Performance Evaluation of Container Runtimes on EKS, GKE & Manual K8S Deployment

Siddhartha Singh[1†], Rachit Jain[1†] and FNU Shivanshi[1†]

[1*]Courant, New York University, NYC, 10001, NY, USA.

Contributing authors: ss13793@nyu.edu; rj2219@nyu.edu; ss14396@nyu.edu;
[†]These authors contributed equally to this work.

Abstract

Containers are equivalent to any other workloads running on a cloud environment. The resources in a cloud environment are not limited. Hence, containers in a Kubernetes Cluster are always in contention for resources. Kubernetes has to manage the scheduling of pods on different nodes. Fitting as many containers on a node is a final state everyone aspires to reach so that optimal utilization of resources is done. This indirectly helps in operational and hardware costs. This paper evaluates the two most popular highe level container run time systems Docker and ContainerD. We study the impact these two container runtimes have over different areas such as CPU, I/O, Memory, and Network Latencys. The evaluation is done on Google Kubernetes Engine (GKE) and Amazon Elastic Kubernetes Service (EKS). We study the performance on both the cloud providers separately. This has helped the authors for a more rigorous and accurate analysis since the underlying hardware would be the same in that case. In the second part of the paper, the authors have deployed their own Kubernetes Cluster from scratch using Terraform and Kubeadm on Amazon Web Services and performed an analysis similar to those mentioned above compared to the Managed AWS Kubernetes offering. Through the two experiments, the authors want to determine which container runtime system is more performant in both the Managed Kubernetes Services. Secondly, whether deploying your own K8S cluster from scratch provides any significant performance gains or losses.

Keywords: Containers, Kubernetes, Docker, ContainerD, EKS, GKE, AWS, GCP, Sysbench, netperf, Stream

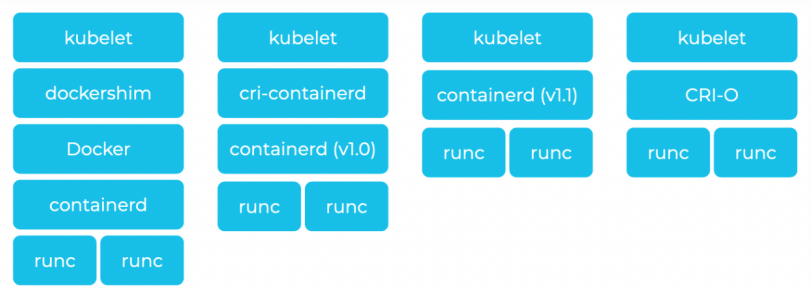
1 Introduction

Kubernetes [1] also known as K8S is an open-source container orchestration system that is used for automating the deployment, scaling, and management of containerized workloads. Development was inspired by its predecessor Google Borg [2] which is a cluster management tool used inside of Google and is used to manage most of the workloads being run by Google. Kubernetes has all the capabilities of Google Borg while it also enhances the system with much more rich features. Containers are at the heart of the Kubernetes Ecosystem and are the building blocks of the services built and managed by K8S. Therefore it is of the utmost importance that containers make judicious use of resources of the underlying hardware that Kubernetes Bootstraps so that maximum scalability and resource efficiency can be achieved. Run time systems which are an essential part of Kubernetes dictate how the image being deployed by a container is managed. Hence, run time systems are a crucial point of the system that can affect the performance of the containerized workloads. This calls for an in-depth evaluation of such run-time systems. There are two types of container runtime systems. First are the High-Level runtime systems, and the second is low-level run time systems. The authors in this paper stick to the performance evaluation of High-Level runtime systems. Docker [3] and ContainerD [4] are the two most popular run-time systems being offered by the various managed k8s cloud providers. In this paper, the authors have made use of Managed Kubernetes Service offered by Amazon AWS called the Elastic Kubernetes Service and the Managed Kubernetes Service by Google Cloud called the Google Kubernetes Engine for the performance analysis. Both the Managed Kubernetes Services provide the option to deploy the infra with ContainerD as the default runtime or Docker as the default runtime.

Docker [5] was a monolithic system since the beginning and it provided various tooling for logging, storage, networking and etc that are functionalities out of just creating containers. Docker later drained down these separate components and one of them was ContainerD; which is the container runtime component. Originally Kubernetes kept Docker as the default abstraction layer over Containerd. Containerd was the abstraction over the lower-level runtime runc. But a lot of additional components that Docker has created maintenance issues and prove to have significant overheads. Also, they are prone to a larger number of surface exploits. With time a lot of new container runtimes were introduced. Kubernetes thus came up with a concept of Container Runtime Interface that acted as a common interface for kubelet to talk with the container runtime system. Docker wasn't compliant with this hence it had to introduce an additional layer called the shim to operate. Soon enough Kubernetes and Containerd developers released a shim that helped kubelet talk directly to containerd and bypass the extra abstractions of Docker. This obviously reduced container capabilities but it was anyways useless to Kubernetes. In 2016, CRI-O was developed as an alternative to Docker. It jumped ahead of containerd's evolution to include a native CRI from the beginning.

In this way, the kubelet talks directly to CRI-O via the CRI to pull an image and launch the lower-level runtime (e.g., runc), which in turn sets up the namespaces, cgroups, root file system, storage, several Linux security modules and conmon, a CRI-O specific monitoring tool. One important difference between CRI-O and containerd was the removal of some Linux capabilities

There is a dearth of papers that compare Docker and Containerd; runtimes being provided by the two largest cloud providers. The authors realised that these are the only two high level runtime options available to users when deploying a managed K8S cluster on Google and AWS. Unavailability of research material about performance impacts of the two container runtimes before deploying your production apps on such public cloud providers is not a good situation to be in.



Different container runtimes

2 Related Work

In [6] the authors have compared the containerd and CRI-O container runtime systems. They compared containerD and CRI-O on two different container runtime initiatives runc and gVisor. Runc is the default low level container run time system while gVisor is the more secure alternative. They evaluated the run time aspects of a running container and scalability aspects. The authors have not included the analysis of Docker container run time system. Also, they lay focus on the affects of Runc and gVisor and not the higher level run time systems. The evaluation findings revealed that the crio/runc combination is the best starting point for almost any use-case, whereas containerd/runc benefits for I/O-intensive workloads. Runtime configurations such as runsc may provide superior security, but when performance is critical, their use must be carefully examined.

In [7] the authors which are The Aristotle Cloud Federation undertook the task of better understanding different challenges that one faces in choosing

various runtimes for scientific purpose. These were Docker, Singularity, and X-Containers. The main crux of the paper was to identify the "pain points" when using containers in Scientific Research. They included their experiences with Kubernetes and container orchestration on cloud and HPC systems. However, they failed to compare the higher level runtimes and lower level runtimes. But again, the research fails to highlight the performance impact of choosing different high level runtimes systems on the applications being deployed.

3 Evaluation Methodology

Cloud services are ever growing and with it, the need to evaluate services over the cloud is also becoming increasingly important. There are multiple cloud service providers who are providing various kinds of services. The variety in these services makes it necessary to evaluate performance of these services and draw comparisons between various kinds of services.

3.1 Problem Identification

Every cloud service provider is providing many Kubernetes services which can be managed or deployed manually. The two container runtimes which are famous are containerd and docker. The areas that the authors identified for evaluating performance are:

- By choosing between two different high level container runtimes, are there some overheads inferred due to these container runtimes?
- Will a self deployed Kubernetes cluster perform better as compared to a managed Kubernetes cluster?
- Does a container runtime affect performance in certain scenarios such as if the workload is memory heavy or if the workload is compute heavy?
- How to benchmark performance over various parameters?

An important aspect of this evaluation is that the authors are not comparing between different cloud services as the underlying hardware can differ and it might be difficult to draw comparisons between the cloud providers. The authors aim to compare parameters on the same cloud provider platform. Since Kubernetes as a service is not offered by all cloud service providers, the authors choose Elastic Kubernetes Service (EKS) by Amazon and Google Kubernetes Engine (GKE) by Google for the comparisons.

3.2 Resources and Benchmarks Selection

The authors considered resources based on the identified problems in the previous part: CPU, Disk, Memory, and Network. Comparison on these resources can give us a good idea since there can be certain container runtimes which are optimized for a certain type of resource. The metrics that were considered

in this are mentioned in Table 1. The authors have also mentioned the benchmarking tools that were used in the table. All the tools used are open source and are used for stressing the resource and evaluating performance.

Table 1 Resources and benchmarks

Resources	Metrics	Benchmarks
CPU	CPU Load	sysbench
Disk	Data throughput (Random I/O)	sysbench
Memory	Data throughput	STREAM
Network	TCP latency	netperf

3.2.1 Sysbench benchmark

For benchmarking of the CPU and Disk resource, the authors have used the Sysbench [8] which is an open source tool which is used for benchmarking of CPU Load and Disk I/O. This benchmark can define the number of worker threads to be used, making it multithreaded. The benchmark itself has very little overhead and can generate extensive statistics while capturing millions of events.

- For CPU load, it stresses the CPU with heavy operations such as calculation of prime numbers. The number of worker threads defined for the experiment were 2 and the prime number calculation was done upto 30000. The metric that was captured from this benchmark was number of events per second.
- For Disk I/O, the benchmark stresses the disk with intensive read and write operations. Sysbench benchmarks the disk's random read and write operations. Random read and write operations are more commonly found in disks when compared to sequential read and write operations. It creates a file and stores it in the disk and then accesses it randomly. To bypass the caching mechanism performed by the RAM of the system, the file size needs to be larger than the memory size so that the file cannot be cached in the memory. The persistent volume claim defined for the experiment was of 10 GB and the file size chosen was 9 GB with a total of 2 worker threads defined. The test files first had to be prepared which entailed creation of the files on the disk. Then the read and write operations were performed on the files by Sysbench. The files were finally removed from the system once the tests were completed. The metric that was captured for this resource was the read and writes in MB/seconds.

3.2.2 STREAM benchmark

Performance of the system memory can be tested using the STREAM [9] benchmark. This benchmark is open source and is developed by University of

Virginia and is the de facto standard for benchmarking memory. It uses vector operations of copy, add, scale, and triad as mentioned in Table 2.

Table 2 STREAM operations

Operation	Kernel
Copy	x[i] = y[i]
Add	x[i] = y[i] + z[i]
Scale	x[i] = scalar * y[i]
Triad	x[i] = scalar * y[i] + z[i]

- The copy operation is the copying of one vector into another vector.
- The add operation is the addition of two vectors and storing it in another vector.
- The scale operation is the multiplication of a vector with a scalar and storing it in a vector.
- The triad operation is the most expensive operation in which one scaled vector is added to a vector and the result is stored in another vector.

This benchmark is an important test since the CPU computation of systems is increasing but the memory bandwidth limits the capabilities. With this benchmark, the authors are measuring how quickly the data can be read from and written to the main memory. For the experiment, the authors measured the data throughput capacity of the main memory which is measured in MB/seconds. The number of elements in the vectors need to be larger than 4 times the size of the largest cache for getting efficient results. Number of elements the authors chose for their experiments are 10000000.

3.2.3 Netperf benchmark

Network evaluation was evaluated using the netperf benchmarking tool which is available on the Kubernetes GitHub page. The aim for this benchmark is to evaluate network performance in a Kubernetes cluster when there are multiple worker nodes and protocol stacks being used. The network in the cluster is stressed by transmitting memory buffers between a client and a server and then checking the round trip network latency which is measured in MB/seconds. The test works by creating a total of three worker pods out of which two worker pods are placed on the same cluster node and one is placed on another cluster node. This test follows a client server architecture in which one worker pod is chosen as a netperf client and one worker pod is chosen as a netperf server. For our experiments, we considered two cases of network latencies:

- **When the client and server are both on the same cluster node**: In this type of experiment, we are measuring the network latency between two pods on the same cluster node.
- **When the client and server are on different cluster nodes**: In this type of experiment, we are measuring the network latency between two pods on different cluster nodes.

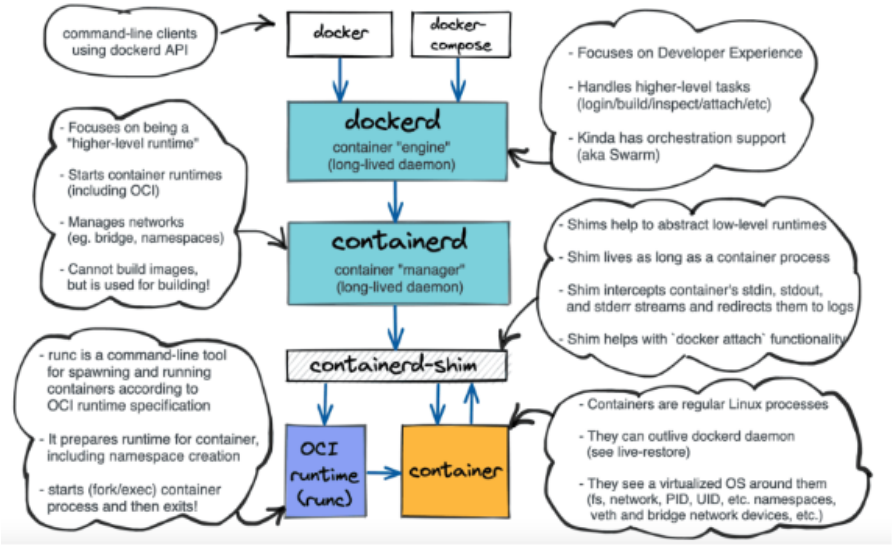
The client which is a netperf pod sends a request to the server which is a netperf pod and waits for an acknowledgement from the server before transmitting the next packet. By doing so, all the packets are sent independently and do not interfere with each other. For the author's experiments, a packet of size 300 bytes was sent from the client and the server.

4 Evaluated Container Runtimes

4.1 Docker

It was the year 2013 when Docker was released. Docker provided developers a lot of tools that helped developers manage the state of containerised applications end-to-end. It helped developers to create images, and to manage them as well. It also allowed developers to manage the instance of containers and share container images as well (via docker push and docker pull). Developers could run their containers with a simple docker run command. Although Docker was a monolithic system none of the subsystems were actually dependent on each other and due to this reason Docker, Google and other vendors introduced the Open Container Initiative (OCI). At first, it was unclear what Docker had contributed to OCI. They supplied little more than a standard technique to "run" containers. The picture format and registry push/pull formats were left out. Only running the container was standardized by Docker. Until that point was clear, everyone assumed that a container runtime would support all of the capabilities that Docker does. Eventually, Docker representatives explained that the original requirement indicated that the runtime consisted solely of "running the container." This divergence persists today, which is why "container runtimes" is such a perplexing topic.

Docker is a container runtime that includes container creation, packing, sharing, and execution. Docker was created as a monolithic daemon, dockerd, and the docker client program, and features a client/server design. The daemon handled the majority of the logic for creating containers, managing images, and operating containers, as well as providing an API. To transmit commands and obtain information from the daemon, use the command line client. Docker originally included both high-level and low-level runtime features, however those components have since been separated into runc and containerd projects. Docker now includes the dockerd and docker-containerd daemons, as well as



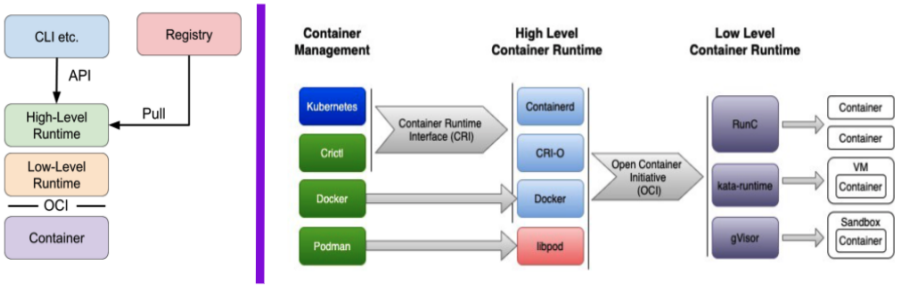
docker-runc. Dockerized versions of vanilla containerd and runc are docker-containerd and docker-runc. The below figure gives details about how the components fit in.

4.2 ContainerD

Containerd was created to be used by Docker and Kubernetes, as well as any other container technology that desires to abstract out syscalls and OS-specific functionality in order to operate containers on linux, windows, suse, as well as other operating systems. With these customers ' needs, containerD made sure that containerd only contains the features they require. Network is one of the most essential components when we are building a system these days. Today network is way more abstractive in nature due to the onset of SDN and service discovery. Whenever a new container is added the route tables need to be updated.

4.3 Low Level and High Level Runtimes

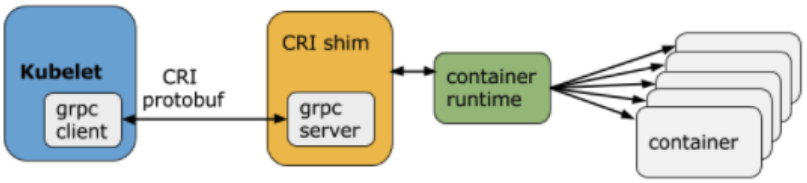
In day to day development developers require much more than just the lower level runtimes. They require bunch of API sets around image creation, image formats, and sharing images. These are the features provided by high level runtimes. Low level runtimes are basically libraries that a developer of high level run times can make use of while developing high level run times to make use of the low lever features. Containers are implemented using namespaces and cgroups which are a Linux features. Namespaces are used to virtualise the underlying file system and networking of the hardware so that each container



can use these system resources. Cgroups are means to limit the CPU, and memory that a container is allowed to use. That's why we set a resource limits quota to the deployment files in Kubernetes. Lower level runtimes like runc, runsc, gVisor are responsible for setting up such namespaces and cgroups. They also run commands in these regions. In day-to-day development a developer won't play around with the low level runtimes.

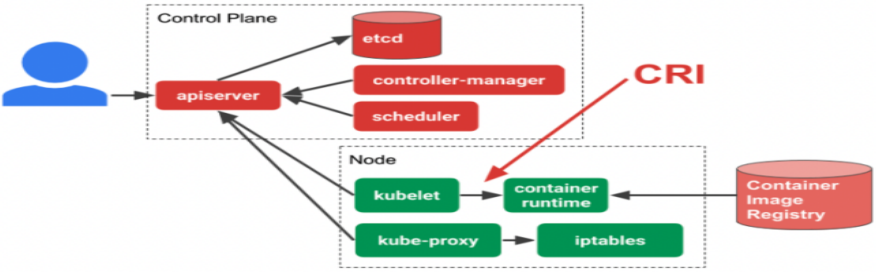
4.4 CRI

The software that starts and stops containers is found at the lowest layers of a Kubernetes node. This is referred to as the "Container Runtime." Docker is the most well-known container runtime, although it is not the only one. In fact, the container runtime environment is rapidly changing. Kubernetes has been developing on a new plugin API for container runtimes in Kubernetes dubbed CRI as part of community's attemtpt to make Kubernetes more adaptable. Kubelet makes use of gRPC framework to communicate to the container runtime. It can also be a CRI shrim for the container runtime. Unix sockets are used for this purpose. Kubelet acts as a client in this case whereas CRI Shim acts as the server.



4.5 Who runs containers in K8S

Kubernetes is an orchestration tool. It only manages the state of containers on the nodes but it does not run containers. Kubelet a component of k8s talks to the container run time to create containers as seen from the figure below.



5 Experiments Performed

In this paper the authors have performed three sets of analyses:

- Analysis of Docker and ContainerD over EKS
- Analysis of Docker and ContainerD over GKE
- Deployment of a manual K8S cluster over AWS and analysis of Containerd over this manual cluster vs amazon managed K8S cluster

5.1 Experimental Design

In this section we talk about the environment setup that we have used over our various experiments. First let's talk about the hardware and software configurations of Experiment 1 and 2. Now, since it's nearly impossible to have the same set of hardware being utilised when research is done among the clouds. We benefit from the fact that we are doing an analysis of Docker and ContainerD. Google Kubernetes Engine and Amazon Elastic Kubernetes Service both allow for deployment of Docker and ConatinerD based worked and master nodes. Hence we study the performance impacts of Docker and ContainerD over the two Managed Kubernetes Services separately. This gives our experimental results authenticity since they are being performed over the same set of hardware.

- The clusters built up in this paper are all based on VMs
- The clusters consists of 1 master and 2 worker nodes each
- All the nodes spawned are in the same geographical regions
- We have executed each benchmark test 10 times and have calculated the mean of all the results so that we ride out the outlier trials. Also, we have distributed the benchmark tests over both the worker nodes to average out any anomaly and hence we achieve more accurate results

5.1.1 Hardware specifications

Table below shows the hardware configurations being used for experiments by the authors

- **Virtual Machine Type:** Since the experiments performed were not using free credits so the authors had to be judicial in the use of cloud resources.

Since, the experiments were analysing each cloud separately hence it was very easy to keep the same set of hardware and software configurations. The VMs chosen for the experiments were kept identical. All the computing resources such as vCPU, memory, storage and network are tightly related to the class of the VM that is being deployed. The authors have made use of general purpose virtual machines offered by both Amazon AWS and Google GKE. Two vCPUs and 8gb of Ram was used as the configuration for the virtual machines. Swap was disabled on all machines as is recommended by Kubernetes best practices.

- **Storage Setup**: Sufficient storage had to be attached to all the worker nodes. We ended up attaching 20gb volume to each Virtual Machine that was spawned and bootstraped into the cluster. Few of our benchmarks required larger storage sizes. For example the I/O disk Sysbench benchmark required 10gb of external storage to perform the tests. Storages were deployed in the same geographical region to maintain homogeneity and minimise network latencys.
- **Networking Benchmarks:** Since our test suite contained Benchmarks that test the communication related parameters these tests were ran in a K8S cluster in the same zone. Since, different zones can result into a lot of external factors affecting the results.

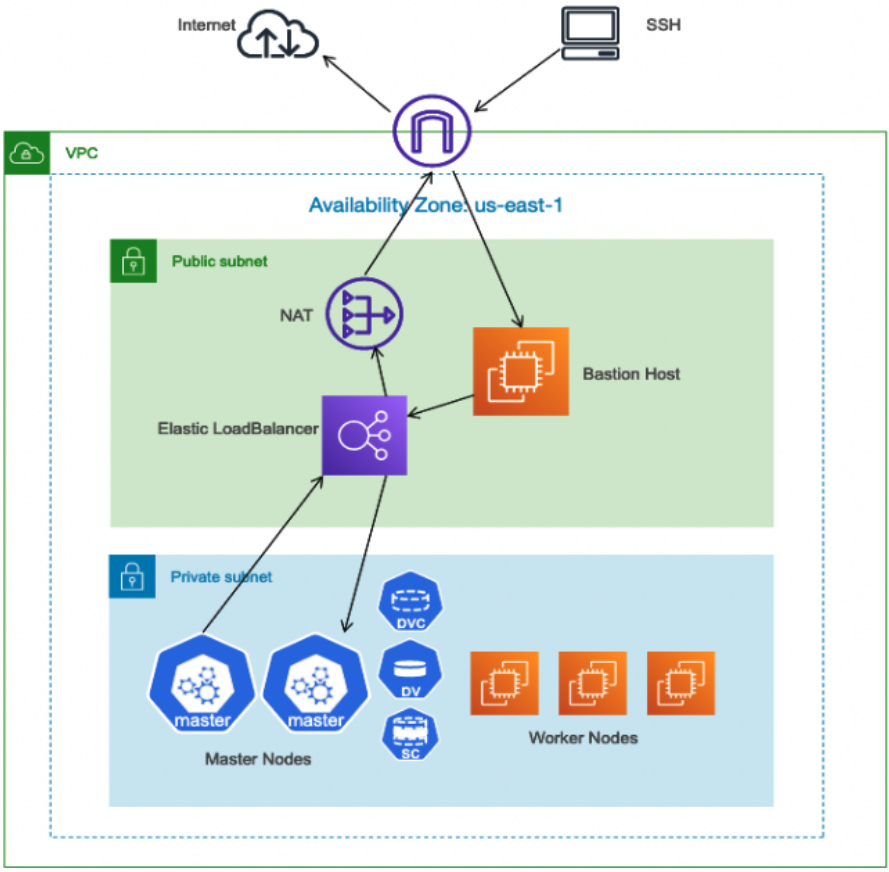
Table 3 Hardware configurations of Managed and Manually deployed K8S Clusters

Cloud	Instance Type	Instance Size	OS	Ram	Disk	Region	CPU
EKS	General	t2.medium	Amazon Linux	8	20	Southeast	2 vCPU
GKE	Custom	-	CentOS	8	20	Southeast	2 vCPU
Manual AWS	General	t2.medium	Amazon Linux	8	20	Southeast	2 vCPU

6 Methodology

6.1 Deployment of Baseline AWS K8S Cluster

In this paper for experiment 3 the authors have deployed their own Kubernetes Cluster on the AWS cloud. After deployment they analyze the performance of a manually deployed cluster as opposed to the managed kubernetes offering of the same cloud provider. All in all to deploy a cluster manually and then bootstrapping it with the Kubernetes API is not a simple task. All in all it required 37 various AWS cloud resources to deploy a full function K8S cluster. Although this cluster was not spanned accross multiple regions but just the one and hence the authors don't advice the setup to be used for production deployments. But still, the cluster was good enough to run the benchmarks and get accurate results to compare with the AWS Elastic Kubernetes service. The Figure below shows the architecture that was used to deploy the K8S cluster in AWS.



The outermost region of the figure is the Virtual Private Cloud. It is spanned across just the single region. AWS allows to span the Virtual Private Cloud across multiple regions but that would have accrued additional cost. Since, the experimental setup was restricted to worker nodes in the same region; it didn't make sense to deploy the VPC over multiple regions. Inside the region two subnets were deployed. One was the public subnet and the other was the private subnet. Since, Master Nodes hosts the Kube API and the Worker Nodes host sensitive customer workloads the authors followed the best practices of deploying these resources in a private subnet. Since these resources are in the private subnet the Authors had complete control over the ingress and egress of traffic from the master and worker nodes and other resources deployed in the private subnet. Now, since Master Nodes need to pull images and need access to the outside internet for update and upgrade purposes it was essential to form a route or a channel throug which the nodes deployed in the private subnet could contact the internet. For this purpose the authors deployed a Public Subnet. The authors deployed 3 main AWS resources in this public subnet. One was the Bastion Host, and the other two

were the Amazon Elastic Load Balancer and the NAT gateway.

Bastion host is basically an AWS EC2 instance with minimum specifications since it doesn't need to perform any computations. It is used as a proxy system through which the kubernetes administrator could ssh into the Master Nodes. This communication is required to facilitate administrators or other users to interact with the kube api and perform cluster operations. Since, there can be multiple Master Nodes(and there are, in production setups) the incoming traffic to the master nodes need to be load balanced. This is achieved through the Amazon's elastic load balancer that balances HTTP traffic to the various master nodes of the k8s cluster. Since the master nodes have no public facing IPs they can't actually directly talk to the internet. For this purpose a NAT gateway was deployed. The NAT gateway encapsulates the incoming traffic from the master nodes with a public facing IP and forwards those communication packets via the internet gateway to the outside internet.

Apart from this some of the benchmark tests required access to persistent volumes. For this purpose the authors deployed a storage class in the cluster and created persistent volumes and persistent volume claims for the same purpose. To deploy various routes route tables and security groups had to be created. Security groups in AWS basically allow the developers to manage what ports are open for communication on the resources attached to these security groups. Five different security groups were created and attached to the bastion host, the master, NAT gateway and the worker nodes. The authors also deployed auto scaling groups. Auto scaling groups in AWS take care of any nodes that may go down in the cluster. These auto-scaling groups were attached to the master and worker nodes.

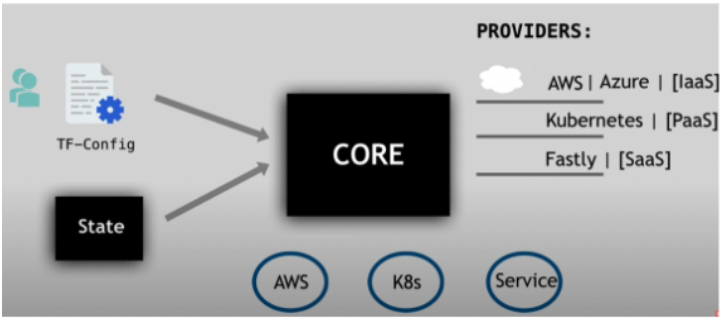
6.1.1 Deployment through Terraform

The authors ended up deploying the baseline cluster multiple times to rule out any ambiguity and for more accurate results using mean. Deploying the cluster multiple times was prone to errors and misconfigurations and was a big operational overhead. To reduce the overhead authors ended up automating the deployment of these 37 resources.

Terraform is an infrastructure as a code tool. It is used to manage and automate infrastructure, platforms over that infrastructure and even the services deployed on top of the platform. Terraform architecture has two main components. The first component is the State. State keeps a tab of the current infrastructure being deployed. The user provides the configuration file that depicts how the end state of the infrastructure should look like. The second component is the Core. The task of the core is to lay out a detailed execution plan of how to reach the end state of the infrastructure. This makes the communication a declarative communication as opposed to an imperative communication. That means, the system has to be told what the end state is

not how to achieve that end state.

Finally after the infrastructure was set up; Kubeadm tool was used to bootstrap the master and worker nodes with kubernetes control api. After that weavenet service was used to provide networking for communication between the containers in different nodes.



7 Results - ContainerD v/s Docker

7.1 Memory Performance

In this section we talk about results of all the experiments consolidated at one place. Let's first look at the results derived from the performance analysis of Docker and Containerd in both AWS EKS and Google GKE. Figure 1 and 2 shows the stream benchmarking results for AWSK EKS and Google GKE for the four vector operations. We can see that ContainerD performs well in all the four operations compared to it's Docker counterpart. This shows that Docker introduces overheads from memory point of view.

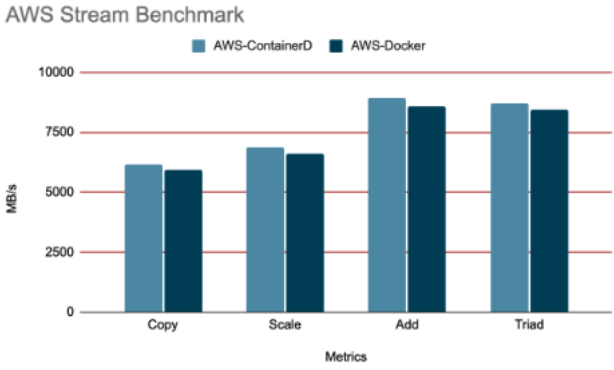


Fig. 1 Stream benchmark results for AWS EKS

The authors experienced similar results for Google's GKE paltform. Here also, ContainerD reduced overheads interms of memory.

Another noteworthy point is that there is not much significant difference in terms of memory as memory is a host OS resource that is managed mainly by the underlying low level container runtime.

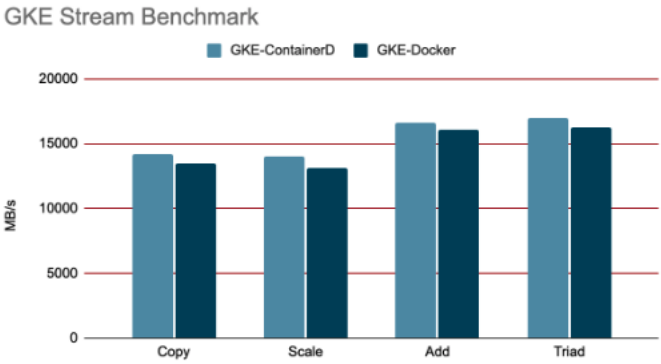


Fig. 2 Stream benchmark results for GKE

7.2 Compute Performance

This result corresponds to the Sysbench benchmark. The main idea behind Sysbench is stress testing. Hence, the problem of prime number generation is taken up. Due to the nature of the problem being solved number of cpu events is the most important performance parameters that needs to be judged. From figure 3 it can be seen that ContainerD performs fairly well than Docker w.r.t compute performance as well. Since the results were averaged aroud several worker nodes the spread of the result is pretty accurate and accounts for any difference between the worker nodes (eventhough the worker nodes had the exact same configurations). From the author's experience the standard deviation of the results were comparatively lower for for ContainerD based environments. This shows that the resutls were fairly consistent over the number of times the experiments were run. This depicts Docker with added abstractions introduces irregular overheards over different trials. Although it wasn't in the scope of the research to compare the different clouds still it is interesting to see that w.r.t to compute capabilities that the Managed Kubernetes Service for almost similar hardware had better compute performance. This shows that the nodes utilised by google for the managed kubernetes services were more compute optimised. It should be noted that there is not much significant difference in terms of CPU as CPU is a host OS resource that is managed mainly by the underlying low level container runtime.

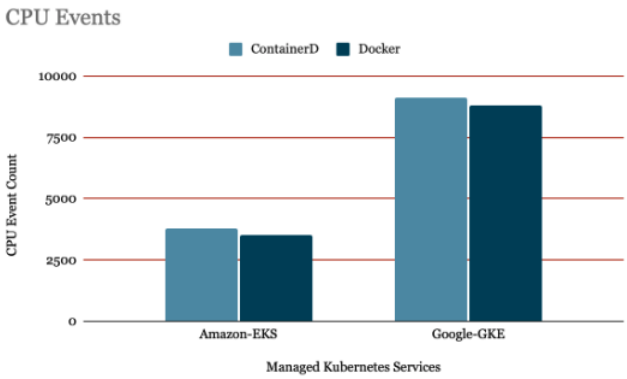


Fig. 3 CPU events for ContainerD and Docker on EKS and GKE

7.3 Network Performance

Since all the worker and master nodes for both GKE and EKS were in a single availability zone. This gives the network results authenticity. Network performance is based on the latency in each Managed Kubernetes environment. The infrastructure that was deployed for network related tests was consistent for both ContainerD and Docker based environments. Two containers were deployed. One acted as the server in one of the worker nodes and the other acted as the client in another worker node, in the same cluster. It can be seen from the Figure 4 that ContainerD base environments in both EKS and GKE had lower latencies. Today, networking is even more platform specific than abstracting away netlink calls on Linux, thanks to SDN and service discovery. The majority of the new overlay networks are route-based, which necessitates updating routing tables every time a new container is formed or destroyed. These modifications must also be communicated to service discovery, DNS, and other systems. The developers of containerd did away with networking in containerd since it will result in supporting a lot of code to support all kinds of networks, interfaces, hooks and etc. The team ended up using containerd's events systems that allows consumers to suscribe to events. The added complexity of Docker networking does being in additional overheads as seen by the results as well. High level conatiner runtime plays a major role in terms of network i.e. they are responsible for network namespace management, how one container will join other container's network and interaction with the Container Network Interface (CNI) plugins for cluster networking. As a result, difference seen in the performance is significant.

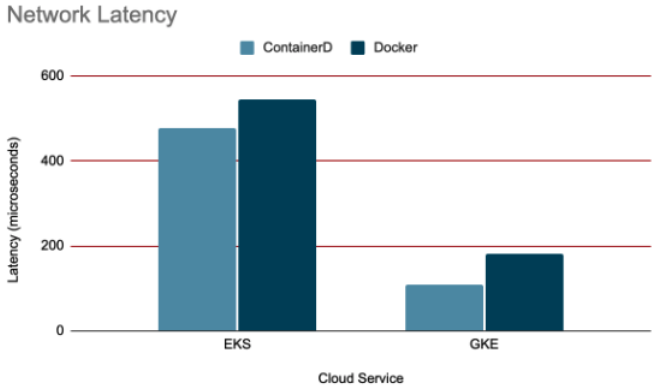


Fig. 4 Network latency for ContainerD and Docker on EKS and GKE

7.4 Disk Performance

To analyse Disk performance Random read and writes were taken as the parameters to judge performance. The authors chose Random reads and writes since in most of the scenarios the storage files are distributed among large number of files. Hence the files on the filesytems are pretty distributed. Hence, it would make more sense to analyse the random reads and writes. Authors have used Persistent Volumes for the purpose of disk storage. From the figures 5 & 6 we can see that the for both the random read and write scenarios ContainerD performed better than Docker based environment. Significant difference can be seen in terms of disk performance as high level container runtimes are responsible for interaction with Container Storage Interface (CSI) to use external block and file storage systems for containerized workloads.

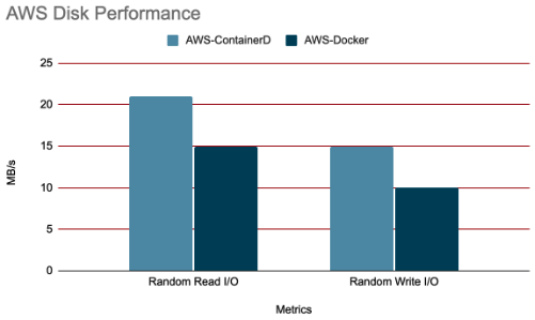


Fig. 5 Random I/O performance of ContainerD v/s Docker on AWS EKS

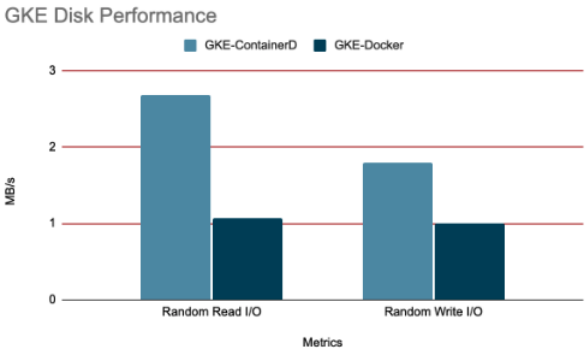


Fig. 6 Random I/O performance of ContainerD v/s Docker on GKE

8 Results - Managed K8S v/s Manual K8S

In this section we will analyse the performance of a Managed Kubernetes Service (EKS) v/s a Manually deployed K8S cluster utilising the AWS cloud resources. The authors have tried to keep both the environments as similar as possible. Similar VM instance, storage classes, network configurations were utilised. This provides for more accurate results. From the Figures 7, 8, and 9 it can be seen that the Manually Deployed Kubernetes cluster performs better than the Managed K8S cluster. This was opposite to the initial hypothesis of the authors. The authors had hoped that Managed K8S services would be providing for more optimised performance. On significant research it was found out that the Managed K8S cluster had additional services as pods being run on the cluster. These services included logging, monitoring agents, service mesh running. These add-ons added to the pressure on the cluster resources hence affecting the benchmark results. Authors found out that deploying similar resources on the Manually Deployed K8S cluster brought down the performance as well. Hence, in conclusion both the manually deployed and managed k8s clusters performed equally well.

9 Conclusion

In this paper, the authors evaluated the performance impacts of using the Docker container runtime system over the ContainerD runtime system. The authors also performed an analysis of a manually deployed K8S cluster from scratch v/s the managed offering of the same cloud vendor. The Managed K8S offerings choosen were AWS Elastic Kubernetes Service and Google's Kubernetes Engine.

The experiment evaluations indicated that the Docker container runtime introduced overheads in all spheres of performance be it compute, networking, disk or memory. This is the reason that even Google is deprecating the use of Docker as the choice of container runtime in GKE. The environments used in

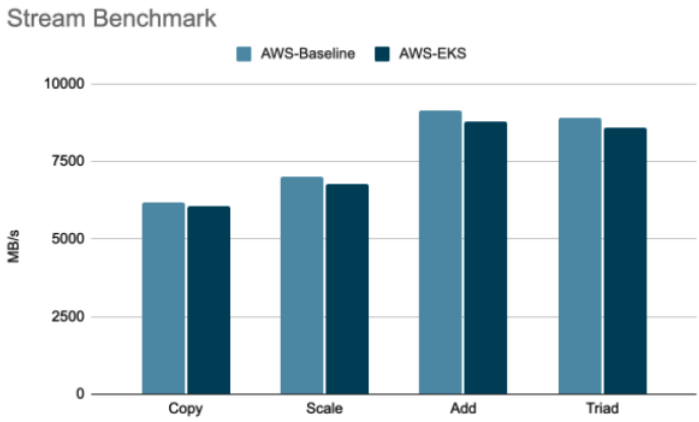


Fig. 7 Memory analysis for Manual v/s Managed AWS K8S Cluster

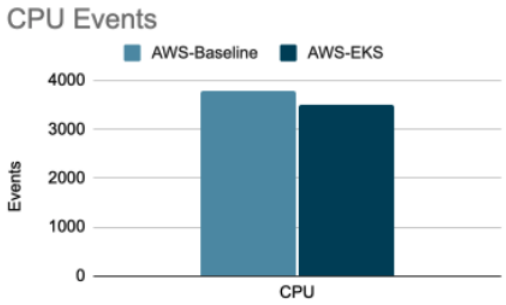


Fig. 8 Compute analysis for Manual v/s Managed AWS K8S Cluster

the experiments were of the same configuration since the goal was not to do an inter cloud analysis. Hence, the the results collected are pretty accurate. It was found out by the authors that eventhough a manually deployed kubernetes cluster performed better than the managed offering; but the managed services provided a lot more capabilities running on the cluster that hampered the performance of the system.

10 Future Work

The authors would want to study the impact on costing when using different container runtimes. Specially in the scenario when the number of pods needs to be scaled to thousands. The authors would also want to analyse the performance when thousands of workloads are running on the cluster. The authors expect to find even more gains when the cluster is scaled up for containerD. The authors would also want to converge to a more production based manual deployment of the K8S cluster. For instance none of the managed kubernetes providers deploy crucial components of the k8s cluster like the etcd database,

Fig. 9 Network analysis for Manual v/s Managed AWS K8S Cluster

the scheduler service on separate nodes. Segregating these kube system services would provide for better availability. The authors would want to study the impact of the same as well.

References

[1] https://kubernetes.io/docs/concepts/architecture/cri/

[2] Large-scale cluster management at Google with Borg Abhishek Verma and Luis Pedrosa and Madhukar R. Korupolu and David Oppenheimer and Eric Tune and John Wilkes, 2015, Proceedings of the European Conference on Computer Systems (EuroSys), Bordeaux, France

[3] https://www.docker.com/

[4] https://containerd.io/docs/

[5] Goasguen, S.: Docker Cookbook: Solutions and Examples for Building Distributed Applications. O'Reilly Media, Inc, Newton (2015)

[6] Performance Evaluation of Container Runtimes, Lennart Espe, Anshul Jindal a , Vladimir Podolskiy and Michael Gerndt, Chair of Computer Architecture and Parallel Systems, TU Munich, Garching, Germany

[7] Aristotle Cloud Federation: Container Runtimes Technical Report, Peter Z. Vaillancourt* Rich Wolski** Bennett Wineholt* Christopher R. Myers* Tristan J. Shepherd* Ben Trumbore* Sara C. Pryor* Resa Reynolds* Jeffrey Lantz* Jodie Sprouse* Richard Knepper* David Lifka*, *Cornell University **University of California, Santa Barbara

[8] https://github.com/akopytov/sysbench

[9] https://www.cs.virginia.edu/stream/