020-2021/p70/report.pdf

Espe, L., Jindal, A., Podolskiy, V., & Gerndt, M. (2020). Performance Evaluation of Container Runtimes. *CLOSER*, 273–281.

Fuhrer, C., Solem, J. E., & Verdier, O. (2021). *Scientific Computing with Python: High-performance scientific computing with NumPy, SciPy, and pandas*. Packt Publishing Ltd.

Gourley, D., Totty, B., Sayer, M., Aggarwal, A., & Reddy, S. (2002). *HTTP: The definitive guide*. O'Reilly Media, Inc.

Gupta, B., Mittal, P., & Mufti, T. (2021). *A review on Amazon web service (AWS), Microsoft azure & Google cloud platform (GCP) services*.

Hammond, D. (2019, October 27). AWS EKS: Managed setup with CloudFormation. *Medium*. https://medium.com/@dhammond0083/aws-eks-managed-setup-with-cloudformation-97461300e952

Honig, R. (2019, July 16). *7 OpenStack Alternatives You Can't Ignore*. https://www.stratoscale.com/blog/openstack/openstack-alternatives/

Hummer, W., Rosenberg, F., Oliveira, F., & Eilam, T. (2013). Testing idempotence for infrastructure as code. *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, 368–388.

Imran, M., Kuznetsov, V., Dziedziniewicz-Wojcik, K. M., Pfeiffer, A., Paparrigopoulos, P., Trigazis, S., Tedeschi, T., & Ciangottini, D. (2021). Migration of CMSWEB cluster at CERN to Kubernetes: A comprehensive study. *Cluster Computing*. https://doi.org/10.1007/s10586-021-03325-0

Jackson, S. L. (2011). *Research methods: A modular approach* (2nd ed). Wadsworth/Cengage Learning.

Jindal, A., Podolskiy, V., & Gerndt, M. (2017). Multilayered cloud applications autoscaling performance estimation. *2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2)*, 24–31.

Luksa, M. (2017). *Kubernetes in action*. Simon and Schuster.

McKinney, W. (2011). pandas: A foundational Python library for data analysis and statistics. *Python for High Performance and Scientific Computing*, *14*(9), 1–9.

Merkel, D. (2014). Docker: Lightweight linux containers for consistent development and deployment. *Linux Journal*, *2014*(239), 2.

Miell, I., & Sayers, A. (2019). *Docker in practice*. Simon and Schuster.

Miller, S., Siems, T., & Debroy, V. (2021). Kubernetes for Cloud Container Orchestration Versus Containers as a Service (CaaS): Practical Insights. *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 407–408.

Murtaza, G., Ilkhechi, A. R., & Salman, S. (2020). *Impact of GDPR on Service Meshes* (p. 5). https://cs.brown.edu/courses/csci2390/2019/assign/project/report/gdpr-service-meshes.pdf

OpenJS Foundation. (2022). *Express—Node.js web application framework*. https://expressjs.com/

Pandas. (2022). *pandas.DataFrame—Pandas 1.4.3 documentation*. https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html

Poetra, F. R., Prabowo, S., Karimah, S. A., & Prayogo, R. D. (2020). *Performance analysis of video streaming service migration using container orchestration*. *830*(2), 22100-. https://doi.org/10.1088/1757-899X/830/2/022100

Poniszewska-Marańda, A., & Czechowska, E. (2021). Kubernetes cluster for automating software production environment. *Sensors*, *21*(5), 1910.

Rad, B. B., Bhatti, H. J., & Ahmadi, M. (2017). An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)*, *17*(3), 228.

Ryan, M., & Lucifredi, F. (2018). *AWS System Administration: Best Practices for Sysadmins in the Amazon Cloud*. O'Reilly Media, Inc.

Ryan, T. P., & Morgan, J. P. (2007). Modern experimental design. *Journal of Statistical Theory and Practice*, *1*(3–4), 501–506.

Statista. (2021). *Most used languages among software developers globally 2021*. Statista. https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/

Tamiru, M. A., Tordsson, J., Elmroth, E., & Pierre, G. (2020). An Experimental Evaluation of the Kubernetes Cluster Autoscaler in the Cloud. *2020 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 17–24.

The Linux Foundation. (2022a). *Container Runtimes*. Kubernetes. https://kubernetes.io/docs/setup/production-environment/container-runtimes/

The Linux Foundation. (2022b). *Horizontal Pod Autoscaling*. Kubernetes. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

The Linux Foundation. (2022c). *Kubernetes Components*. Kubernetes. https://kubernetes.io/docs/concepts/overview/components/

The Linux Foundation. (2022d). *Overview of Kubernetes Services*. Kubernetes. https://kubernetes.io/docs/tutorials/kubernetes-basics/expose/expose-intro/

The Linux Foundation. (2022e). *Resource Management for Pods and Containers*. Kubernetes. https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

UC Berkeley. (2022). 11 Most In-Demand Programming Languages in 2022. *Berkeley Boot Camps*. https://bootcamp.berkeley.edu/blog/most-in-demand-programming-languages/

Vayghan, L. A., Saied, M. A., Toeroe, M., & Khendek, F. (2019). Microservice based architecture: Towards high-availability for stateful applications with Kubernetes. *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, 176–185.

Wang, Y., & Cai, J. (2022). A Node-Level Model for Service Grid. *Mobile Information Systems*, *2022*, e4720114. https://doi.org/10.1155/2022/4720114

Yussupov, V., Soldani, J., Breitenbücher, U., Brogi, A., & Leymann, F. (2021). From Serverful to Serverless: A Spectrum of Patterns for Hosting Application Components. *CLOSER*, 268–279.

APPENDICES

APPENDIX A

Board of Studies approval letter

From: Sarah Humphries <Sarah.Humphries@weltec.ac.nz>
Sent: Tuesday, September 21, 2021 10:56
To: pavel.gri@outlook.com <pavel.gri@outlook.com>
Cc: Marta Vos <Marta.Vos@whitireia.ac.nz>
Subject: Research Proposal Application

21 September 2021

Dear Pavel Grigoryev

The School of Information Technology and Business Board of Studies met on 20 September 2021 to consider your application to proceed with your Research Proposal entitled **"Migration from EC2 to EKS based infrastructure. Performance testing and recommendations."**

The Board has approved the research project ***No: RP2021-035*** for your Master of Information Technology qualification. This research proposal has been assessed as of low ethical risk and you can now proceed as planned with your research project.

Please insert this email as an appendix into your report.

Yours sincerely

Mary-Claire Proctor
Chair
School of IT and Business
Board of Studies

APPENDIX B

GitHub resources

Infrastructure and testing code repository: https://github.com/taro-ball/thesis-infra

Application code repository: https://github.com/taro-ball/thesis-apps

Results repository (raw data): https://github.com/taro-ball/thesis-results

Interactive results explorer: https://taro-ball.github.io/thesis-results/

APPENDIX C

Infrastructure git repository listing

Filename	Description	Covered in
./deploy-all.sh	Main infrastructure deployment script	Section 4.6.1
./aws-asg ./aws-asg/asg-template-classicELB.yaml ./aws-asg/deployASG.sh	Infrastructure code and deployment script for ASG platform	Section 4.6.3
./aws-tools ./aws-tools/deployJumpHost.sh ./aws-tools/exclSecrets.sh.example ./aws-tools/jumphost.yaml	Infrastructure code and deployment script for jump host	Section 4.6.4
./data/cli ./data/cli/0.setup.sh ./data/cli/1.load.sh ./data/cli/2.jh-get-data.sh ./data/cli/3.upload.sh.example	Load testing script modules and test specifications	Section 4.8
./data/cli/alb-query-template.json ./data/cli/asg-query-template.json ./data/cli/k8s-metrics.sh	metrics collection and CloudWatch query templates	Section 5.1
./data/processing ./data/processing/cleanup.sh ./data/processing/csv_gen.sh ./data/processing/foldergraph.py ./data/processing/Fortio_prc.sh ./data/processing/process.sh ./data/processing/resultsExplorer.ipynb ./data/processing/run_explorer.sh ./data/processing/simplegraph.py	Data processing scripts	Sections 5.2 and 5.3
./eksctl ./eksctl/Cgenerated.yml	Infrastructure automation for EKS platform	Section 4.6.2
./etc ./etc/collateral/ ./etc/snippets/ ./etc/gsync.sh	Auxiliary automation scripts, code snippets and listing of AWS resources from live deployments (collateral)	Sections 5.1 and 4.5.2
./eksctl ./eksctl/Cgenerated.yml	Kubernetes cluster code for EKS platform	Section 4.6.2
./k8s ./k8s/apache.yaml ./k8s/apache2.yaml ./k8s/raupi.yaml ./k8s/raupi2.yaml ./k8s/riwai.yaml ./k8s/taewa.yaml ./k8s/taewa2.yaml ./k8s/apply-k8s.sh	Kubernetes application manifests	Section 4.6.2
./k8s/cluster-autoscaler-autodiscover.yaml	Kubernetes cluster autoscaler manifest	Section

APPENDIX D

Detailed results table

Test name	N of Samples	Platform	Application	Duration max	Duration mean	Duration min	qps max	qps mean	qps min	CPU max	CPU mean	CPU min	Errors max	Errors mean	Errors min
k8s_apache_3_	13	k8s	apache	627	**442**	310	26815	**25515**	24550	99.18	**97.81**	95.98	90	**40**	21
asg_apache_3_	11	asg	apache	716	**618**	366	65527	**57233**	48678	99.90	**99.60**	97.57	1404	**1048**	802
k8s_apache2_3new	2	k8s	apache2	384	**321**	262	26733	**26436**	26025	98.68	**97.86**	97.27	71	**48**	24
k8s_apache2_3_	9	k8s	apache2	533	**329**	248	26654	**26141**	25532	99.15	**97.93**	96.32	69	**44**	20
asg_raupi_3x_	8	asg	raupi	355	**274**	30	3218	**2969**	2746	85.86	**83.07**	79.23	1502	**1263**	1002
k8s_raupi_3_	8	k8s	raupi	535	**357**	252	6256	**5088**	4578	95.87	**89.48**	83.77	253	**137**	86
asg_raupi_3 (combined)	20	asg	raupi	-	-	-	3218	**2939**	2711	89.07	**82.81**	77.39	1710	**1301**	1002
asg_raupi_3_70_	12	asg	raupi	759	**650**	383	3124	**2919**	2711	89.07	**82.63**	77.39	1710	**1326**	1014
k8s_raupi2_3_	6	k8s	raupi2	410	**344**	268	4178	**3947**	3672	67.52	**66.68**	65.43	219	**102**	23
asg_riwai_3_	7	asg	riwai	780	**619**	390	4881	**4696**	4159	52.67	**52.05**	51.18	1319	**1203**	1013
k8s_riwai_3_	5	k8s	riwai	482	**354**	299	5626	**5302**	4624	99.88	**98.31**	90.34	382	**198**	40
k8s_taewa_3lite_	12	k8s	taewa	511	**385**	308	55341	**49757**	42553	99.71	**96.64**	86.90	407	**237**	46
asg_taewa_3lite_	11	asg	taewa	712	**607**	385	65120	**52333**	42420	63.96	**60.52**	58.78	1840	**1453**	809
k8s_taewa_3_	12	k8s	taewa	504	**388**	267	6900	**6428**	5485	99.76	**97.32**	85.46	270	**185**	79
asg_taewa_3_	11	asg	taewa	751	**656**	589	6279	**5968**	5136	55.69	**52.03**	51.23	1515	**1203**	911
asg_taewa_3extra_	9	asg	taewa	747	**611**	383	1736	**1625**	1399	54.52	**50.99**	50.15	1534	**1225**	818
k8s_taewa_3extra_	8	k8s	taewa	512	**387**	272	1860	**1623**	1194	99.71	**92.02**	71.10	537	**233**	69
k8s_taewa_6_	2	k8s	taewa	743	**686**	529	11522	**9945**	8859	99.01	**91.86**	79.48	346	**241**	136
asg_taewa_6_	2	asg	taewa	1416	**1289**	1122	10103	**9774**	9450	55.60	**53.62**	51.64	2021	**1875**	1729
k8s_taewa2_3_	13	k8s	taewa2	467	**363**	251	5818	**5218**	4113	57.79	**56.36**	54.69	376	**167**	14

For individual experiment run results and graphs please refer to the Results Explorer https://taro-ball.github.io/thesis-results/ .

For raw data please refer to the results repository: https://github.com/taro-ball/thesis-results .

APPENDIX E

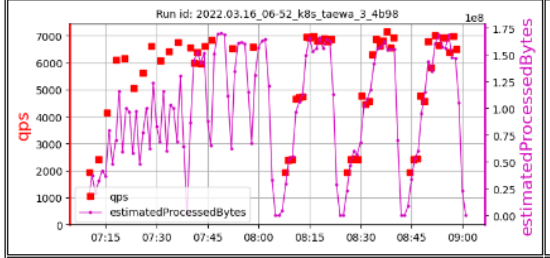
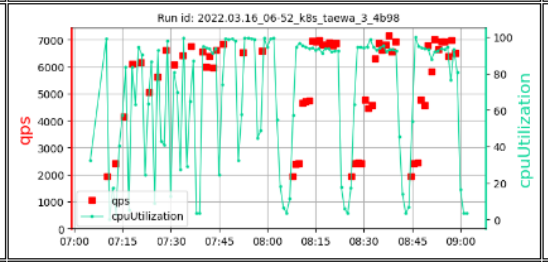
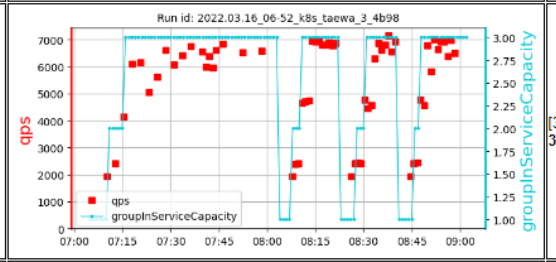
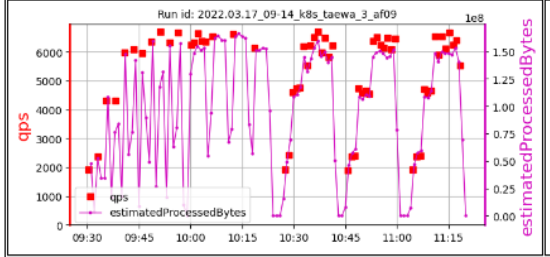
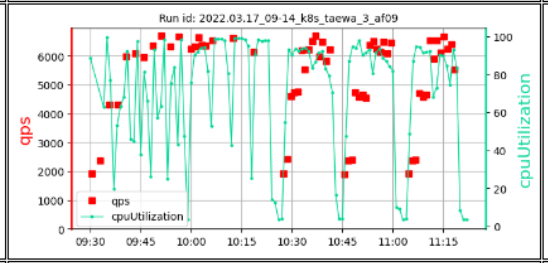
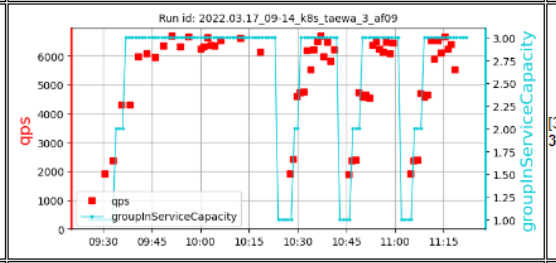
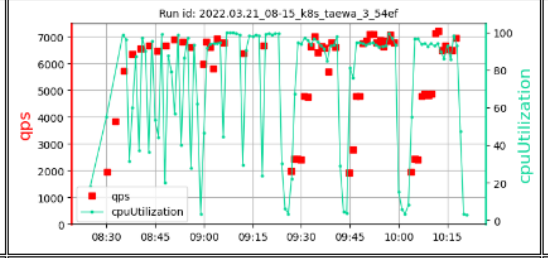
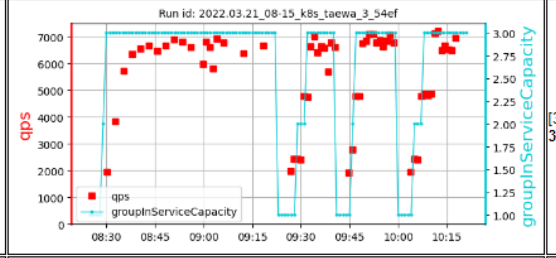
Software package and AMI image versions

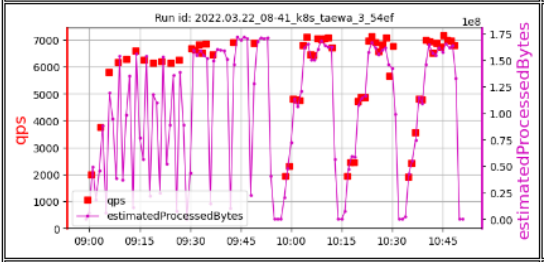
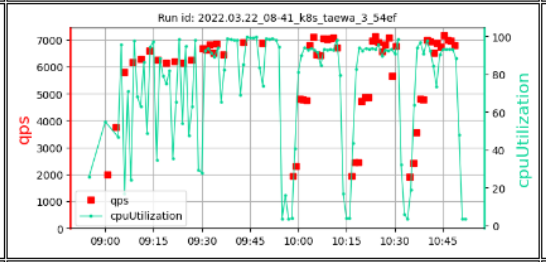
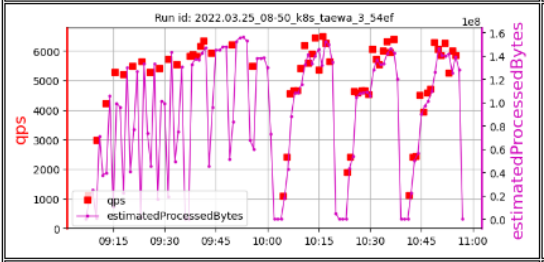
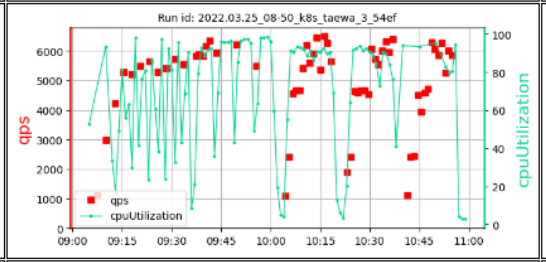
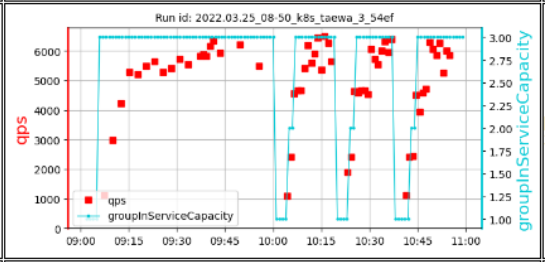
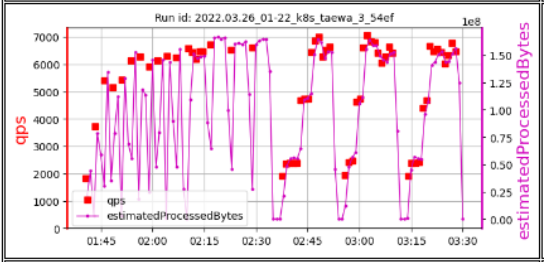
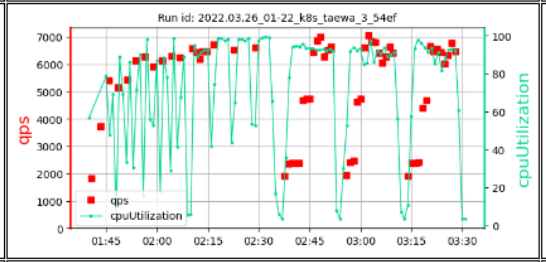
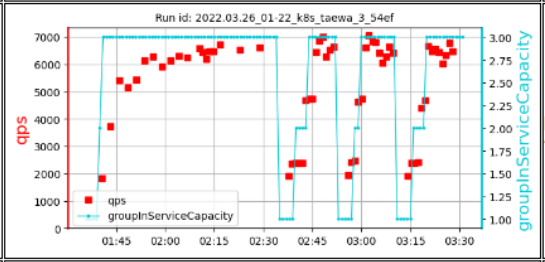
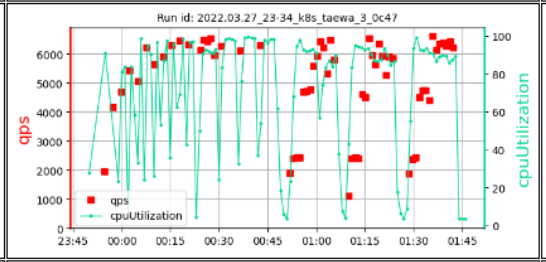
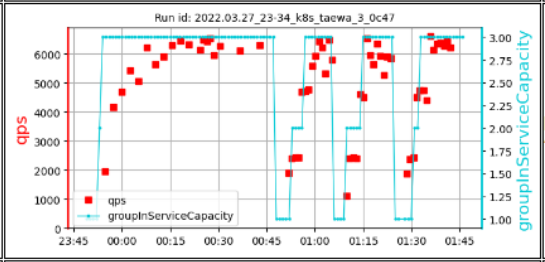
Application	AWS Linux 2 (ASG and jump host)	Docker Container	Windows (developer laptop)
Apache	httpd-2.4.51-1.amzn2.x86_64 (RPM pre-packaged)	httpd:2.4.51 (official image from Docker Hub)	n/a
Node.js	16.13.1 (via NVM)	node:16.13.1-slim (official image from Docker Hub)	n/a
Python	python3-3.7.10-1.amzn2.0.1.x86_64 (RPM pre-installed)	python:3.7.10-alpine (official image from Docker Hub)	n/a
AWS CLI	1.18.147 (RPM pre-installed)	n/a	2.4.7 (official website)
eksctl	0.83.0 (official GitHub)	n/a	0.83.0 (official GitHub)
kubectl	1.21.2 (AWS EKS repository)	n/a	1.22.4 (official website)
Fortio	1.19.0 (official GitHub)	n/a	n/a
docker	n/a	n/a	20.10.11

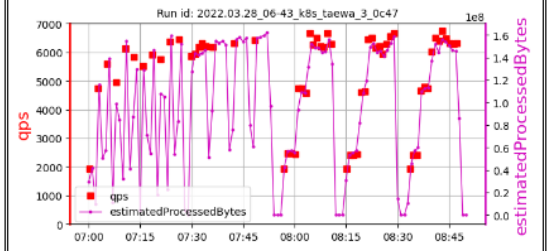
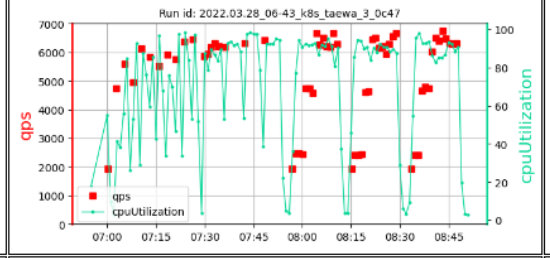
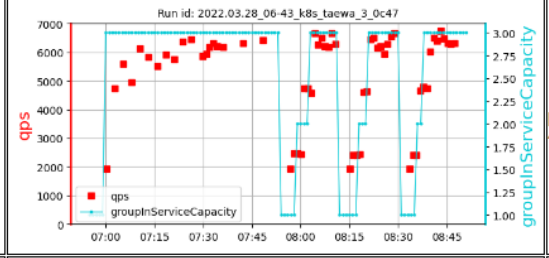
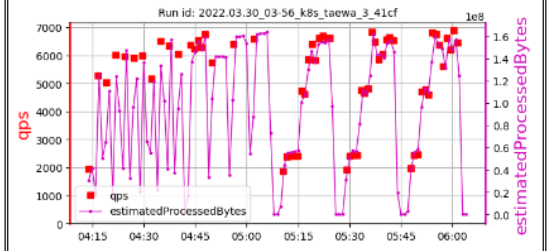
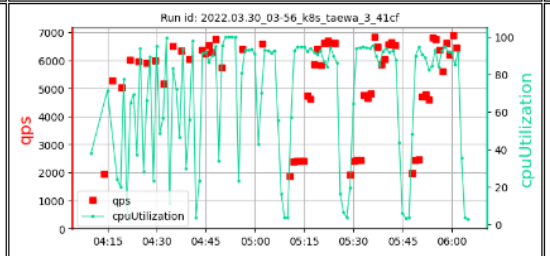
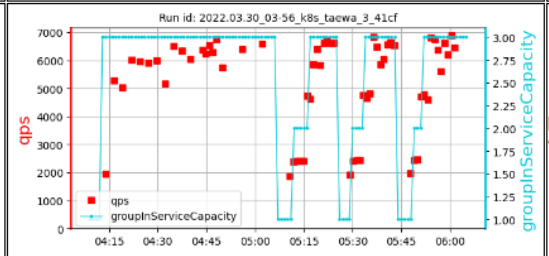
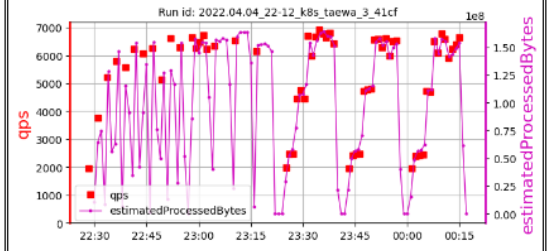
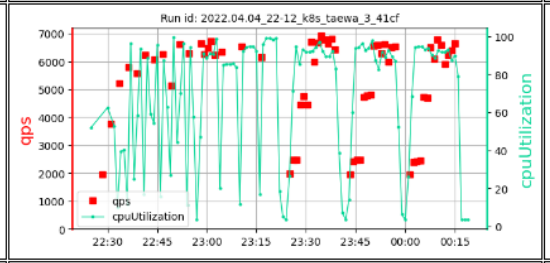
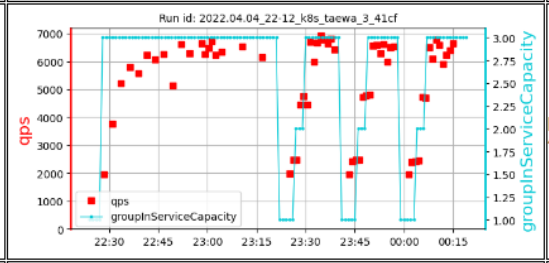
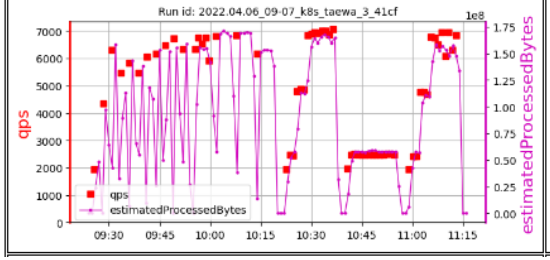
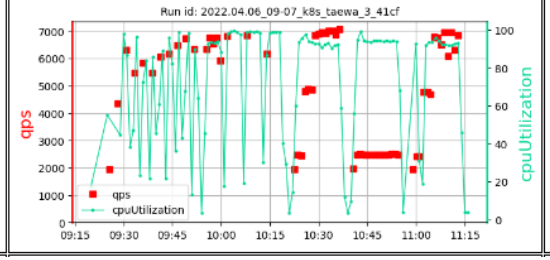
	ASG instances	EKS node instances	Jump host instance
AMI image	/aws/service/ami-amazon-linux-latest/amzn2-ami-hvm-x86_64-gp2	/aws/service/eks/optimized-ami/1.21/amazon-linux-2/recommended/image_id	/aws/service/ami-amazon-linux-latest/amzn2-ami-hvm-x86_64-ebs

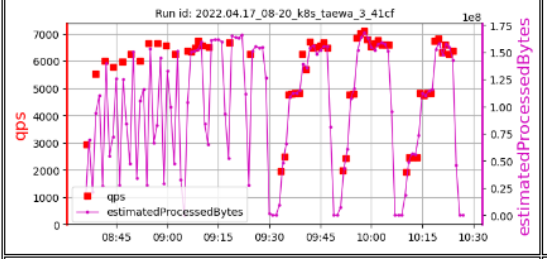
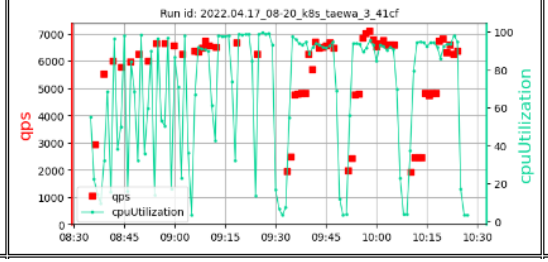
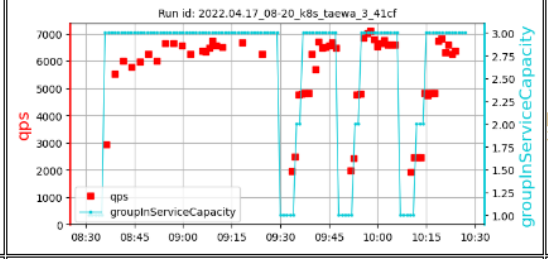
APPENDIX F

A sample view of the Results Explorer

k8s_taewa_3_ samples: **12** date generated: 2022-06-26 20:58:03.138054		**Overall:**	max: 504 min: 267 mean: **388**	max: 6900 min: 5485 mean: **6428**	max: 99.76 min: 85.46 mean: **97.32**	max: 270 min: 79 mean: **185**
1. test id: **2022.03.16_06-52_k8s_taewa_3_4b98** [raw data]	run settings: cpu=70 pods=6 treads=25 sc_min=14 warmup=90		**Duration s**	**qps**	**CPU %**	**Err**
			[389, 447, 319] **385**	[6839, 6517, 6575] **6643**	[98.52, 99.17, 97.52] **98.40**	**92**
2. test id: **2022.03.17_09-14_k8s_taewa_3_af09** [raw data]	run settings: cpu=70 pods=6 treads=25 sc_min=14 warmup=90		Duration s	qps	CPU %	Err
			[300, 435, 396] **377**	[6535, 6613, 6147] **6432**	[98.18, 98.74, 97.84] **98.25**	**79**
3. test id: **2022.03.21_08-15_k8s_taewa_3_54ef** [raw data]	run settings: cpu=70 pods=6 treads=25 sc_min=14 warmup=90		Duration s	qps	CPU %	Err
			[374, 267, 472] **371**	[6781, 6381, 6671] **6611**	[99.76, 98.31, 99.33] **99.13**	**145**

			Duration s	qps	CPU %	Err
4. test id: **2022.03.22_08-41_k8s_taewa_3_54ef** [raw data]	run settings: cpu=70 pods=6 treads=25 sc_min=14 warmup=90		[326, 336, 288] **317**	[6455, 6900, 6869] **6741**	[98.44, 99.48, 98.79] **98.90**	**270**
						
5. test id: **2022.03.25_08-50_k8s_taewa_3_54ef** [raw data]	run settings: cpu=70 pods=6 treads=25 sc_min=14 warmup=90		[280, 443, 457] **393**	[5933, 6218, 5485] **5879**	[95.9, 96.96, 98.32] **97.06**	**144**
						
6. test id: **2022.03.26_01-22_k8s_taewa_3_54ef** [raw data]	run settings: cpu=70 pods=6 treads=25 sc_min=14 warmup=90		[481, 376, 383] **413**	[6712, 6523, 6617] **6617**	[98.13, 97.81, 98.5] **98.15**	**236**
						
7. test id: **2022.03.27_23-34_k8s_taewa_3_0c47** [raw data]	run settings: cpu=70 pods=6 treads=25 sc_min=14 warmup=90		[390, 405, 446] **414**	[6281, 6123, 6306] **6237**	[98.3, 99.06, 97.49] **98.28**	**268**
						

			Duration s	qps	CPU %	Err
8. test id: **2022.03.28_06-43_k8s_taewa_3_0c47** [raw data]	run settings: cpu=70 pods=6 treads=25 sc_min=14 warmup=90		[463, 411, 349] **408**	[6181, 6323, 6424] **6309**	[92.65, 97.9, 93.32] **94.62**	**211**
						
9. test id: **2022.03.30_03-56_k8s_taewa_3_41cf** [raw data]	run settings: cpu=70 pods=6 treads=25 sc_min=14 warmup=90		[436, 431, 348] **405**	[5732, 6408, 6579] **6240**	[99.75, 92.31, 92.19] **94.75**	**143**
						
10. test id: **2022.04.04_22-12_k8s_taewa_3_41cf** [raw data]	run settings: cpu=70 pods=6 treads=25 sc_min=14 warmup=90		[375, 450, 405] **410**	[6333, 6549, 6152] **6345**	[85.46, 94.62, 98.81] **92.96**	**241**
						
11. test id: **2022.04.06_09-07_k8s_taewa_3_41cf** [raw data]	run settings: cpu=70 pods=6 treads=25 sc_min=14 warmup=90		[353, 376] **365**	[6816, 6834, 6161] **6603**	[98.92, 98.92, 98.69] **98.84**	**260**
						

12. test id: **2022.04.17_08-20_k8s_taewa_3_41cf** [raw data]	run settings: cpu=70 pods=6 treads=25 sc_min=14 warmup=90		Duration s	qps	CPU %	Err
			[385, 279, 504] **390**	[6516, 6668, 6254] **6479**	[97.78, 98.66, 99.18] **98.54**	**132**

Note: This is just a sample. For full results explorer view please visit https://taro-ball.github.io/thesis-results/

APPENDIX G

Load testing script (thesis-infra/data/cli/1.load.sh)

```
#!/usr/bin/bash
set -x
mydir=`dirname "$0"`
cd $mydir
test=$(cat mytest)
type=$(cut -d "_" -f 1 <<< $test)
exec >> load-${type}.log
exec 2>&1
# exec 2> load-errors.log # also sends all fortio output to err log
git log -3
app=$(cut -d "_" -f 2 <<< $test)
export AWS_DEFAULT_REGION="us-east-1"
line='=============================='

echo [$(date +%FT%T)]${line}[starting in $PWD]${line}

source 0.setup.sh
source .k8sSecrets.noupl
aws sts get-caller-identity

############# Functions #############
check_stats () {
  echo [$(date +%FT%T)]${line}[STATS]
  echo jh CPU load avg:
  cat /proc/loadavg
  if [ "$1" == "asg" ]; then
    aws autoscaling describe-auto-scaling-groups --query 'AutoScalingGroups[*]' | \
    jq --raw-output '.[]| .Instances[] |
(.InstanceId, .LifecycleState, .HealthStatus, {})'
  fi
  if [ "$1" == "k8s" ]; then
    kubectl get hpa
    kubectl get deployment
    kubectl top nodes 2> /dev/null
    #make sure all nodes have monitoring on
    aws ec2 monitor-instances --instance-ids `aws autoscaling describe-auto-scaling-
groups --query 'AutoScalingGroups[*].Instances[*].InstanceId' --output text` --output
text | grep enabled # > /dev/null
  fi
}

############# Ready check #############
while [ -z "$mycheck" ]
do
  if [ "$type" == "asg" ]; then
    mycheck=`aws autoscaling describe-auto-scaling-groups --query
'AutoScalingGroups[*].AutoScalingGroupName' --output text | sed 's/\s\+/\n/g' | grep
myASG`
  fi
  if [ "$type" == "k8s" ]; then
    mycheck=`aws autoscaling describe-auto-scaling-groups --query
'AutoScalingGroups[*].AutoScalingGroupName' --output text | sed 's/\s\+/\n/g' | grep
workers`
  fi
  echo [$(date +%FT%T)] waiting for instances ...
  sleep 60;
done
echo export t_start=$(date +%FT%T) >> metrics_vars.txt
myasg=`aws autoscaling describe-auto-scaling-groups --query
'AutoScalingGroups[*].AutoScalingGroupName' --output text`
```

```
if [ "$type" == "k8s" ]; then
  # enable advanced ec2 metrics in the lt (have to apply new ver in console
mannually)
  template_name=`aws ec2 describe-launch-templates --query
'LaunchTemplates[*].LaunchTemplateName' --output text | sed 's/\s\+/\n/g' | grep eks-
`
  aws ec2 create-launch-template-version  --launch-template-name ${template_name} --
version-description EnableAdvMonitoring --source-version 1 --launch-template-data
'{"Monitoring": {"Enabled": true}}'
  aws ec2 modify-launch-template --launch-template-name ${template_name} --default-
version 2
  # aws autoscaling update-auto-scaling-group \ #(disabled auto apply as it doesn't
seem to work, see 6 March notes)
  #   --auto-scaling-group-name $myasg \
  #   --launch-template LaunchTemplateName=${template_name},Version='$Latest'
  ## refresh asg to apply the lt
  # aws autoscaling start-instance-refresh --auto-scaling-group-name $myasg
  # enable worker ASG metrics
  aws autoscaling enable-metrics-collection --auto-scaling-group-name ${myasg} --
granularity "1Minute"
fi
# wait for stack to stabilise
sleep 180
################ Prepare to run the load test ################
if [ "$type" == "asg" ]; then
  # get lb, asg and policy
  lb_dns=`aws elb describe-load-balancers --query
'LoadBalancerDescriptions[*].DNSName' --output text | sed 's/\s\+/\n/g' | grep asg`
  mypolicy_name=`aws autoscaling describe-policies --query
"ScalingPolicies[*].PolicyName" --output text | sed 's/\s\+/\n/g' | grep asg`
  policy_json='{ "PredefinedMetricSpecification": { "PredefinedMetricType":
"ASGAverageCPUUtilization" }, "TargetValue":'" ${cpu_perc}.0, "'"DisableScaleIn":
false}'
  # set max, scale to max
  echo scaling to $max_capacity;
  aws autoscaling update-auto-scaling-group --auto-scaling-group-name ${myasg} --
desired-capacity $max_capacity --max-size $max_capacity
  # fix alb healthcheck
  myalb=`aws elb describe-load-balancers --query
'LoadBalancerDescriptions[*].LoadBalancerName' --output text`
  aws elb configure-health-check --load-balancer-name ${myalb} --health-check
Target=HTTP:$warmup_url,Interval=10,UnhealthyThreshold=6,HealthyThreshold=2,Timeout=5
fi
if [ "$type" == "k8s" ]; then
  # log on to k8s
  aws eks update-kubeconfig --name $cluster_name
  kubectl get svc # quick check
  # get lb
  lb_dns=`kubectl get svc/${app}-svc -o json | jq --raw-output
'.status.loadBalancer.ingress[0].hostname'`

  # start k8s metrics collection
  nohup ./k8s-metrics.sh&

  # scale k8s nodes to max
  eksctl scale nodegroup --cluster=$cluster_name --name=standard-workers --
nodes=$max_nodes --nodes-max=$max_nodes

  # enable hpa
  kubectl autoscale deployment ${app}-deployment --cpu-percent=$hpa_perc --min=1 --
max=$max_pods
fi
# wait for healthecks
sleep 60
# quick test
curl http://${lb_dns}:${warmup_url}; echo
```

```
############### LB warmup run ###############
for((i=$warmup_min_threads;i<=$warmup_max_threads;i+=1));
do
    check_stats $type
    echo [$(date +%FT%T)]${line}[WARMUP RUN c${i}]${line}
    fortio load ${fortio_options} -c $i -t ${warmup_cycle_sec}s -labels "${test}-
warmup-${i}" http://${lb_dns}:${warmup_url}
    echo fortio exit code: $?
    check_stats $type
    sleep 60
done

aws elb describe-load-balancers
aws autoscaling describe-auto-scaling-groups
aws ec2 describe-instances  --query 'Reservations[*].Instances[*]' >
instance.dump.json # save instance state after warmup, too much to keep in log

############### Control run ###############
echo [$(date +%FT%T)]${line}[PERFORMANCE CHUNK RUN ${i}]${line}
sleep 60
    for((x=1;x<=5;x+=1));
    do
      check_stats $type
      fortio load -quiet ${fortio_options} -c $warmup_max_threads -t 60s -labels
"${test}-performance-chunk-${i}-${x}" http://${lb_dns}:${testing_url}
      echo fortio exit code: $?
    # check_stats $type
    done

############### Performance run ###############
for((i=1;i<=3;i+=1));
do
    sleep 60
    check_stats $type
    echo [$(date +%FT%T)]${line}[PERFORMANCE RUN ${i}]${line}
    fortio load ${fortio_options} -c $warmup_max_threads -t ${performance_sec}s -
labels "${test}-performance-${i}" http://${lb_dns}:${testing_url}
    echo fortio exit code: $?
    check_stats $type
done

echo export t_scaling=$(date +%FT%T) >> metrics_vars.txt

# flush instances to avoid running out of space
if [ "$type" == "asg" ]; then
  echo scaling to 0;
  # 99cpu policy to prevent immideate scaleout on historical data
  aws autoscaling put-scaling-policy --auto-scaling-group-name ${myasg} --policy-name
$mypolicy_name --policy-type TargetTrackingScaling --target-tracking-configuration
'{ "PredefinedMetricSpecification": { "PredefinedMetricType":
"ASGAverageCPUUtilization" }, "TargetValue": 99.0, "DisableScaleIn": false}'
  aws autoscaling update-auto-scaling-group --auto-scaling-group-name ${myasg} --min-
size 0 --desired-capacity 0;
  sleep 30
fi
############### Scaling run ###############
for((i=1;i<=3;i+=1));
do
    echo [$(date +%FT%T)]${line}[SCALING RUN $i: INIT]${line}
    if [ "$type" == "asg" ]; then
      # 99cpu policy to prevent immideate scaleout on historical data
      aws autoscaling put-scaling-policy --auto-scaling-group-name ${myasg} --policy-
name $mypolicy_name --policy-type TargetTrackingScaling --target-tracking-
configuration '{ "PredefinedMetricSpecification": { "PredefinedMetricType":
"ASGAverageCPUUtilization" }, "TargetValue": 99.0, "DisableScaleIn": false}'
      # scale to min
      echo scaling to 1;
```

```
      aws autoscaling update-auto-scaling-group --auto-scaling-group-name ${myasg} --
desired-capacity 1;
      echo [$(date +%FT%T)]${line}[SCALING RUN $i: ENABLE SCALING - SLEEP]${line}
      sleep 160;
      # back to initial policy
      aws autoscaling put-scaling-policy --auto-scaling-group-name ${myasg} --policy-
name $mypolicy_name --policy-type TargetTrackingScaling --target-tracking-
configuration "${policy_json}"
    fi
    if [ "$type" == "k8s" ]; then
      # delete hpa to prevent immediate scaleout on historical data
      kubectl delete horizontalpodautoscaler.autoscaling/${app}-deployment

      # scale to min
      echo scaling to 1;
      kubectl scale --replicas=1 deployment/${app}-deployment
      eksctl scale nodegroup --cluster=$cluster_name --name=standard-workers --
nodes=1
      echo [$(date +%FT%T)]${line}[SCALING RUN $i: ENABLE SCALING - SLEEP]${line}
      sleep 160;
      # create the hpa
      kubectl autoscale deployment ${app}-deployment --cpu-percent=$hpa_perc --min=1
--max=$max_pods

    fi

    sleep 20 # let hpa stabilise (collect initial metrics)

    echo [$(date +%FT%T)]${line}[SCALING RUN ${i}: FORTIO]${line}
    for((y=1;y<=$scaling_minutes;y+=1));
    do
      check_stats $type
      echo [$(date +%FT%T)]${line}[SCALING RUN ${i}: CHUNK $y]${line}
      fortio load -quiet ${fortio_options} -c $warmup_max_threads -t 60s -labels
"${test}-scaling-${i}-${y}" http://${lb_dns}:${testing_url}
      echo fortio exit code: $?
    done
    check_stats $type
done

# wait for CloudWatch logs to catch up
sleep 240

# save vars for metric query
myalb=`aws elb describe-load-balancers --query
'LoadBalancerDescriptions[*].LoadBalancerName' --output text` # repeat here as in the
beginning k8s alb isn't yet ready
echo export asg_name=${myasg} >> metrics_vars.txt
echo export lb_name=${myalb} >> metrics_vars.txt
echo export t_end=$(date +%FT%T) >> metrics_vars.txt

echo [$(date +%FT%T)]${line}[GET DATA]${line}
./2.jh-get-data.sh
echo [$(date +%FT%T)]${line}[UPLOAD $(cat 3.upload.noupl.sh | egrep -o
'/2022\S+')]${line}
# in case logs grow big, compress them
find . -maxdepth 1 -type f -size +500k -exec zip -m9 backup {} \;
./3.upload.noupl.sh
```

APPENDIX H

Results Explorer script (thesis-infra/data/processing/resultsExplorer.ipynb)

```
from pathlib import Path
from IPython.display import HTML, display
import ipywidgets as widgets
import datetime
import pandas as pd
import numpy as np
import re
github_repo="https://github.com/taro-ball/thesis-results/"
base_path = Path.cwd().parent.parent.parent / "thesis-results"
p = Path(base_path)

tsts = ["k8s_taewa_3lite_","asg_taewa_3lite_", "k8s_apache_3_", "k8s_taewa_3_",
"asg_apache_3_", "asg_taewa_3_"]
tsts = tsts + ["asg_taewa_3extra_", "k8s_taewa_3extra_", "asg_raupi_3x_",
"k8s_raupi_3_", "k8s_raupi2_3_", "asg_raupi_3","k8s_taewa_6_", "asg_taewa_6_"]
tsts = tsts + ["asg_riwai_3_", "k8s_riwai_3_", "k8s_taewa2_3_", "k8s_apache2_3new",
"k8s_apache2_3_", "asg_raupi_3_70_"]
test = widgets.Dropdown(
       options=tsts,
       value='k8s_apache_3_',
       description='Test:')
all = widgets.Checkbox(value=True,
       description='All')
graphs=["QPS_estimatedProcessedBytes.png","QPS_cpuUtilization.png","QPS_groupInServ
iceCapacity.png"]
date_filter = widgets.Text(
     value='2022.0[3-9]', # use glob pattern http://pymotw.com/2/glob/
     description='Filter', )
box = widgets.HBox([date_filter, all, test ])
display(box)

#print(test._options_values)
tests=[]
if all.value:
    tests=test._options_values;
else:
    tests.append(test.value)
csv_path=p/'o_summary.csv'

# backup old csv in case processing fails
if  csv_path.exists():
    #Path(csv_path).unlink()
    modified=datetime.datetime.fromtimestamp(Path(csv_path).stat().st_mtime).strfti
me('%m_%d_%Y_%H_%M_%S')
    csv_path.rename(p / ('o_summary.csv.'+modified ))

# loop through specs
for tst in tests:
    print('\n','='*40,f'\nprocessing {tst}\n')
    search=f'*{date_filter.value}*{tst}*'
    nw = datetime.datetime.now()
    count=0
    iList=[]
    total_err_list=[]
    all_qps_list=[]
    all_cpu_list=[]
    all_duration_list=[]
    all_err_list=[]

    # loop through each test run
    for i in p.glob(search):
```

```
        print(i)
        count+=1
        dir = i.name
        platform=dir.split("_")[2]
        if not (i/"csv").exists():
            line = f"<tr><td rowspan='1'><b>No data found, skip:
{dir}</b></td></tr>"
            iList.append(line)
            continue

        #================ read test settings from the log file
        log_path = list(i.glob('load*.log'))[0]
        max_pods=0
        with open(log_path, "r") as file:  # the a opens it in append mode
            for l in range(100):
                statement = next(file).strip()
                if m := re.match(".*scaling_minutes=(\d+).*", statement):
                    scaling_minutes=int(m.group(1))
                elif m := re.match(".*warmup_max_threads=(\d+).*", statement):
                    max_threads=int(m.group(1))
                elif m := re.match(".*max_pods=(\d+).*", statement):
                    max_pods=int(m.group(1))
                elif m := re.match(".*_perc=(\d+).*", statement):
                    cpu_target=int(m.group(1))
                elif m := re.match(".*warmup_cycle_sec=(\d+).*", statement):
                    warmup_seconds=int(m.group(1))

        ##================ get json Fortio data ================
        json_paths = (i/"csv").glob('*.json')
        fortio_path=list(json_paths)[0]

        df = pd.read_json(fortio_path, lines=True)
        df['StartTime'] = pd.to_datetime(df['StartTime'])
        df.set_index(['StartTime'],inplace=True)

        # get performance QPS
        perf_df = df[df['Labels'].str.contains('performance-[1-3]', regex=True)]
        qps_lst=list(perf_df.ActualQPS)
        avg_qps=np.mean(qps_lst)
        all_qps_list.extend(qps_lst)

        # get scaling start times
        scaling_starts_df = df[df['Labels'].str.contains('scaling-.-1$',
regex=True)]

        aws_metric0="backendConnectionErrors"
        c_path_err=list((i/"csv").glob('*'+aws_metric0+'.csv'))[0]
        err_df = pd.read_csv(c_path_err, parse_dates=['datetime'],
index_col="datetime")

        ## ================ PERFORMANCE CloudWatch data ================
        aws_metric1="cpuUtilization"
        csv_paths = (i/"csv").glob('*'+aws_metric1+'.csv')
        c_path=list(csv_paths)[0]
        mdf = pd.read_csv(c_path, parse_dates=['datetime'], index_col="datetime")
        avg_perfomance_cpu_list=[]
        #print(c_path)
        for index,value in perf_df.iterrows():
            perf_start=index#pd.to_datetime(stime)
            #print(perf_start)
            # get 4 max:
            # mmdf=mdf[perf_start:perf_start+ pd.Timedelta(5, "m")]
            # mmmdf=mmdf.nlargest(4, aws_metric1).sort_index()

            # get 3 mid:
            mmmdf=mdf[perf_start+ pd.Timedelta(1, "m"):perf_start + pd.Timedelta(4,
"m")]
```

```
            cpu_list=list(mmmdf[aws_metric1])

            average_cpu=np.mean(cpu_list)
            avg_perfomance_cpu_list.append(average_cpu)
            #print(avg_perfomance_cpu_list)
            if value.Labels[-1] == '1':
                first_perf_start=index
                #print(value.Labels)
        all_cpu_list.extend(avg_perfomance_cpu_list)
        avg_cpu=np.mean(avg_perfomance_cpu_list)

        # get number of errors from the start of first performance till the end of
last scaling
        merr_df=err_df[first_perf_start:first_perf_start + pd.Timedelta(68, "m")]
        err_list=list(merr_df[aws_metric0])
        total_err_list.append(sum(err_list))
        #print(total_err_list)

        ##================ get json K8s data ================
        #print(i)

        max_replicas=99
        k8s_depl_path=i/"k8s-deploy-metrics.json"
        if platform=="k8s":
            k8s_df = pd.read_json(k8s_depl_path, lines=True)
            k8s_df['time'] =
pd.to_datetime(k8s_df['time'],format='%d-%m-%Y %H:%M:%S') #26-03-2022 08:44:30
            k8s_df.set_index(['time'],inplace=True)
            k8s_df=k8s_df.tz_localize(tz='UTC') # localize, Pandas hates tz-naive
timestamps
            max_replicas=max(k8s_df['replicas'])
        else:
            max_replicas=int(tst.split("_")[2][0])
            max_pods="NA"


        setup=f"cpu={cpu_target} pods={max_pods} treads={max_threads}
sc_min={scaling_minutes} warmup={warmup_seconds}"
        ## ================ get json CloudWatch data ================
        aws_metric2="groupInServiceCapacity"
        csv_paths2 = (i/"csv").glob('*'+aws_metric2+'.csv')
        c_path2=list(csv_paths2)[0]
        cwdf2 = pd.read_csv(c_path2, parse_dates=['datetime'],
index_col="datetime")

        ## ================ Process scaling data ================
        scaling_durations=[]
        #print(c_path2)
        for index,value in scaling_starts_df.iterrows():
            scale_start=index
            if platform=="k8s":
                kk8s_df=k8s_df[scale_start:scale_start+
pd.Timedelta(scaling_minutes, "m")]
                max_scaled=kk8s_df[kk8s_df['readyReplicas'].ge(max_replicas)]
            elif platform=="asg":
                ccwdf2=cwdf2[scale_start:scale_start+ pd.Timedelta(scaling_minutes,
"m")] # only need to look 14 min ahead as it's the length of the scaling run
                max_scaled=ccwdf2[ccwdf2[aws_metric2].ge(max_replicas)]

            if max_scaled.size > 0:
                cw_reach_max_time=max_scaled.index[0]
                duration = cw_reach_max_time - index
                duration_in_s = duration.total_seconds()
                scaling_durations.append(duration_in_s)
            else:
                duration_in_s = 0
```

```
        avg_duration=np.mean(scaling_durations)
        all_duration_list.extend(scaling_durations)

        if np.isnan(avg_duration): avg_duration=0
        #avg_duration_list.append(avg_duration)

        print(f"[{platform} ({max_replicas})] [ CPU:{avg_cpu} | Dur:{avg_duration}
| qps:{avg_qps} ]")

        ##================ generate rows ================
        line = ""
        accent = "<b>" if count==1 else ""
        github_path=f"{github_repo}tree/master/{dir}"
        line = f"""<tr><td rowspan='1'>{count}. test id: <b>{dir}</b> <a
href='{github_path}'>[raw data]</a></td><td>run settings: {setup}</td><td></td>
                <td>{accent}Duration
s</td><td>{accent}qps</td><td>{accent}CPU %</td><td>{accent}Err</td>
                </tr><tr>"""
        ####====== Images
        for y in graphs:
            #print(i.name)
            img=f'{i.name}/csv/{y}'
            line+=f"<td><img alt='{y}' src='{img}' /></td>"

        ####====== Data

        line+=f"<td>{list(map(round,scaling_durations))}<br><b>{avg_duration:.0f}</
b></td>"
        line+=f"<td>{list(map(round,qps_lst))}<br><b>{avg_qps:.0f}</td>"
        line+=f"<td>{[round(x,2) for x in
avg_perfomance_cpu_list]}<br><b>{avg_cpu:.2f}</b></td>"
        line+=f"<td><b>{total_err_list[-1]}</b></td>"
        line+="</tr>"
        iList.append(line)

    # fix durations list
    if not all_duration_list: all_duration_list=[0]

    total=f"""<tr><td>
        <h3>{tst} </h3>samples: <b>{count}</b><br>date generated:
{nw}<br></td><td></td><td
align="right"><h3>Overall:    </h3></td>
        <td>max:<br>{max(all_duration_list):.0f}<br>min:<br>{min(all_duration_list)
:.0f}<br>mean:<br><b>{np.mean(all_duration_list):.0f}</b><br></td>
        <td>max:<br>{max(all_qps_list):.0f}<br>min:<br>{min(all_qps_list):.0f}<br>m
ean:<br><b>{np.mean(all_qps_list):.0f}</b><br></td>
        <td>max:<br>{max(all_cpu_list):.2f}<br>min:<br>{min(all_cpu_list):.2f}<br>m
ean:<br><b>{np.mean(all_cpu_list):.2f}</b><br></td>
        <td>max:<br>{max(total_err_list):.0f}<br>min:<br>{min(total_err_list):.0f}<
br>mean:<br><b>{np.mean(total_err_list):.0f}</b><br></td></tr>"""
    iList.append(total)
    iList.insert(0,total)
    imagesList = ''.join(iList)

    ##================ save HTML ================

    html_path=p/f'o_{tst}.html'
    header=f"<!DOCTYPE html><html><head><title>{tst} - Result Explorer</title>"
    header+='<style>table, th, td {border: 1px solid black;font-size: 20px;} body
{font: 20px Arial, sans-serif;}</style></head><div style="color:red; font-weight:
bold">Please zoom out to see the full table.</div>'
    menuf=p/"o_menu.html"
    # the menu can be manually customized, will be generated if it does not exist
    if not menuf.exists():
    #================ generate menu
        menu=''
        def sorter(item):
```

```
            """sort by app and nodes"""
            x = ''.join(item.name.split("_")[2:4])
            return x
        menu+="<br><table><tr>"
        files = sorted(p.glob('*.html'), key=sorter)
        for h in files:
            test_name=f"{h.stem}"
            if (files.index(h)% 2) == 0:
                menu+=f"\n<td><a href='{h.name}'>{test_name}</a> "
            else:
                menu+=f"<br><a href='{h.name}'>{test_name}</a> </td>"
        menu+="</table>"
        print("saving menu, exists:",menuf.exists() )
        with open(menuf, 'w') as mf:
            mf.write(menu)
    ## external menu:
    mf = open(menuf, "r")
    header+=mf.read()
    mf.close()

    header+="<table>"
    footer="</table></html>"

    page= header + imagesList + footer
    a = HTML(page)
    html_src = a.data
    with open(html_path, 'w') as f:
        f.write(html_src)
    csv=""
    csv+=f'{tst},{count},{tst.split("_")[0]},{tst.split("_")[1]},'
    csv+=f'"{list(map(round,all_duration_list))}",{max(all_duration_list):.0f},{np.
mean(all_duration_list):.0f},{min(all_duration_list):.0f},'
    csv+=f'"{list(map(round,all_qps_list))}",{max(all_qps_list):.0f},{np.mean(all_q
ps_list):.0f},{min(all_qps_list):.0f},'
    csv+=f'"{all_cpu_list}",{max(all_cpu_list):.2f},{np.mean(all_cpu_list):.2f},{mi
n(all_cpu_list):.2f},'
    csv+=f'"{total_err_list}",{max(total_err_list):.0f},{np.mean(total_err_list):.0
f},{min(total_err_list):.0f},"{setup}"\n'

    with open(csv_path, "a") as csvfile:
        csvfile.write(csv)
```