Aalto University
School of Science
Degree Programme of Computer Science and Engineering

Rajagopalan Ranganathan

# A highly-available and scalable microservice architecture for access management

Master's Thesis
Espoo, September 17, 2018

| | |
|---|---|
| Supervisor: | Professor Mario Di Francesco, Aalto University |
| Instructors: | Markku Rossi M.Sc., Computer Science, SSH Communications and Security Oyj |
| | Gopika Premsankar, M.Sc. (Tech.), Aalto University |

Aalto University
School of Science
Degree Programme of Computer Science and Engineering

ABSTRACT OF MASTER'S THESIS

| | | | |
|---|---|---|---|
| **Author:** | Rajagopalan Ranganathan | | |
| **Title:** A highly-available and scalable microservice architecture for access management | | | |
| **Date:** | September 17, 2018 | **Pages:** | 73 |
| **Professorship:** | Mobile Computing, Services and Security | **Code:** | SCI3045 |
| **Supervisor:** | Professor Mario Di Francesco | | |
| **Instructors:** | Markku Rossi M.Sc., Computer Science, SSH Communications and Security Oyj<br>Gopika Premsankar, M.Sc. (Tech.), Aalto University | | |

Access management is a key aspect of providing secure services and applications in information technology. Ensuring secure access is particularly challenging in a cloud environment wherein resources are scaled dynamically. In fact keeping track of dynamic cloud instances and administering access to them requires careful coordination and mechanisms to ensure reliable operations. PrivX is a commercial offering from SSH Communications and Security Oyj that automatically scans and keeps track of the cloud instances and manages access to them. PrivX is currently built on the microservices approach, wherein the application is structured as a collection of loosely coupled services. However, PrivX requires external modules and with specific capabilities to ensure high availability. Moreover, complex scripts are required to monitor the whole system.

The goal of this thesis is to make PrivX highly-available and scalable by using a container orchestration framework. To this end, we first conduct a detailed study of mostly widely used container orchestration frameworks: Kubernetes, Docker Swarm and Nomad. We then select Kubernetes based on a feature evaluation relevant to the considered scenario. We package the individual components of PrivX, including its database, into Docker containers and deploy them on a Kubernetes cluster. We also build a prototype system to demonstrate how microservices can be managed on a Kubernetes cluster. Additionally, an auto scaling tool is created to scale specific services based on predefined rules. Finally, we evaluate the service recovery time for each of the services in PrivX, both in the RPM deployment model and the prototype Kubernetes deployment model. We find that there is no significant difference in service recovery time between the two models. However, Kubernetes ensured high availability of the services. We find that Kubernetes is the preferred mode for deploying PrivX and it makes PrivX highly available and scalable.

| | |
|---|---|
| **Keywords:** | docker, container, kubernetes, container orchestration, microservices, access management |
| **Language:** | English |

# Acknowledgments

I would like to thank my thesis supervisor Professor Mario Di Francesco for giving me an opportunity to work on this thesis and providing valuable insights. I would like to specifically thank my instructors Markku Rossi and Gopika Premsankar for their continuous support and patience during this entire tenure. I am grateful to them for constantly guiding me through out this thesis work.

I would like to thank my father, in-laws and friends for their support and motivation. Special thanks to my wife and daughter for constantly encouraging me throughout my academic journey and writing this thesis. This thesis would not have been possible without their help and support.

Thank you.

Helsinki, September 17, 2018

Rajagopalan Ranganathan

# Abbreviations and Acronyms

| | |
|---|---|
| AD | Active Directory |
| API | Application Programming Interface |
| AWS | Amazon Web Services |
| CD | Continuous Delivery |
| CI | Continuous Integration |
| CLI | Command-line Interface |
| CNCF | Cloud Native Computing Foundation |
| DNS | Domain Name System |
| GCP | Google Cloud Platform |
| GKE | Google Kubernetes Engine |
| GUI | Graphical User Interface |
| IaaS | Infrastructure as a Service |
| IPC | Interprocess Communication |
| MITM | Man In The Middle |
| OS | Operating System |
| PAM | Privileged Access Management |
| RBAC | Role Based Access Control |
| RDP | Remote Desktop Protocol |
| REST | Representational State Transfer |
| RPC | Remote Procedure Call |
| SaaS | Software as a Service |
| SSH | Secure Shell |
| TCP | Transmission Control Protocol |
| UDP | User Datagram Protocol |
| VM | Virtual Machine |
| VMM | Virtual Machine Monitor |

# Contents

# List of Figures

# Chapter 1

# Introduction

Software development processes have been evolving at a fast pace. Competition has become very fierce and being early to market has become a critical factor in attracting customers and determining the eventual success of a product or a service [52]. Software has evolved to rapidly-changing web-based services that emphasize on the agility of organizations to develop and deploy new software components and features [53]. Traditional software development methodologies like the classic waterfall model do not scale well to facilitate iterative software development with reduced release cycles [61]. This has paved the way for lean and agile software development processes, which enable iterative software development with short release cycles [66].

Cloud computing provides unlimited access to computing resources and supports agile deployment of software services [75]. More specifically, cloud computing refers to the applications delivered as services over the Internet and the hardware and systems software in the datacenters that provide these services [31]. Virtualization plays an important role in cloud computing as it provides an abstraction layer for storage and other computing resources. The most popular virtualization technologies available for servers are: platform and container virtualization. Linux containers are the most widely used implementation of operating system-level virtualization. Virtualization helps in creating Virtual Machines (VMs) which are efficient, isolated duplicates of a real machine [63].

Containers are lightweight and easily manageable especially if compared to VMs. These attributes have made containers highly desirable in the recent years [67]. Containerization has enabled rapid and quick deployment of software by establishing a unified environment from development until deployment. There are several container platforms; Docker is one of the most popular container platform [40]. The rise of containers has paved the way for new software development practices based on microservices. In the microservice-based architecture, the application is structured as a collection of loosely bound services [44]. Services are organized around their capabilities and communicate with each other using well defined interfaces [47]. This is a shift from the prevalent monolithic software architecture wherein the legacy or ‘existing monolithic’ applications, and services, can be “Containerized” by splitting it into microservices, each running on a single or multiple containers. Containers are one of the primary enabling technologies for microservices. The scalability and high-availability of containerized applications is obtained through by having multiple replicas of the individual services.

Containers are ephemeral and managing their life cycles in large deployments is challenging. Ensuring their availability and replica count in an distributed environment is particularly demanding. Container orchestration frameworks and tools help manage the containers. Automated container deployment, scaling, resiliency, rescheduling the failed containers and container management are some of the features offered by container orchestration frameworks [72]. Kubernetes, Docker Swarm and Nomad are the most popular container orchestration frameworks.

With the advent of cloud computing, the information technology infrastructure of organizations has undergone a massive change. Managing dynamically allocated resources on the cloud and securely providing access to them is a key aspect. Privileged Access Management (PAM) solutions enable organizations to securely provide access to its assets and conform with compliance requirements by securing, monitoring and managing the accounts and their access. Modern PAM solutions should support dynamic instances on the cloud as well as static on-premise infrastructure. In this context, PrivX is a lean on-demand access management software developed by SSH Communications Security Oyj [23]. It supports all major cloud infrastructure providers such Amazon, Google, Azure, OpenStack as well as the traditional on-premise infrastructure. It simplifies access management with short-lived certificates used to grant access to resources.

PrivX is based on the microservice architecture but the deployment model does not leverage containers. Complex scripts are required to ensure high availability. Addressing scalability is a challenge as it requires external load balancers with sticky session support. PrivX requires an external Postgres database. Additionally, PrivX has a watchdog script to monitor and restart the failed services of PrivX. These reasons have motivated us to containerize PrivX and deploy it with a container orchestration framework.

## 1.1 Scope and goals

The goal of this thesis is to containerize PrivX, eventually making it highly available and scalable using a container orchestration framework. A containerized PrivX enables faster release cycles with Agile development methodologies. In fact, it accelerates Development and Operations (DevOps) and simplifies the Continuous Integration (CI) and Continuous Delivery (CD). The container orchestrator should simplify PrivX deployment by eliminating the need for external load balancers. The goal is to create a single unified solution that could be deployed both on-premise and on cloud environments.

To this end we identify the following goals.

- Find a container orchestration framework suitable for the considered scenario.
- Containerize PrivX according to the selected solution.
- Create the Postgres database as a service within PrivX according to the microservice architecture.
- Automate deployment of PrivX through a container orchestration framework.

## 1.2 Contribution

The contributions of this thesis are the following

- We identify the main features required from the container orchestration framework in the context of PrivX. We then evaluate the features of the available open source container orchestration frameworks (Kubernetes, Docker Swarm and Nomad) in light of these requirements.
- Next, we implement different microservices of PrivX as Docker containers. This also includes creating a database as a service.
- Finally, we deploy the containerized PrivX using Docker on a Kubernetes cluster. We also consider the automated build and deployment of PrivX.
- A proof of concept smart scaling tool is created that scales services in PrivX based on predefined rules. Additionally, an application level log aggregation tool is created to collect the logs.

## 1.3 Structure

The rest of this work is organized as follows. Chapter 2 introduces the relevant background about virtualization, containers, and Docker. Chapter 3 introduces the microservice architecture and selected container orchestration frameworks. Chapter 4 describes PrivX, identifies its specific requirements and evaluates the features of the container orchestration frameworks in this context. Chapter 5 describes our prototype implementation of PrivX on a custom Kubernetes cluster with the experiments and evaluation performed. Finally, Chapter 6 provides some concluding remarks as well as directions for future work.

# Chapter 2

# Virtualization

The chapter provides background information on the technologies and concepts of virtualization relevant to this thesis. Section 2.1 introduces the concept of virtualization. Section 2.2 describes hypervisor-based virtualization and section 2.3 describes operating system level virtualization. Section 2.4 introduces Docker and describes its components and architecture in detail.

## 2.1 Virtualization

Virtualization is a methodology of sharing and dividing the resources of a computer into multiple execution environments, by applying one or more technologies such as hardware and software partitioning, time-sharing, partial or complete machine simulation, emulation, quality of service, and many others [50]. The usage of system virtualization in modern software development and deployment is prevalent. It enables software and service providers to serve a multitude of customers with physical and virtual resources on demand [37]. With system virtualization, virtualized resources can be scaled up or out to meet the increase in demand. When the demand decreases, the virtualized resources can be removed (or scaled down). Scaling down seamlessly enables resources to be free and available for other purposes and helps in keeping the infrastructure cost for the company low.

Virtualization enables the development and production environments for the software to be the same. It enables running several virtual machines on a single server hardware. The machines run in complete isolation and provide better scalability and usage of the underlying hardware resources. The most popular virtualization technologies are *hypervisor-based virtualization* and *Operating-system-level virtualization.*

## 2.2 Hypervisor-based virtualization

A hypervisor or Virtual Machine Monitor (VMM) is a piece of computer software that enables creating and running of virtual machines (VMs). It allocates the host resources such as memory and CPU to a collection of VMs (guest machines). The VMs provide an execution environment of that of a real machine [63]. A VMM has three essential characteristics [63]: VMM provides an environment for programs that is identical with that of a real machine; programs that run in such virtualized environments only show a minor reduction in performance and the VMM is in complete control of the system resources in the host system.

There are two hypervisor-based virtualization architectures (Figure 2.1): *Type 1, which is native or bare metal* and *Type 2, hosted hypervisors*. In the type 1 architecture, the hypervisor runs directly on top of the host hardware. It controls the host hardware and manages the virtual machines. Xen, Microsoft Hyper-V, VMware ESX/ESXi are some of the notable hypervisors based on this architecture. In type 2 architecture, the hypervisor runs as an additional software layer on top of the host operating system. The hypervisor does not directly control the system resources such as CPU or other resources, but rather relies on the underlying host operating system to perform the same. VMware Workstation, VirtualBox, QEMU are notable examples of type-2 hypervisors.

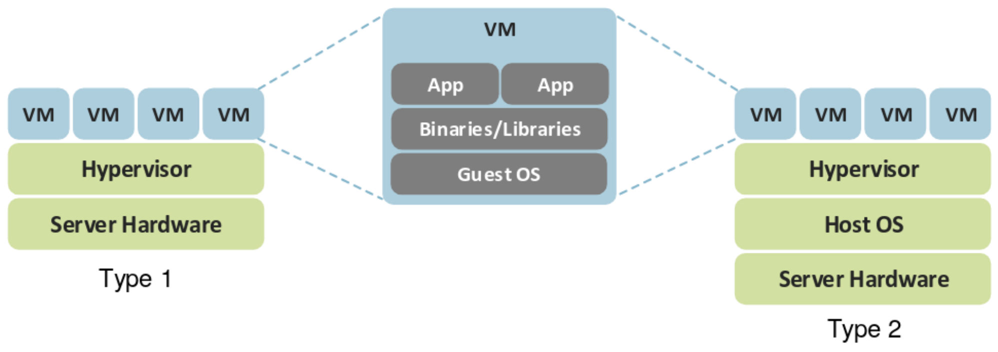

Figure 2.1: Hypervisor-based virtualization architecture [56].

Hypervisors ensure that there is total isolation between the different VMs that are running on the same hardware. Additionally, they ensure that the virtual machines behave no differently than real hardware i.e., the same behavior is expected from the programs when running in a virtual machine as compared to running on a physical hardware. Hypervisors provide a management interface to control the running and execution of VMs and managing their respective resources. Hypervisors provide the ability to take backups and create snapshots of the VMs. This enables portability and deployment at ease. In the cloud computing environment, creation, deletion and management of VMs can be controlled through a set of well-defined Application Programming Interfaces (APIs) enabling applications to scale-up and scale-down as per the requirements. This has notably driven the Infrastructure as a Service (IaaS) model in the cloud computing world [38].

Despite their benefits hypervisor-based solutions have their drawbacks. Programs that require isolation but rely on the same operating system lead to creation of multiple VMs with the same operating systems. This leads to poor utilization of the resources. Since VMs require booting up an operating system they require considerable time to be started. Even though they provide an experience as close to a real operating system, there is performance degradation when compared to physical machines [34].

## 2.3 Operating system level virtualization

Operating system level virtualization also known as containerization or container-based virtualization is a lightweight alternative to the hypervisor based virtualization. There is no hypervisor involved and virtualization is done at host operating system (OS) level. All virtualized instances, i.e., containers, share the host OS kernel. From a user perspective the containers give an experience of stand-alone operating systems [74]. Since there is no hypervisor involved, this largely reduces the runtime overhead and sharing a same operating system also reduces the storage overhead. Container-based virtualization has a weaker isolation when compared to hypervisor-based virtualization solutions.

As shown in Figure 2.2, containers provide a level of abstraction on top of host operating system kernel, allowing each container to behave as an independent operating system with isolation. Containers are created based on an image. A container image is a stand-alone executable package that is lightweight and contains all the necessary artifacts that are needed for the container to run such as: code, runtime, system resources [27]. This package can be backed-up, copied and distributed. Multiple instance of containers can be executed in the same host machine or different host machines depending upon the requirement for fault tolerance.

From a user perspective, the containers provide an illusion that the processes executing inside them are running on different physical machines. Linux kernels are predominant in supporting container-based virtualization

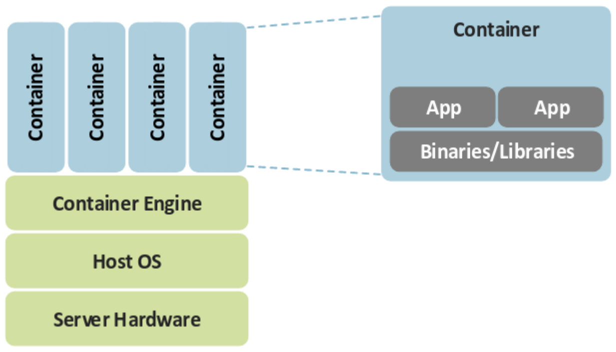

Figure 2.2: Container-based virtualization architecture [56].

and the popular container-based virtualization solutions such as Docker rely on the features provided by it. The management interface provided by the Linux kernel enables monitoring, execution and administration of the containers. Isolation is achieved using *namespaces*, which allows each container to have a different view of the underlying system. Resource management such as CPU, memory and I/O are administered and controlled with the help of *control groups* (cgroups).

The technologies enabling container-based virtualization are discussed further below.

### Control groups

Control groups commonly known as cgroups enables to control the system resources such as CPU, network bandwidth, memory, block I/O or a combination of these resources for a user-defined groups of processes running on a given system [4]. Cgroups helps in achieving fine-grained control over allocating, denying, dynamically reconfiguring, managing and monitoring the system resources. The hardware resources can be divided up among tasks and users, increasing the overall efficiency and resource utilization [4]. The control group config (cgconfig) service can be used to persist the cgroup configurations and make them available from system boot time [4].

cgroups are organized hierarchically with child groups inheriting some attributes from the parents. However, like the Linux process model, in cgroups many different hierarchies can coexist in a system simultaneously. The cgroup model is best described as one or more distinct tree of processes as shown in Figure 2.3. A subsystem is the representation of a single system resource such as system memory, CPU time, network, I/O operations and others. Multiple separate hierarchies of cgroups are necessary because each hierarchy is attached to one or multiple subsystems [4] as shown in the Figure 2.3.

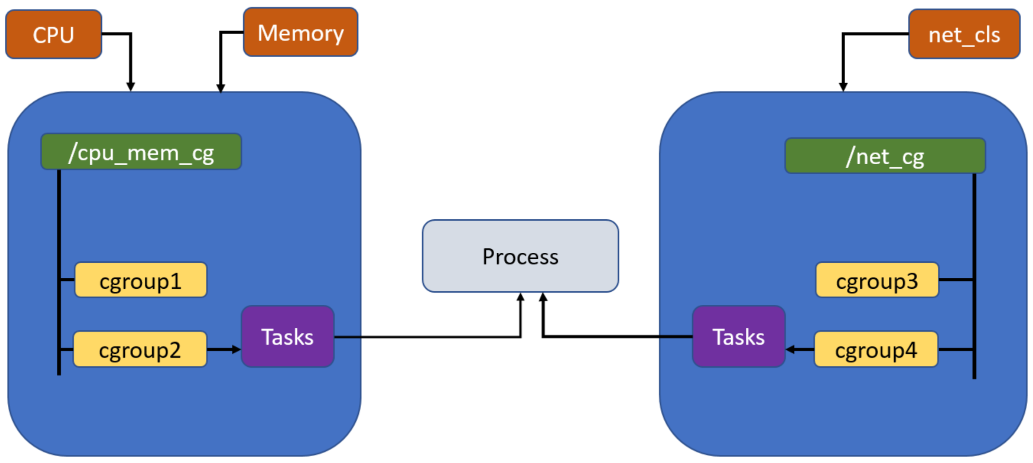

Figure 2.3: cgroups hierarchical structure.

### Namespaces

Namespaces are used to provide resource isolation. They provide an abstraction for the global system resource that makes it appear as if the process within the namespace owns its own isolated instance of that resource. Changes made to a resource inside a namespace do not affect the global resource. This enables different namespaces (or processes) to have a different view about the physical resources of the system [5]. This is a key feature which enables container-based virtualization. There are 7 different namepsaces available from Linux kernel version 4.10[1].

- *Mount*[2] *(mnt)* namespaces provide isolation on the list of mount points in each namespace instance. This enables each process belonging to different mount namespace instance to see different directory hierarchies.

- *Process ID*[3] *(pid)* namespace provides isolation between process IDs. Processes in different PID namespaces can have the same PID number.

[1]http://man7.org/linux/man-pages/man7/namespaces.7.html
[2]http://man7.org/linux/man-pages/man7/mount_namespaces.7.html
[3]http://man7.org/linux/man-pages/man7/pid_namespaces.7.html

This enables containers to suspend and resume a process or a set of process. Additionally, this enables migrating containers between hosts retaining their PID numbers and avoiding any potential conflict that may arise otherwise.

- *Network*[4] *(net)* namespaces provide isolation of networking system resources such as: network devices, IPv4 and IPv6 protocol stacks, IP routing tables, firewall rules. A physical networking device can reside in only one network namespace. However, with the help of virtual network (VETH) device, communication can be provided between namespaces and external hosts. It provides the namespaces with its own networking interface. When a namespace is freed or destroyed, the VETH devices contained in it are also destroyed.

- *Interprocess Communication*[5] *(ipc)* namespaces provide isolation for IPC resources and Posix message queues. Each IPC namespace has its own set of IPC resources. The IPC objects are visible to all processes that are inside the same IPC namespace, but are not visible to processes that are in a different namespace.

- *UTS*[6] namespaces provide isolation of the following two system identifiers: The host name and Network Information Service (NIS) domain name. This enables containers to have their own namespace when compared to the host.

- *User ID*[7] *(user)* namespaces provide isolation of security identifiers and attributes such as user IDs, Group IDs, root directory, keys and capabilities. A process can have a different user and group ID inside and outside a given namespace. This allows for a process running with full privileges inside a namespace to be unprivileged outside the namespace. This enhances the security of the system and protects it from misuse and arbitrary attacks.

- *Control group*[8] *(cgroup)* namespaces provide a virtualized view of the process's cgroups. Each cgroup namespace has its own cgroup root directories.

[4]http://man7.org/linux/man-pages/man7/network_namespaces.7.html
[5]http://man7.org/linux/man-pages/man7/namespaces.7.html
[6]http://man7.org/linux/man-pages/man2/uname.2.html
[7]http://man7.org/linux/man-pages/man7/user_namespaces.7.html
[8]http://man7.org/linux/man-pages/man7/cgroup_namespaces.7.html

## 2.4 Docker

Docker is a computer program that performs containerization. It enables separation of application and infrastructure, thereby facilitating faster software deliveries[9]. Linux cgroups and namespaces are the building blocks of Docker [30]. Dockers enable developing, maintaining, shipping and running of container-based applications. It is an open source platform and enables scaling of applications at ease. Docker provides a mechanism to package application, runtime, libraries and other necessary configurations into images. Docker provides tools for version management, container management, monitoring and deployment of containers. Since Docker leverages container-based virtualization it is possible to run multiple containers on a single host in a secure isolated manner. Docker enables the development and production environment to be the same, meaning that the necessary infrastructure could be created inside a Docker container. This reduces production issues and reduces the time to market a new product. These features of Docker have made them popular with continuous integration (CI) and continuous development (CD) work flows required for modern software development methodologies such as Agile [46].

Containers are stateless and they can be brought up and down dynamically. For example: systems that require large data storage, the database component (the database container) can be scaled up horizontally and when the need is no more, it can be scaled down. This enables smart scaling and workload management. Docker is tailor made for these requirements.

Docker engine is a part of Docker which creates and runs the Docker containers [6]. Figure 2.4 illustrates the Docker Engine architecture. It is based on the client-server architecture model and contains three major components:

- *Server* is the daemon process (the dockerd command).
- *REST API* specifies interfaces that application programs can use to communicate with the server(daemon) and provide instructions to the daemon.
- *Command Line Interface (CLI)* is the client (docker command), it uses the REST APIs to communicate and control the daemon using scripts or direct commands.

[9]https://docs.docker.com

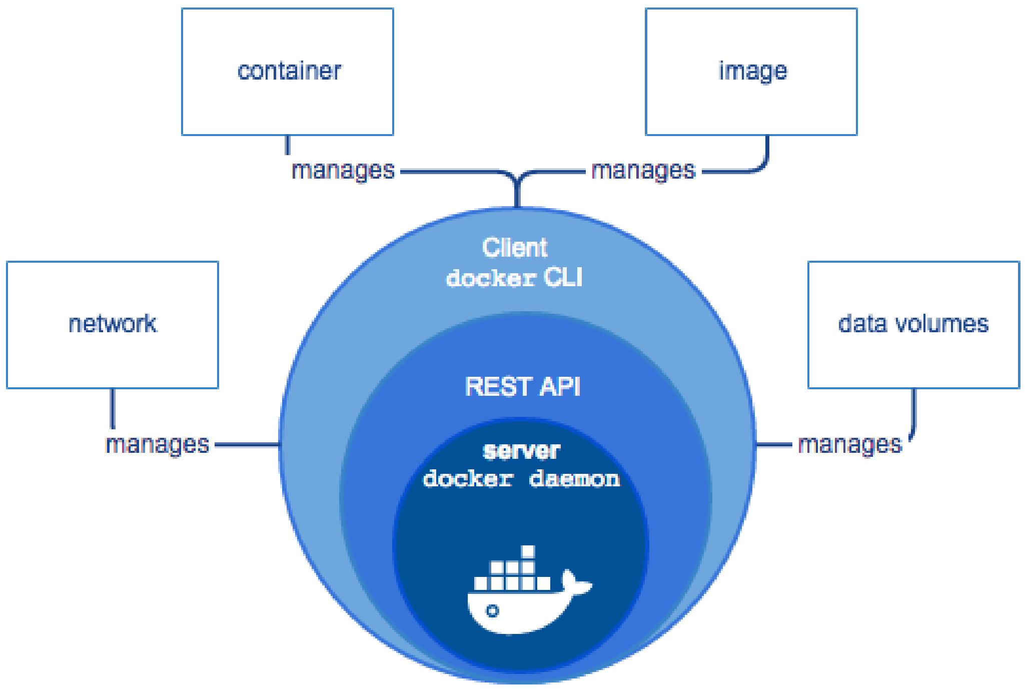

Figure 2.4: Docker Engine architecture [6].

### Docker Architecture

Docker Engine is the component that creates and runs Docker containers [6]. Figure 2.4 illustrates the Docker Engine architecture. It is based on the client-server architecture model and contains three major components:

- *Server* is the daemon process(the dockerd command).
- *REST API* specifies interfaces that application programs can use to communicate with the server(daemon) and provide instructions to the daemon.
- *Command Line Interface (CLI)* is the client (docker command), it uses the REST APIs to communicate and control the daemon using scripts or direct commands.

Users manage and control the Docker containers using the Docker client. Docker client communicates with the Docker daemon. The Docker daemon does the important tasks such as building, running and distributing the Docker containers. It is not necessary that the client and the daemon run on the same system, the client can communicate with a remote daemon running on a different system. The client and daemon communicate using the REST APIs, over Unix sockets or a well-defined network interface. Figure 2.5 depicts the Docker architecture.

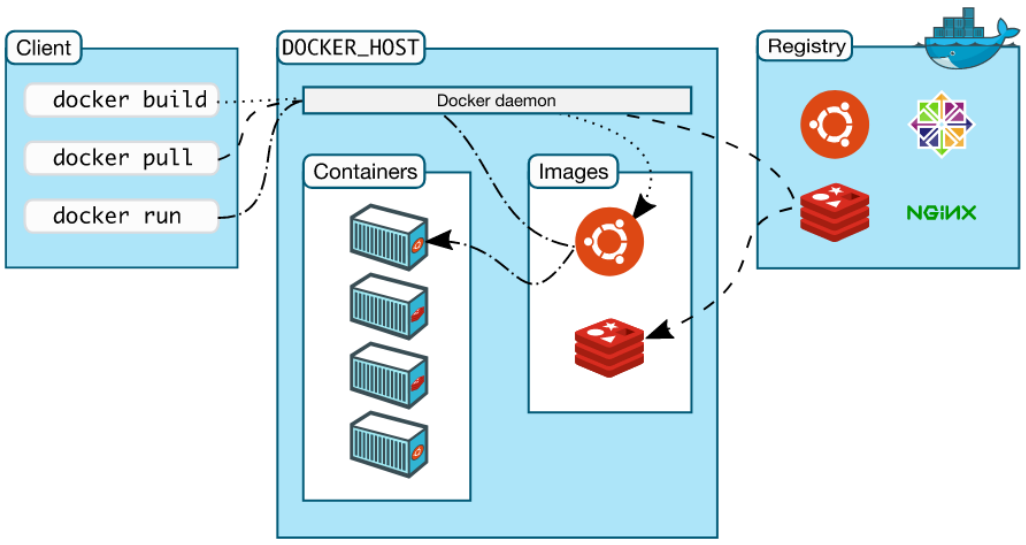

Figure 2.5: Docker Architecture [6].

- *The Docker daemon (dockerd)* manages all the Docker objects such as images, containers, networks and volumes. It communicates with the Docker client for managing the Docker objects notably the containers. A Docker daemon can communicate with other daemons to manage the Docker services effectively. Only the Docker client is authorized to communicate with the daemon, meaning direct communication with the daemon is restricted.

- *The Docker client (docker)* is the primary interface for communicating with the Docker platform. Docker users interact with the Docker platform using the CLI provided by the client to execute the commands. User commands are sent to the Docker daemon and the daemon executes them on users behalf. One single client can communicate with multiple Docker daemons.

- *Docker registries* are used to store Docker images. The popular public registries are Docker cloud and Docker Hub. By default, Docker looks for images in Docker Hub. The registries can also be private and access can be restricted as desired. Developing and storing of images is streamlined with registries, which simplifies their distribution. Docker client is used to access these registries and download the images. Commands like "docker pull" or "docker push" are used to pull and update the images to the registry respectively. It can be noted that these commands are similar to source code versioning systems [55] like GIT[10].

- *Docker Images* are read-only templates for creating Docker containers. A Docker image is usually based on a base image with some additional customizations that are needed for the application to run. It encompasses the libraries and binaries that are needed to build and run the application and services. Users can create a new image or reuse an existing one in the registry. Creating a Docker image involves creation of a Dockerfile with a simple syntax that defines the steps necessary for creating an image and running it. Each step in the Dockerfile creates a layer in the image and any subsequent changes to the Dockerfile results in only the changed layers getting rebuilt. This effectively means that the subsequent versions of the images are the difference between the previous versions. This enables to have a complete audit trail of changes that went between the different versions [39]. Thus, Docker images are designed to be small, fast and lightweight.

- *Dockerfile* is a simple text file that contains all the commands and instructions on how to build a Docker image, configure a container and run the application within the Docker container. The instructions inside the Dockerfile are executed line-by-line in a sequential manner. The command "docker build" is used to build a Docker image by pointing it to a correct Dockerfile. As explained earlier, the build system checks for differences in the lines in the Dockerfile with previous versions and only adds (builds) new layers for the changed ones. A Dockerfile must start with a 'FROM' instruction, specifying the base image to be used, it can also be mentioned as 'SCRATCH' indicating that there is no base image and instructs the Docker to start with an empty file system. System configurations are also specified in the Dockerfile; for instance, listening ports and environment variables for applications. These configurations are stored in separate layer as a JSON configuration file.

[10]https://git-scm.com/

- *Multi-stage Dockerfile* is a relatively new approach to reduce the size the Docker images. The images produced by the build system may contain compilers or other tools that are not needed for the final deployment. For example, let us consider a web server written in Golang, that is serving requests in a designated port. When creating a Dockerfile to build and compile the source code we might include a Golang compiler and other needed libraries. Once the final server executable is created, the resources such as the compiler are obsolete. To minimize the size of the image and improve the build efficiency, multi-stage Dockerfile is composed of various stages. The build artifacts from one stage could be copied to the next [21]. In this way, a pipeline of the needed build artifacts can be defined and the final stage can contain only the minimal set of artifacts that are needed for the application to run.

- *Docker containers* are the runnable instance of an image. Docker containers are well isolated from each other and the host machine where they are running. Multiple containers can be instantiated from a single Docker image. Containers network, storage and other system properties can be controlled and modified according the requirements. Each container has its own writable container layer, enabling them to share a same underlying image but having individually different data states. This is shown in the Figure 2.6. The changes are stored in the writable layer, and when the container is deleted or removed, the writable layer associated with it is also deleted. This ensures that the underlying image is unchanged. The changes can be specifically saved to a persistent storage if required.

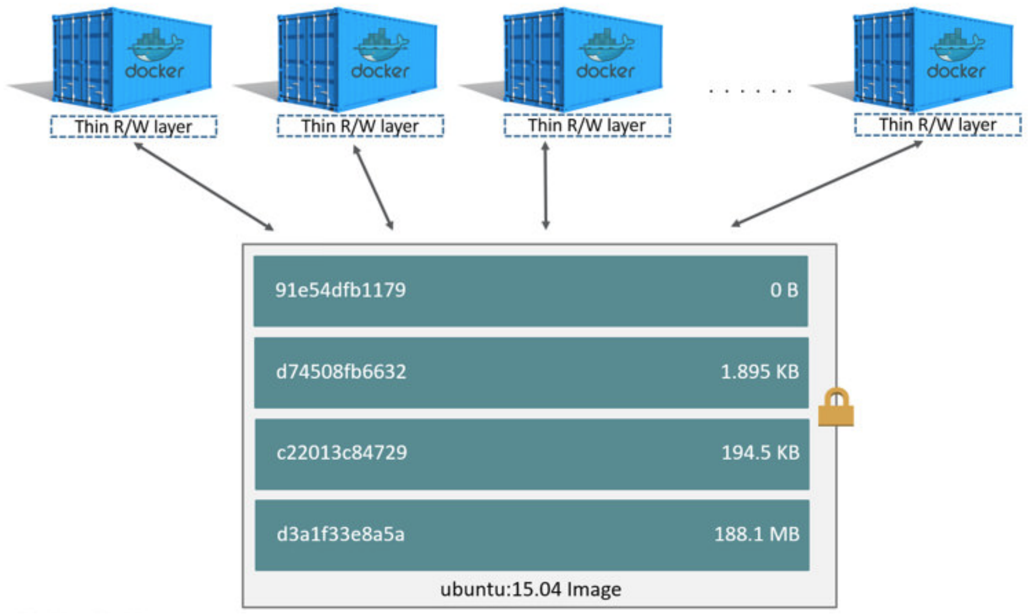

Figure 2.6: Multiple Docker containers using the same base image [12].

- *Docker Services* allows scaling of containers across multiple Docker daemons[11]. They work together as a swarm with multiple managers and workers. Each member of the swarm is a Docker daemon and they communicate with each other using the well-defined APIs. The desired number of replicas can be defined in the service and it takes care of load balancing. For a Docker user, it stills give the perception of a single application.

[11]https://docs.docker.com/engine/docker-overview/#docker-objects

# Chapter 3

# Microservices and container orchestration

This chapter provides the necessary background on microservice architecture and container orchestration frameworks. Microservice architecture enables designing of applications as a collection of loosely coupled independent services in contrast to the traditional monolithic architecture. Containers are used to encapsulate these individual services with good isolation. Section 3.1 introduces monolithic applications and its draw backs. Section 3.2 describes microservice architecture and its advantages. Section 3.3 discusses how Docker containers can be used to leverage the benefits of microservice architecture. Finally, Section 3.4 provides information about the need for container orchestration and describes different container orchestration frameworks.

## 3.1 Monolithic application

Software applications were traditionally developed using a monolithic architecture. This approach consists of a single tier wherein different functions and software components are combined into a single program (Figure 3.1). For example, user interface, data processing, business logic and data storage functionalities are all interwoven to create a large self-sufficient single application [44]. The main issue with this approach is the poor support for continuous integration, continuous delivery and scaling requirements. In the cloud era, one of the main advantages is that the application can be scaled according to actual demand. However, monolithic applications cannot fully utilize these benefits. To scale a monolithic application the entire application needs to be scaled. This results in more utilization of processing, storage and other computing resources, even when only perhaps one component of the application needs to be scaled.

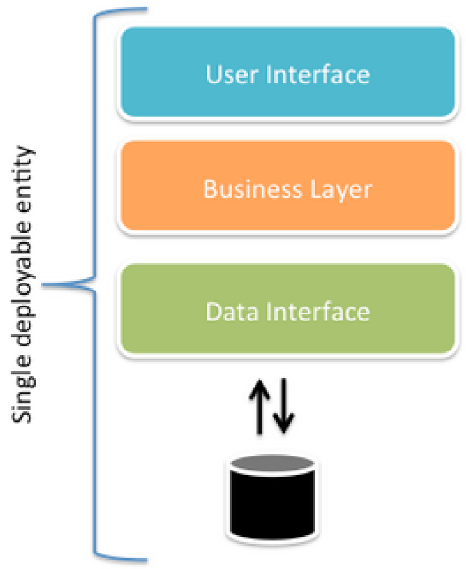

Figure 3.1: Monolithic architecture [19].

There are several other drawbacks with the monolithic approach. This approach is challenging for incremental software development processes such as Agile which emphasizes on shorter delivery cycles. Even a small change (in one part of the application) impacts on large portions of the remaining application code and results in “cascading effect” and requiring longer testing and release cycles [36]. For instance, the change in one part of the code requires the whole application to be rebuilt and installed for testing purposes. The entire software application needs to be in one specific technology or programming language. Such an approach does not allow to use the strong features of different programming languages. For instance, Python is most suited for data processing, whereas Golang is suitable for business logic. However, in a monolithic approach, the entire application has to be implemented in a single programming language regardless of its suitability for a particular component. This limits the freedom of choice for developers [45].

## 3.2 Microservices

The term “microservice” was first coined in May 2011 during a workshop of software architects held in Venice [47]. Since then microservice architecture has been gaining a lot of attention and interest among software developers. Microservice architecture structures the application as a collection of loosely coupled services. It relies on modular, independent software entities that are highly scalable. Dependencies are decoupled into logical entities and functional concerns are separated. Each single modular software entity addresses a single concern. Each microservice has well defined interfaces through which other entities can communicate with the service. This enables each service to change its internal implementation as long as the interface is compatible with the rest of the services. Figure 3.2, shows the differences between the monolithic and microservice based architectures.

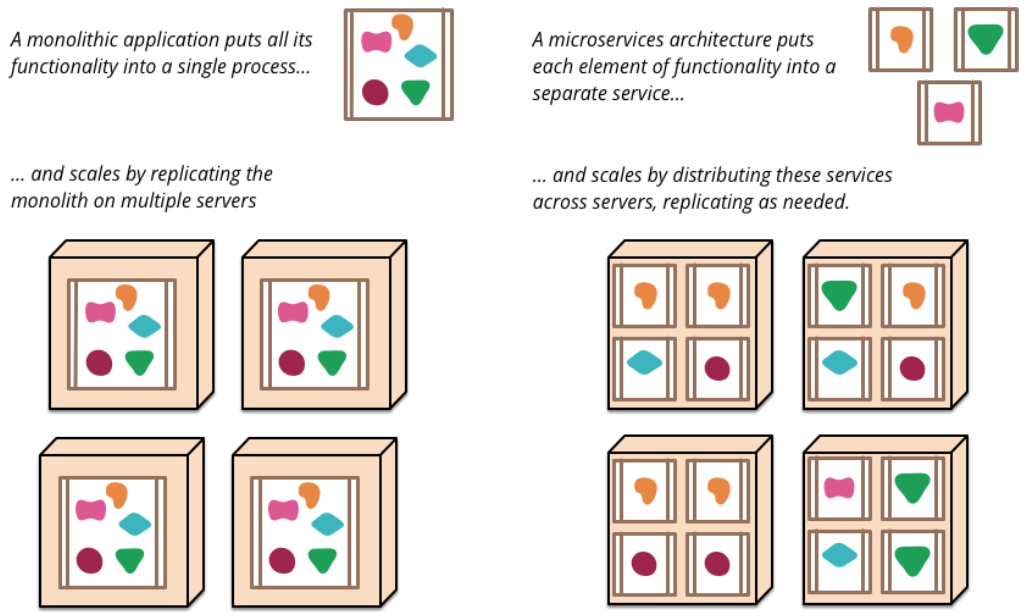
A monolithic application puts all its functionality into a single process...
A microservices architecture puts each element of functionality into a separate service...
... and scales by replicating the monolith on multiple servers
... and scales by distributing these services across servers, replicating as needed.

Figure 3.2: Monoliths and microservices [47].

The design philosophy of the microservices architecture is a simple one, “Do one thing well”. The microservice architecture should adhere to the following principles:

- Keep the services small such that they each address only a single concern [58]

- Any software development organization must adopt and embrace automation for software testing, deployment and delivery. This enables the development and testing teams to works on small individual deployable units of software [57].
- The individual services must be [20]:
  - *Resilient:* A single service should fail without impacting other services.
  - *Elastic:* Service should be able to scale up or down independently.
  - *Complete:* The service should be functionally complete.
  - *Minimal:* The given service should perform a single business function and must contain highly cohesive units.
  - *Composable:* Service should have well defined uniform interfaces.

Microservices have been recognized as a reliable software solution to develop and scale large software products [68]. Microservice architecture is usually best suited for Continuous Integration and Continuous Delivery [32]. Its modular approach eases software development and release cycles, thereby reducing the time to market. DevOps (Development and Operations) is a set of practices intended to minimize the time between committing a change to the system and the change being applied in production with high quality [35]. Figure 3.3 shows the increasing trends of DevOps and microservices according to Google keywords search index. This trend clearly indicates the growing importance of microservice architecture with respect to DevOps. Several large organizations such as Netflix have embraced the microservice architecture [59] and have successfully deployed large web scale products. Each microservice can be developed in a different programming language and framework, thereby allowing software developers the freedom of choice. This also gives the added benefit of choosing the right programming language and framework for the desired functionality. The microservices communicate with each other using well defined interfaces and communication methods such as REST API or message passing. It is much easier to accommodate changes and new feature requests, as only the affected microservice needs to be modified keeping the interface definitions consistent. Scaling up with microservices architecture is easy and efficient. Instead of scaling the whole application or product as in the case of monolith applications, only the desired services are scaled and replicated. This results in cost savings for cloud-based applications as the cloud computational resources are billed according to their usage. For these reasons microservices architecture is usually known as a cloud ready architecture [32].

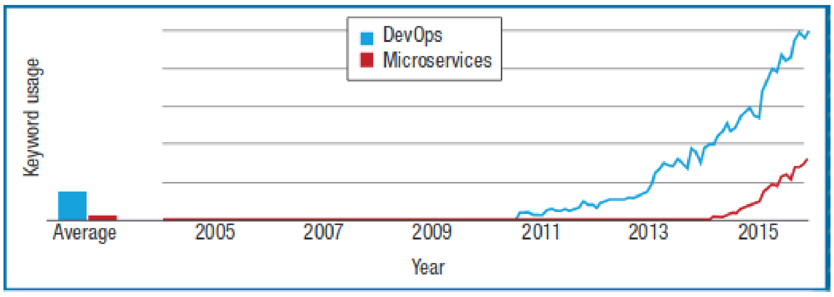

Figure 3.3: The increase in the use of the keywords DevOps and microservices, according to a Google Trends report [32].

Microservice is no silver bullet, it comes with its own limitations and constraints. Microservice architecture introduces complexity with respect to network latency, network communication, load balancing, fault tolerance, message formats [62] [60]. The distributed nature of the architecture introduces complexity during debugging and error isolation [55]. Network communications are prone to failures and microservices depend heavily on network communication for inter-microservice communication. Information flow between the different services could be a problem, since each microservice addresses a single concern they could become information barriers when trying to get a complete view of the system internals. A microservice depending on another microservice has to block itself until the dependent microservice is ready. Usually, microservices have individual owners (developers) and communication gaps between different developers adds to some additional delay [10]. A congested network or poor bandwidth can have a profound impact on the performance of the product, since most of the functionality depends on inter microservice communication. An in-memory call in monolith application is always faster when compared to microservice architecture that relies on network communication [45]. Even though addition of new services is perceived as an easier task, maintaining different microservices with different technology stacks can be challenging. From an organizational management point of view, it can end up requiring several individuals with specific technology expertise to handle and maintain the different services which could increase the cost. Unit testing and individual service level testing could be done comprehensively with this architecture. The complexity of building and maintaining integration and system test suites grows as the number of services grow. As the number of services increases within the system, the system state is also shared among several individual services. All of these services need to work in synchronization to maintain the collective state of the system. This leads to more errors since one misbehaving service could disrupt the harmony of the entire application [45].

## 3.3 Docker containers for microservices

Docker container is ideal for implementing the microservices. Docker enables running of applications in an isolated manner within a container, effectively each service could be run inside separate container. The containers could be on the same machine or on a different machine. Docker has several features like image repository that enables upgrading and downgrading of applications in a trusted manner. Managing the Docker containers is eased by the Docker client. Docker client can communicate with more than one daemon and enables users to monitor and manage containers across hosts [6]. Docker addresses most of the challenges faced by microservice architecture [55] such as:

- *Automation:* Creation and launching of Docker containers is scriptable and this drives automation.
- *Independent:* Docker containers are self-contained with the application and its required run time environment. This gives the freedom for developers to choose any technology stack they deem fit for the desired functionality.
- *Portable:* Docker containers are portable and allows testing of individual microservices. The containers could be easily ported across different machines and environments, this ensures that the development and production environments are the same [18].
- *Resource utilization:* Docker containers contain a minimal set of artifacts needed for the application to run, while sharing the kernel and other resources from the host operating system with other containers. This maximizes resource utilization.
- *Support and popularity* Dockers are widely adopted and supported by multiple different platforms and this enables development teams to choose a technology they seem best fit for the given the microservice and start developing it.

Docker has several open source tools built around it to facilitate rapid deployment, automation and ease of maintenance of microservices such as

Docker registry, Dockerfile and Docker client. The Dockerfile enables automation and ease of deployment of the microservice with its image managed by Docker registry [55]. The Docker client is used to communicate and control the microservice (container) and enables in logging and monitoring of the service. Docker containers enable rapid automated deployment and testing of the application. Automated deployment and testing are the building blocks of modern CI/CD pipeline. Thus, Docker containers have become the enabling technology for modern CI/CD. Popular CI/CD tools like like Jenkins[1] and Gitlab[2] support Docker. These features have made Docker containers popular and well suited for implementing the microservice architecture.

## 3.4 Container orchestration

Docker containers have become the standard way to build modern scalable applications. However, the microservice architecture requires an increasing number of containers for the growing number of microservices. This raises several challenges in managing the the containers. The distributed nature of the containers further complicates their management. Deploying these distributed microservices requires many manual intervention steps by the system administrators [71]. Automating these steps results in long scripts that grow in complexity and are hard to maintain. Thus, there is a growing need for frameworks and tools to manage the containers. Container management tools and frameworks satisfy the below requirements:

- *Automated deployment* of containers both in test and production environments.
- *Scaling up and scaling down* of containers are required as per the traffic patterns.
- *Resilient:* Containers should be resilient and always available for the application to work as expected [8].

Container orchestration frameworks provide many features, including provisioning of hosts, instantiating containers, stopping containers, rescheduling failed containers, linking containers though their agreed interfaces, enabling resilience, exposing the containers outside of their cluster in a secure way, scaling out and scaling down containers, updating the images in the containers. Kubernetes, Docker Swarm and Nomad are the most popular container orchestration frameworks available now.

[1]https://jenkins.io/solutions/docker
[2]https://docs.gitlab.com/omnibus/docker

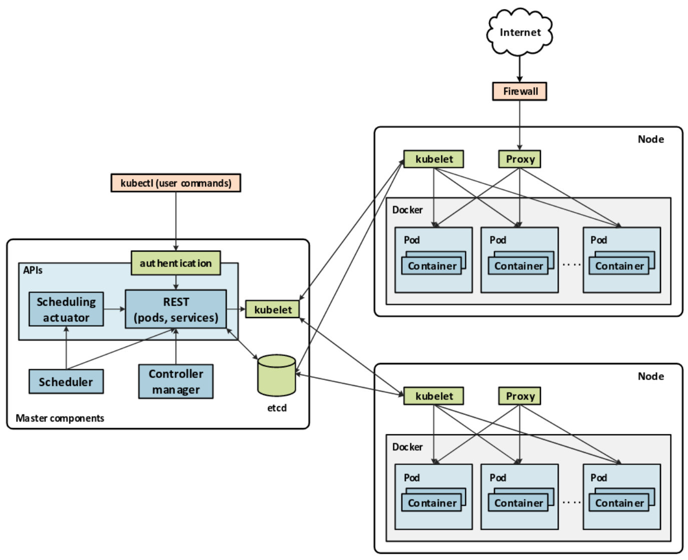

Figure 3.4: Kubernetes architecture.

## 3.4.1 Kubernetes

Kubernetes is an extensible open-source platform for managing containerized workloads and services. It supports both declarative configuration and automation [28]. Kubernetes is built on top of time tested Google's internal cluster management system called Borg [73]. The Borg system has been used by Google to run containers in its data centers for more than a decade. It is a cluster management system that can run hundreds of thousands of jobs from different applications across many clusters [73]. Kubernetes was open-sourced in 2014 and since then has been the most popular container management and orchestration system.

The design philosophy of Kubernetes is to create an ecosystem of various components and tools that would enable deploying, scaling and managing applications with ease. Kubernetes enables ease of portability of applications across different environments and provides a container-centric management platform.

### Kubernetes Architecture

Figure 3.4 shows the high-level architecture of Kubernetes. We explain them in detail next.

- *The* Pod[3] is the smallest and simplest unit that can be created and deployed in the Kubernetes object model. Pods are considered as the basic building blocks of Kubernetes. In simple terms, a pod represents a running process on the cluster. A Pod can contain a single container or a small set of containers that share common resources like network and storage. Even though the most common use case is one container per pod, there may be requirements for multiple containers to be tightly coupled in a single pod. Containers inside a single pod are always co-located either in the same host or virtual machine. Containers inside a pod communicate with each other using the localhost and communicate outside the pod using the shared network resources such as ports. Each pod is assigned a unique IP address that is used to communicate with the external world and other pods. Kubernetes always schedules and orchestrates the pods and not the containers. So, when an application needs to scale, it translates to adding more pods. Adding pods is referred as "replication" in Kubernetes. Pods are ephemeral entities and they can be created and destroyed at will. Pods can be terminated automatically for various reasons such as lack of resources, hence it is important to maintain statelessness in the pods. Pods themselves do not run, rather it is an environment where containers run.

- *Node*, also known as, minion can either be a physical or a virtual machine. Nodes provide the run time environment for the pods and are managed by the master components. The important components of a node are kubelet and kube-proxy.

  - *Kubelet* is the prominent controller in Kubernetes and drives the container execution layer. It implements the pod and node APIs. Kubelet takes a set of PodSpecs (YAML or JSON representation that describes the pod) and ensures that the containers mentioned in the PodSpec specification are running without any issues. It should be noted that kubelet does not manage containers that are created outside the Kubernetes environment. The main responsibilities of Kubelet include creation, continuous monitoring and graceful termination of containers running on the node and status reporting of the nodes to the cluster [64]. The PodSpec is usually provided to the Kubelet using the API server. There are three different options other than the API server for providing the PodSpec to the Kubelet: path to a file from the command line, HTTP endpoint as a parameter in the command line, and finally, Kubelet

[3]https://kubernetes.io/docs/concepts/workloads/pods/pod/

  can listen for HTTP requests and request for a PodSpec. The refresh intervals for requesting a new a PodSpec can be configured from the command line[4].

  - *Kube-proxy* is run on each node and programs the IP tables. This enables the service traffic to be redirected to the correct backends. This additionally provides a highly-available load balancing solution and has a low performance overhead[5]. For example, a service request originating from a node is best served within the same node. It basically maps the individual containers to a single service. Kube-proxy works in the transport layer level and can do a simple TCP or UDP stream forwarding. It can also perform a round-robin based forwarding to achieve load balancing[6]. Service end points are resolved and discovered using DNS.

- *Master* components serve as the control plane for a Kubernetes cluster. A cluster refers to a collection of node(s) and master(s). The individual master components can be run on any machine in the cluster. For simplicity and easy management of the cluster, the master components are all started in the same machine. User containers are not scheduled to run in the master machines. Kube-apiserver, etcd, kube-scheduler, kube-controller-manager are some of the main master components that are described below [16].

  - *kube-apiserver* is the front-end of the Kubernetes control pane and exposes the Kubernetes API. It is a simple server which mainly processes REST operations and updates the corresponding objects in etcd and other stores[7]. The API server acts as a gateway to the cluster providing access to clients that are outside the cluster. The API sever is horizontally scalable. Command line tools such as Kubectl and Kubeadm use the API server to communicate and control the cluster and the objects contained in it. API server is responsible for all communication between the master and the nodes [64]. It should be noted that Kubernetes does not support atomic transactions that result in multiple resources being updated.
  - *etcd* stores all the cluster data including the state of the cluster. It is a distributed, highly-available and consistent key-value store

[4]https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/
[5]https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/
[6]https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/
[7]https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/

which provides reliable access to critical data[8]. It provides watch support with which multiple components can be notified of any changes to the cluster state in a timely manner.

- *kube-scheduler* is responsible for assigning nodes to pods. It watches for the newly created pods that do not have a node assigned and selects a node for the pod to run. The availability of requested resources, hardware constraints, software constraints, data locality, affinity and anti-affinity requirements are some of the factors considered by the scheduler to select a node for the pod[9]. The scheduler is also responsible for evicting a pod from a node taking into account the current state of the cluster.
- *kube-controller-manager* runs the controllers. It is a control loop that watches the shared state of the cluster using the kube-apiserver and makes appropriate changes to the components to move the current state to the desired state. Even though each controller could be treated as an individual process, in order to reduce complexity they are all combined as one single process. Some of the controllers are described below [16].
    * *Node controller* is responsible for managing the nodes. It watches each node and responds when a node goes down.
    * *Replication controller* maintains the desired number of pods.
    * *Endpoints controller* links services to pods. It populates the Endpoints object.
    * *Service account and token controllers* are responsible for creation of default accounts and API access tokens for new namespaces.
- *cloud-controller-manager* runs controllers that interact with the underlying cloud providers[10]. It is relatively new feature that is in alpha state released with Kubernetes version 1.6 [15]. The cloud-controller-manager daemon runs the cloud-specific control loops. Even though this feature was previously present inside the control manager, the development cycles of Kubernetes and cloud vendors were not the same which led to creation of this separate binary. With this separation the cloud vendors could abstract their cloud specific code. Most of the popular cloud vendors such as Amazon and Google have implemented their own cloud-controller-manager.

[8]https://github.com/coreos/etcd

[9]https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/

[10]https://kubernetes.io/docs/concepts/overview/components/

- *Replication controller* is responsible for ensuring that a single pod or a homogeneous set of pods are up and always available. Set of pods is called a replica. Replicas are needed for scalability i.e., during horizontal scaling more pods are added to handle the traffic and more than one replica is needed for high availability. ReplicationController supervises multiple pods across multiple nodes[11]. Pods that are monitored and maintained by the replication controller are automatically replaced with new ones if some pods fail. Pods are ephemeral and can be deleted or terminated for various reasons. This is one of the strong compelling factors to use *replication controller* even if the application requires only a single pod. The replication controller uses pod templates (containing a set of rules) to manage the pods. Replication controller is also responsible for deleting the excess pods that may be present in the cluster. The controller can be terminated along with the pods managed by it or the controller can be terminated without terminating the pods and pods can be disassociated from the controller.

- *Services* are an abstraction that defines a logical set of pods and policies to access them. As seen earlier pods are ephemeral and replication controller can evict or delete or replace them at any point of time. Thus, the IP address associated to a pod is not fixed. These factors complicate communication with the pods. This problem is solved with the service abstraction where pods that provide the same services are grouped using the label selector[12]. In Kubernetes, services are REST objects like pods and RESTful operations could be performed on them. Both TCP and UDP are supported protocols for services with TCP being the default. The service object is updated when the associated pod(s) change. Service endpoint object is created when the service is created in Kubernetes and contains the details of the pods. The main goal of the service is to ensure that the pods are accessible to its users all the time.

- *Addons* are pods and services that implement the cluster features [16]. They extend the functionality of Kubernetes. There are several addons available for features like: networking and defining the network policy, service discovery, and visualization and control. WeaveScope[13] is one such addon for graphically visualizing the containers, pods, services and

[11]https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/
[12]https://kubernetes.io/docs/concepts/services-networking/service
[13]https://www.weave.works/docs/scope/latest/introducing/

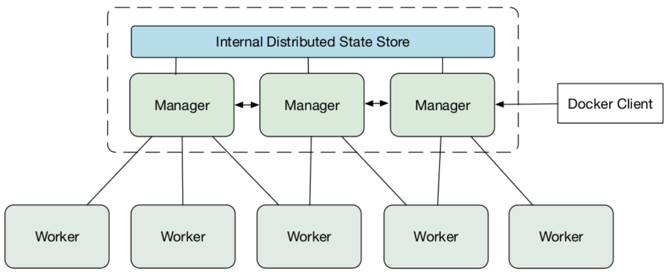

Figure 3.5: Docker Swarm architecture [33].

other Kubernetes objects. There is also a Dashboard[14] addon which provides a web interface for visualizing and monitoring the Kubernetes cluster.

### 3.4.2 Docker Swarm

Docker Swarm is a cluster management and orchestration framework embedded in the Docker Engine from v1.12. It is built using the swarmkit, a Docker orchestration layer. A swarm is made up of multiple Docker hosts that are running in swarm mode. A Docker host (node) can be a manager, worker or both. Docker swarm abstracts many of the details such as network communication between the containers in the given cluster and it is relatively easy to setup. The commands necessary to deploy a container in the swarm is similar to the commands required to deploy the containers in standalone mode. This is due to the fact that both modes use the Docker engine APIs and CLIs. This enables a swarm to setup easily. The framework supports and abstracts seamless network communication between the different containers. Docker Swarm also supports service discovery by name and provides basic load balancing features.

#### Docker Swarm Architecture

Figure 3.5 shows the high-level architecture of the Docker Swarm, the components of which are explained in detail below.

- *Node* is a host participating in a swarm. As explained previously nodes can take up either the manager or worker roles or both of them. Nodes

[14]https://github.com/kubernetes/dashboard

can be run on the same physical machine or across multiple servers which is usually the case[15]. For an application to be deployed in swarm a corresponding service definition must be submitted to the manager node. The manager then dispatches the tasks to the worker nodes. Manager nodes performs functions to maintain the desired state of the swarm. Their main functionalities include cluster management and orchestration. Worker nodes execute the tasks dispatched by the master nodes. Each worker node has a running agent, which reports the status of the tasks assigned to the corresponding worker node. The worker node constantly notifies and updates the manager node regarding the status of the tasks being executed in it. Manager nodes can also run services as the worker node or they can be configured to explicitly perform management related tasks alone.

As shown in the Figure 3.5, there could be more than one manager node in each swarm. They exchange control information among themselves to maintain a consolidated view of the cluster. The node which executes the command to initiate the swarm becomes the main manager or the leader of the swarm. Any master node can replace a failed manager. Container orchestration is performed by the scheduler that runs on the leader [11]. Nodes have three different availability statuses: *Active* state indicates that the nodes are available and containers can be scheduled to run in them; *Pause* state indicates that nodes cannot take in more containers, but the containers that are running inside the given node can continue to run; *Drain* state indicates that no containers can be scheduled on them and the containers that are executing inside the nodes are terminated and relaunched in other nodes. User defined labels can be part of node, which can be used to schedule certain containers on the given node. Labels are also used to group a set of containers to a specific node.

- *Services and tasks* are the fundamental building blocks of Docker Swarm. Services are used to deploy the applications on a swarm. Service is the primary mode of interaction for a user with the swarm. The service defines the set of tasks that are to be executed by the nodes. Service specification specifies the container image to be used and a list of commands that are to be executed inside the container. It also contains parameters such as the number of replicas, container placement and the set of ports to be exposed. The two types of service deployment modes are *global* where the task is scheduled on every available node

[15]https://docs.docker.com/engine/swarm/key-concepts/

of the cluster and *replicated* mode where the number of tasks are specified in the service definition and the leader schedules them across the swarm. It should be noted that in the global mode, each node has exactly one container allocated to execute the task. On the other hand, in the replicated mode, a given node can have more than one container executing the same task. Moreover, nodes are chosen based on any container placement criterion defined in the service specification.

*Tasks* are the atomic scheduling units of swarm. Tasks are execution units that contain a set of commands that are to be executed inside a container. Tasks get assigned to nodes and run on them until the desired state is reached [11]. Once a task is assigned to a specific node it cannot be moved to another node. Manager nodes create and assign tasks based on the number of replicas specified in the service definition. Tasks move from one state to another until they complete or fail. Tasks are initialized with *new* state move towards the *completed* state. A task that has finished execution (completed state) will not be run again, but a new task can take its place[16].

### Service Discovery

Service discovery refers to the process of how the service requests are routed to the appropriate containers. Containers are ephemeral and their associated IP addresses can change making it challenging to manage communication manually. In the swarm, service discovery is network-scoped, meaning containers belonging to the same network can resolve addresses among them. The Docker engine acts as DNS server, resolving the DNS queries (service names) sent to it by individual containers.

If the query cannot be resolved by the Docker engine, the queries are forwarded to the configured default DNS server. The swarm further supports round robin mode and Virtual IP (VIP) mode to resolve DNS queries. In the round robin mode, the query results contain addresses of all the containers executing the service and the service request could be sent to any of the addresses[17]. In the VIP mode, the response is the virtual IP address of the service. For requests originating outside the swarm, the mechanism still remains the same, IP address and port numbers are used in the query rather than the service names.

[16]https://docs.docker.com/engine/swarm/how-swarm-mode-works/swarm-task-states/

[17]https://docs.docker.com/docker-cloud/apps/service-links/#hostnames-vs-service-links

#### Load Balancing

Load balancing is supported in swarm. The manager nodes use internal load balancing to evenly distribute the requests among the services within the same cluster[18]. As discussed in the previous section, when using the round robin mode, the DNS query response contains a list of IP addresses corresponding to the service. It is then left to the client to effectively load balance the requests. Client-side load balancing is not preferred as it does not guarantee uniform load balancing. Client side applications usually use caching and can have application specific design issues that might result in poor load balancing. The VIP mode inherently guarantees effective load balancing, since the load balancing is done at the server end. The other advantage is that the actual containers are well abstracted behind a VIP. The VIP of the service will remain the same immaterial of what happens during container orchestration and management.

### 3.4.3 Nomad

Nomad is an open source scheduler for scheduling containers and standalone applications. Nomad is developed by HashiCorp. It is a tool for managing a cluster and the applications running on them. Nomad uses a declarative job file for scheduling the applications by abstracting the machines and location of the applications. Nomad handles where and how the applications will be run [13]. Nomad supports Docker containers. It uses the Docker driver to deploy the containers in the cluster. The number of required instances of the containers can be specified in the job. Nomad ensures that the specified number of containers are running and recovers from any failures automatically. Nomad is lightweight and is shipped as a single binary. It encompasses the features of resource manager and scheduler into a single system. Nomad provides flexible workload to the users by enabling them to start Docker containers, virtual machines or any application run time. Nomad is built for scaling and it can scale to cluster sizes which have more than ten thousand nodes[19]. Nomad models the infrastructure as a group of data centers which constitute a larger region, thus enabling Nomad to be Multi-Datacenter and Multi-Region aware. Several regions could federate among themselves allowing jobs to be registered globally [13]. Nomad provides a platform for managing the microservices and enables organizations to adopt to the microservice architecture paradigm.

[18]https://docs.docker.com/engine/swarm/key-concepts/#load-balancing
[19]https://www.nomadproject.io/intro/index.html

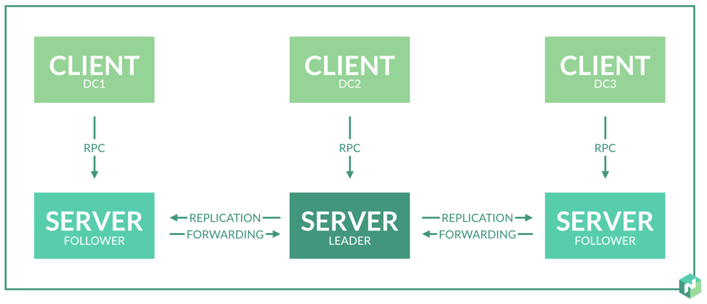

Figure 3.6: Nomad architecture [2].

### Nomad Architecture

Figure 3.6 shows the high-level architecture of Nomad for a single region [2]. The following sections describes the components in detail.

- *Task* is the smallest unit of work in Nomad. Drivers are static binaries like Docker that are the means of executing the tasks. Task specifies the required driver, constraints and resources. Tasks are executed by the corresponding drivers. A *Task Group* is a group of tasks that are scheduled to run in the same node and cannot be split.
- *Job* is composed of one or more task groups. It is a specification provided by the user expressing the desired state. Nomad works towards maintaining the desired state of the job.
- *Clients* are machines where tasks are executed. A Nomad agent runs on each client which is responsible for communicating with the servers and listening for tasks to be assigned.
- *Evaluation* is the process by which scheduling decisions are made by Nomad. Nomad continuously monitors the state of the job and if a change is required it makes a new evaluation. A new evaluation could result in new allocations. The mapping between the task groups in the job and the client machines is termed as allocation.
- *Servers* are responsible for managing the jobs, clients, creating task allocations and performing evaluations. There are a set of servers per region. The data is replicated between servers in the same region and can federate across different regions to make them globally aware. A leader is elected among the servers and servers are highly available.

- *Bin packing* is the process of packing tasks to the client machines. Nomad automatically applies the job anti-affinity rules which reduces collocation of multiple instances of the same task group. This ensures high resource utilization of the clients.

As shown in the Figure 3.6, each region consists of a set of servers and clients. Clients communicate with their respective regional servers using Remote Procedure Calls (RPC) as seen in the Figure 3.6. During the registration clients provide information regarding "themselves", such as a list of drivers, resources available and other system attributes that are used by the servers to make scheduling decisions. Clients also send periodical keep alive messages to the server to indicate their availability.

Servers are responsible for accepting the jobs from users and scheduling and executing them on the client machines. All the servers within a given region are a part of a single consensus group. They elect a single leader, which is responsible for processing all the queries and executing all the transactions. All servers make scheduling decisions in parallel and the master ensures that there is a uniform distribution of requests and no client is over loaded. It is always necessary to have more than two servers to ensure high availability. More the number of servers, the time to reach consensus would be more. However, there is no limitation on the number of clients.

Nomad CLI or APIs are used by the users to submit jobs to the servers. Nomad uses Consul[20] (also developed by HashiCorp) as a service discovery tool. Consul needs to be installed alongside Nomad in the client machines and scheduled to run. DNSMasq[21] is used by Nomad to intercept the requests and feed to a local load balancer such as HAProxy[22]. This enables services to be requested by the service name, which get forwarded internally to the correct IP address and port number. Users need not worry about the dynamic nature of containers and address them with service names instead. Nomad provides a web- based user interface to monitor and examine the cluster. The Nomad CLI can be used to communicate with a remote Nomad cluster as well. This allows remote administration of different Nomad clusters from a single machine.

[20]https://www.consul.io/
[21]http://www.thekelleys.org.uk/dnsmasq/doc.html
[22]http://www.haproxy.org/

# Chapter 4

# Access management and PrivX

The chapter presents the general concepts of access management and describes PrivX architecture in detail. The goal of this chapter is to identify a suitable container orchestration framework for PrivX. Section 4.1 outlines access management. Section 4.2 describes PrivX and the desired characteristics of the container orchestrator.

## 4.1 Access management

Access management is a broad area that comprises of methodologies, tools, processes and policies to maintain and comply with access privileges. It is the process of monitoring, managing and identifying authorized access to resources such as hosts, applications and other information technology resources. It enforces the appropriate up to date access policy for every login attempt to access the resources. Access management is challenging in the cloud environment. The infrastructure is no longer located on the premise but could be located across the globe. Traditional access management software does not scale well for the cloud as it was originally built for more static on-premise environments. Infrastructure as a Service (IaaS) model is the most popular offering from cloud vendors, allowing enterprises to host their infrastructure in the cloud and scale at will. With cloud computing, organizations have large computing resources at their disposal. Enterprises scale up when there is a surge in the traffic and scale down when the traffic is low. For example, during the launch of a new product, the surge in the traffic would be huge and organizations need to add more servers to handle the demand. Once the initial bookings and sales are done the traffic pattern returns to normal. This requires access to a few hundreds of servers in a short time and revoking the access after the traffic has become normal. Servers require access to initially provision and configure them to handle requests. The access needs to be revoked when the need is no more, this is challenging and time consuming.

An access control model defines relationships among permissions, operations, objects, and subjects [1]. Role based access control (RBAC) is a popular access control model [65]. It provides access control based on the user's roles and privileges. Let's take an example of an organization that needs to manage its database servers. The organization has a test and production environment. For such a scenario, two roles, "Test-DBA" and "Prod-DBA", are sufficient to enforce access control. As the names imply, a user with "Test-DBA" role has access only to the test database servers and similarly "Prod-DBA" role for production database servers. This simple philosophy makes RBAC one of the most popular and widely used access control mechanisms. It is also well suited to a distributed environment and restricting access in cloud environment [43].

PAM helps organizations to securely provide privileged access to its assets and conform with any compliance requirements by securing, monitoring and managing the privileged accounts and their access [29]. Some of the common features that all PAM solutions offer [41] are:

- *Automatically discover all user (privileged) accounts across the enterprise*
- *Establish and enforce custom work flows to gain privileged access*
- *Securely store privileged credentials*
- *Record and monitor privileged access for audits*
- *Automatically rotate passwords*

PAM solutions should be easy to integrate with an existing organization's infrastructure and should improve their efficiency by reducing the manual steps involved in provisioning and access management. So, any PAM solution should integrate well with the cloud and on-premise infrastructure and should ideally provide a one stop solution for organizations.

## 4.2 PrivX

PrivX is a state of the art PAM solution developed by SSH Communications and Security Oyj. PrivX uses RBAC and enables privileged access management. PrivX uses short-term credentials to provide just-in-time access to resources. Access is granted based on the user's role. Most of the traditional PAM solutions rely on password vaulting, password rotation and additional client and server pieces of software needed to be installed. PrivX requires no additional software and creates credentials that are not persisted anywhere enabling faster and safe access to resources. PirvX implements certificate based authentication to authenticate a user against a target server.

Certificate based authentication allows secure access by exchanging a digital certificate instead of a username or password. This prevents the common security threats in the password based authentication such as keyboard logging or Man In The Middle (MITM) attacks. Certificate based authentication can work well in the heterogeneous environment provided the certificate is created with the username and the corresponding access groups to which the users belong. This information enables the target servers to decide on the access levels and restrictions for the user [42]. PrivX employs a similar mechanism, wherein the username and the corresponding roles for the user are included inside the certificate information enabling the right access to resources in the target server.

Identity access management is one of the key security challenges in the cloud computing environment [70]. Services are provisioned dynamically in the cloud and same cloud instances could be reused to provision different services and applications. Access granted for a particular user to a target machine using traditional methods such as passwords or SSH keys do not address the security challenge of the user having access to a server that runs a different application or service from the time he was originally granted access with [54]. Users access should be revoked immediately when the server is used for a different purpose. This is achieved with RBAC, where the access is controlled based on the users role which prevents access to the newly deployed application. PirvX uses RBAC with short lived certificates for authenticating the users, this means that the user cannot have access to the target servers on a permanent basis. Moreover, PrivX validates the users role for every login, preventing the user to get access to target servers that have their roles changed.

One of the other major security issue with cloud applications is 'Insecure APIs' [69]. APIs are used by the users for authenticating, provisioning and carrying out all their necessary tasks. APIs must be designed to circumvent any potential misuse [69]. PrivX uses role based access restrictions at API level to verify that the user or service invoking the API has sufficient privileges to do so. This prevents any potential misuse of the public APIs exposed by PrivX. Moreover, the native cloud access mechanisms and tools do not provide extensive auditing [51]. Auditing the access to a cloud resource is important in terms of security and compliance requirements. PrivX provides extensive auditing of the access to cloud resources and helps in compliance requirements of the organization.

Privileged access management in cloud is challenging attributing to its dynamic nature. Some of the issues with PAM in the cloud era are [25]:

- *Request approvals* takes a long time with traditional solutions. The dynamic nature of the cloud means that there could be a sudden increase in the number of servers. Handling approvals for such scenarios can be time consuming. This slows DevOps and reduces productive time.
- *Web of roles* has become too complex. Organizations tend to create many roles over time and maintaining them becomes complex. Manually mapping roles has become untenable with current infrastructure such as Active Directory (AD).
- *Vaulted credentials* are a single point of failure and are a common target for hackers. With other third-party software integrations it only gets more complex and hard to maintain.
- *Slow and expensive* for the cloud era. PAM solutions do not scale well at business speed. They are not compatible with the Agile development methodologies.

PrivX eliminates most of the issues mentioned above. As shown in the Figure 4.1, PrivX scans the cloud environments from all regions and updates the hosts automatically. All the hosts and users are up to date and access is provided based on the user's role. There is no requirement for manual access provisioning. PrivX does not require any additional software both on client and server side and this reduces maintenance costs and disruptions to the infrastructure. PrivX saves a lot of time for system and infrastructure administrators by providing a single point solution to access, grant and monitor users access to their resources.

### 4.2.1 PrivX features

PrivX is designed with a philosophy to support Agile teams that require to scale their cloud and on-premise infrastructure cost effectively. Some of the notable features of PrivX are:

- *Dynamic role-based access* is used to check the role and identity for each connection. PrivX can easily integrate with user directories such as AD to keep its user data up-to-date. Role requests and approvals are streamlined and administrators can have full control over server access.

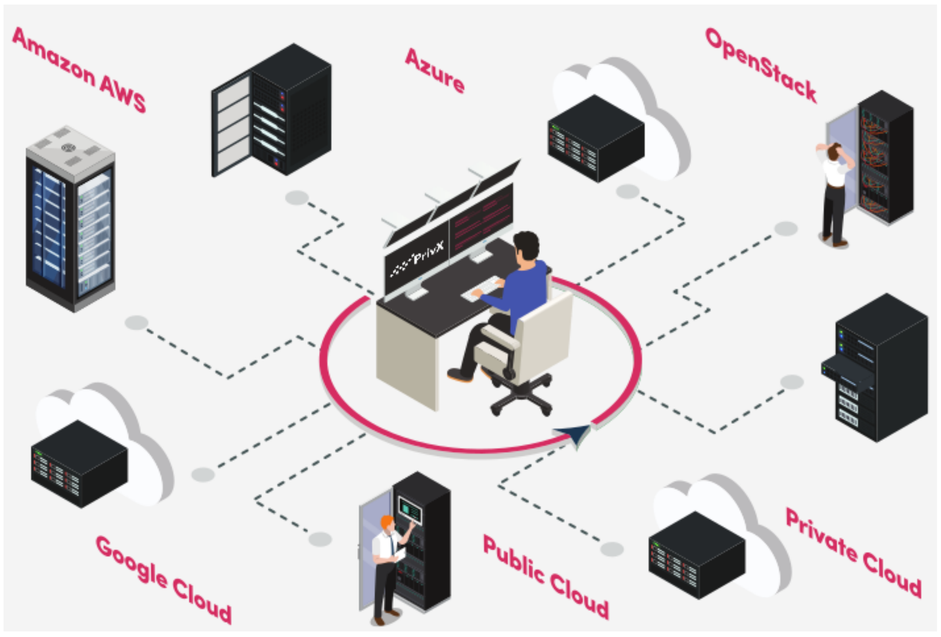

Figure 4.1: PrivX - Managing different cloud and on-premise entities [25].

- *Rapid easy deployment* with PrivX. PrivX does not require any additional software on the target hosts. SSH and RDP connectivity provided via the web browser eliminates the need for any custom or third-party clients.

- *Automatic host detection and configuration* is supported in PrivX. PrivX can discover the hosts from across cloud regions and on-board them with minimal effort. The added advantage is that PrivX continuously scans and update the environment in cloud, it can detect new hosts added and the ones that have been removed or terminated automatically.

- *Password less authentication* is a unique feature of PrivX. Maintaining passwords is always a hassle and usually requires passwords to be periodically changed.

- *Web-based SSH terminal* is included in PrivX which eliminates the need for SSH clients.

### 4.2.2 PrivX architecture

PrivX is based on the microservice architecture. The different microservices communicate over HTTPS using REST. Figure 4.2 shows the high-level architecture of PrivX. The circled services in yellow in Figure 4.2 indicate the services that we implemented as a part of this thesis. The role of each microservice is described below.

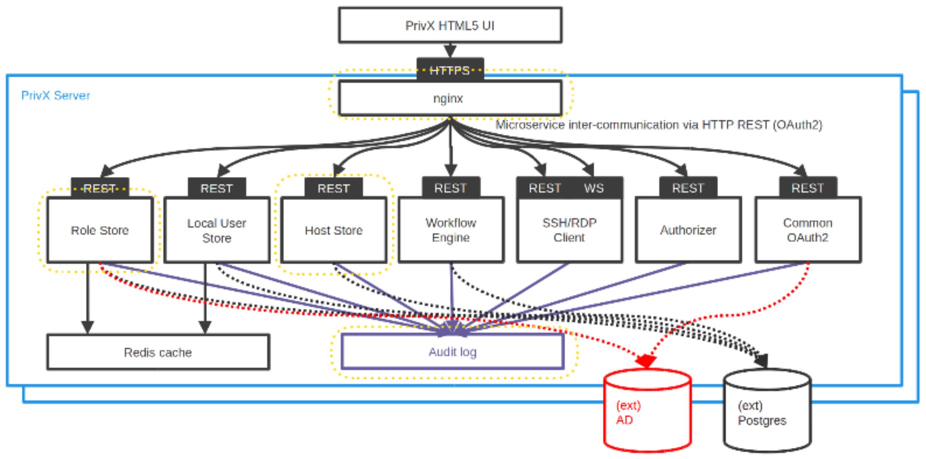

Figure 4.2: PrivX architecture [24].

- *Authentication service* implements OAuth2[1] to validate the user credentials against identities stored in the role store. Microservices authenticate among each other using this service [24].
- *Authorizer* is responsible for creating the short-lived role-based certificates. These certificates are used for authenticating the users against target hosts.
- *Host store* is a place holder for all hosts and hosts-related information. It stores information such as services (SSH or RDP) supported by the host and target-user-to-role mappings.
- *Key vault* encrypts and stores PrivX secrets. Stored credentials for target hosts, database encryption keys and web token signing keys are some of the secrets that are stored in the key-vault.

[1]https://oauth.net/2/

- *Local user store* helps to maintain a list of users locally. It can be needed in scenarios where there are no external user directories such as AD.

- *Role store* binds the users to roles. The rules for binding a role to the user are defined, and users are bound to the roles . It integrates with external user directories such as AD to retrieve and maintain up-to-date user information.

- *Workflow engine* allows administrators to define work flows for role granting and removal. An automated email notification is sent for the concerned users in each step.

- *SSH and RDP clients* enable PrivX users to connect to the target hosts from a web based user interface.

- *Web GUI* of PrivX is used to interact with the PrivX system. It is used by the administrators to configure and maintain PrivX. Examples of actions that can be carried out from the web GUI include adding hosts, defining roles, defining work flows and interfacing with cloud.

### 4.2.3 PrivX orchestrated with Kubernetes

PrivX is based on a microservice architecture and each microservice can be realized in a Docker container. Currently PrivX is distributed as an RPM package with each individual microservice as a service (process). A highly available setup of PrivX is achieved through complex scripts. An external load balancer with sticky session support is required to maintain high availability. A watchdog service is required to watch and restart the failing services in PrivX. It is complex to monitor the system when multiple instances of each microservice are running. Moreover, scalability is challenging under this deployment model. The goal of this thesis is to Dockerize the microservices in PrivX to address the scalability and availability requirements. Such a setup requires a container orchestration framework to facilitate deploying, monitoring and scaling the Docker containers.

We described the features of open source container orchestration frameworks in the previous chapter. The goal of this section is to identify a suitable container orchestration framework for PrivX. We compare the features provided by these frameworks in the context of PrivX. The list of requirements from the container orchestration framework required by PrivX are listed below:

- *Auto-scaling* is an important requirement for PrivX. Built-in auto scaling support allows to scale the service based on actual traffic patterns.

- *Load balancing and service discovery* should be built-in the framework. This enables a simple deployment model without the need for additional software.

- *Logging and monitoring* support is required. PrivX requires extensive monitoring and logging as it enables access to resources.

- *Secrets management* for keys, tokens, passwords should be supported by the framework and no additional software to manage the cluster secrets should be required.

- *Integrated cloud control* should be built into the framework. Support for all major cloud service providers is required. This enables a unified deployment model that would work across the different cloud service providers.

- *Deployment model* should support both on-premise and all major public cloud service providers. This will enable a unified solution. Additionally, the deployment model should support rolling updates and canary deployment. In canary deployment pattern, a new version of the service developed can be first tested in production in a smaller scale along with its previous version. Once the results are satisfactory, the new version can be scaled-up and the older version can be scaled down.

- *No additional software* should be required other than the chosen container orchestration framework itself. This will keep the deployment model simple.

- *Active support from community* is required to clarify and troubleshoot the issues.

The three container orchestration frameworks which were previously studied - Kubernetes, Docker Swarm and Nomad. As per our study the three frameworks fairly provide similar features. They are all developed and supported by large organizations. Load balancing and scaling is provided by all of them. Nomad is gaining more traction even though relatively new in this segment. Kubernetes has built-in auto-scaling support based on CPU utilization or custom metrics. Nomad has minimal auto-scaling support based on the number of requests pending to be served by the application. Docker Swarm has no auto-scaling support. Load balancing and service discovery is inbuilt both in Kubernetes and Docker Swarm. The CoreDNS[2] component of Kubernetes facilitates in service discovery. Docker engine acts as the DNS server in Docker Swarm. With respect to load balancing Kubernetes and Docker swarm provides built-in load balancing for services. Nomad relies on external tools and software for the same. Logging and monitoring support is natively available in Kubernetes. Both Docker Swarm and Nomad rely on third party tools. Kubernetes and Docker Swarm provide secrets management as an native feature. Nomad does not support this feature. Kubernetes provides 'integrated cloud control' with its in-built cloud control manager. The cloud control manager interacts with the underlying cloud providers and can implement the following [14]:

- *node controller* responsible managing Kubernetes nodes. It can update the Kubernetes nodes using cloud native APIs and deleting the nodes from the cluster that are removed from the cloud;
- *service controller* implements the load balancers in cloud;
- *route controller* is responsible for enabling network routes in the cloud;
- *persistent volume labels controller* enables setting up of zone and region labels on persistent volumes created; Currently it is only supported for Amazon Web Services (AWS) and Google Cloud Platform (GCP).

Such a feature is not available in Docker Swarm or Nomad. Additionally, both Kubernetes and Nomad have a web UI dashboard to monitor the cluster. This feature is not available in Docker Swarm. All the three frameworks support both on-premise and cloud deployment models. Rolling updates and canary deployment pattern is supported by all the three frameworks. As seen from the discussion, Kubernetes does not require any additional software for the requirements stated above. Docker Swarm and Nomad require third party software to satisfy certain features. Table 4.1 summarizes the same.

Finally, we examine the popularity of the frameworks in terms of GitHub statistics. As seen in Table 4.2, Kubernetes is the most popular framework with more than 67,000 commits[3]. It has one the largest number of contributors with more than 1,700 people actively contributing[4]. Docker Swarm and Nomad are relatively less popular and have fewer contributors when compared to Kubernetes.

[2]https://kubernetes.io/docs/tasks/administer-cluster/coredns/
[3]https://github.com/kubernetes/kubernetes
[4]https://github.com/kubernetes/kubernetes/graphs/contributors

| | **Kubernetes** | **Docker Swarm** | **Nomad** |
|---|---|---|---|
| **Auto-scaling** | Available. Based on CPU-utilization or custom metrics | Not available | Available based on pending requests |
| **Load balancing and service discovery** | In-built | Available | Relies on third party tools |
| **Integrated cloud control** | In-built cloud control manager | Not available | Not available |
| **Logging and monitoring** | In-built | Relies on third party tools | Relies on third party tools |
| **Secrets management** | Available | Available | Not available |
| **Dashboard** | Available | Not Available | Available |

Table 4.1: Kubernetes vs Docker Swarm vs Nomad

| | **Kubernetes [17]** | **Docker Swarm [7]** | **Nomad [22]** |
|---|---|---|---|
| Commits | 67,918 | 3,532 | 12,198 |
| Contributors | 1,743 | 165 | 259 |
| Forks | 13,625 | 5,202 | 731 |
| Stars | 38,978 | 1,066 | 3,696 |

Table 4.2: GitHub Statistics for various frameworks

From Table 4.1 and previous discussion it could be seen Kubernetes satisfies all the requirements for PrivX. All major cloud infrastructure providers such as Amazon, Microsoft Azure and Google support Kubernetes. Deploying applications with Kubernetes across multiple cloud vendors is streamlined. The process of deployment with Kubernetes is well supported and documented by the service providers. Moreover, Kubernetes is hosted and supported by the Cloud Native Computing Federation (CNCF) a large open source foundation that is also a part of the Linux foundation has organizations like Google, Microsoft, Amazon as its members. This ensures constant development and support for the framework. Furthermore, the case studies from large organizations such as Capital One and ING support the case for Kubernetes[5]. For example, 'adform' an advertising technology enabler

[5]https://kubernetes.io/case-studies/

had a large OpenStack[6] based private cloud infrastructure with 1,100 physical servers in 7 datacenters around the world. They had long release cycles attributing to the fact that their developers needed to maintain these virtual machines and they did not have a self healing infrastructure. They got their release cycle reduced to minutes from hours by using Kubernetes and redesigning their software by adopting the microservices architecture. They were able to achieve 6 times faster auto-scaling with Kubernetes when compared to their previous semi-manual VM bootstrapping [3].

Based on the above discussion, we choose Kubernetes to orchestrate PrivX. With Kubernetes the pod specification templates can be created for each service and they can be modified by the customers depending upon their needs. For example, an organization using PrivX has users in the range of several hundred thousands but only a few hundreds of servers requiring access. This would require certain services like role store, local user store to be scaled more than host store. These kind of customer specific use cases can easily be addressed with Kubernetes. Moreover, with a strong community support, the issues get resolved much quickly[7].

[6]https://www.openstack.org/

[7]https://github.com/kubernetes/kubernetes/issues?q=is%3Aissue+is%3Aclosed

# Chapter 5

# Implementation

This chapter details the implementation of PrivX microservices as Docker containers and their orchestration with Kubernetes. Section 5.1 describes the preliminary steps performed to create the prototype environment. Section 5.2 describes how PrivX was deployed with Kubernetes and Section 5.3 describes the additional software that was realized. Section 5.4 describes the experiments and evaluations performed.

## 5.1 Preliminary steps

Following the design decision to employ Kubernetes, we create a prototype system to test the high availability and scalability of Dockerized PrivX. The deployment model also eliminates the need for an external load-balancer and manual configuration of an external Postgres database. To this end, we perform the following steps:

- Dockerize the microservices in PrivX
- Install and configure a custom Kubernetes cluster on on-premise infrastructure
- Create database as a service with persistent storage
- Deploy PrivX Docker containers on Kubernetes
- Test the high availability and scalability requirements of PrivX in the prototype implementation

### 5.1.1 Dockerizing PrivX

We first Dockerized PrivX before deploying it on a Kubernetes cluster. In the current RPM deployment model of PrivX, the microservices are realized as individual services (processes) with their respective service start-up scripts. Dockerizing the microservices involves encapsulating them inside Docker containers. The required ports, environment variables are defined in the Docker file. Additionally, the required directory structure and configuration files for each microservice are defined inside the Docker file. A common configuration file (toml[1]) is created to define the common configuration requirements for all the containers, such as database connection parameters and service end points. A private GitLab Docker registry is created to host the built Docker images. Build scripts are created to build the microservices as well as the related Docker images and finally push the images to the registry.

The current architecture of PrivX requires a database (Postgres) to be installed and configured externally. The Postgres[2] database was Dockerized with persistent storage through Network File System (NFS), to eliminate the external dependency and to simplify PrivX deployment. The database is now created as a service and initialized automatically through a startup script inside the Docker container. The nginx service is built with sticky session support to act as a load balancer. This eliminates the need for external load balancers. All the microservices are self-contained with the required runtime, directories, environment variables and configuration files.

### 5.1.2 System environment

We create an on-premise prototype environment for our implementation. An on-premise prototype environment is chosen over a cloud native environment for the implementation. Moreover, the deployment model created in Kubernetes for on-premise should work the same in cloud environments provided the same system configurations are met. The prototype system comprises of 3 VMs (nodes): one master and two worker nodes. Virtualbox[3] is used to create the nodes. Table 5.1 illustrates the configuration of the nodes in the Kubernetes cluster. The master node has 4096 MB RAM memory whereas, each of the two worker nodes have 2048 MB of RAM memory. The host machine has a 2.60GHz 8-core processor, 16 GB RAM and runs Ubuntu 16.04 LTS. The nodes have a bridged adapter network configuration with the host system. The master node also serves as the Network File System (NFS) server

[1]https://github.com/toml-lang/toml
[2]https://www.postgresql.org/
[3]https://www.virtualbox.org/

for the system. The nodes swap memory was disabled for performance reasons[4]. Kubernetes tightly packs the nodes with pods to maximize resource utilization. The deployed pod should run on the node and should not use swap memory since swap memory is much slower than RAM memory. With swap memory in place, it is difficult for Kubernetes to pin the deployments with CPU and memory limits.

| | |
|---|---|
| CPU | 1 Core |
| Operating System | Ubuntu 16.04 LTS |
| Kubernetes Version | 1.11.2 |
| Docker Version | 17.03.2-ce |

Table 5.1: Configuration of Kubernetes cluster nodes

### 5.1.3 Kubernetes installation

Once the prototype environment is setup, we then create a Kubernetes cluster using Kubeadm[5]. The Kubeadm toolkit helps in bootstrapping a minimum viable Kubernetes cluster in a secure way. Kubeadm supports cluster life-cycle functions, such as upgrading, downgrading and securely managing the bootstrap tokens. Kubeadm provides `kubeadm init` and `kubeadm join` commands for creating the Kubernetes clusters. The `kubeadm init`[6] command is run to initialize the master node.

```
$ kubeadm init --pod-network-cidr <subnet-address>
```

It creates the necessary pods required for the master node and the token required by the worker nodes to join the cluster. The configuration file "admin.conf" is generated by this command, this is required to connect to the API server. The two worker nodes use `kubeadm join` command with the token generated by the `kubeadm init` command to join the cluster.

```
$ kubeadm join --token <token> <masternode-ip
    address:port> --discovery-token-ca-cert-hash <
    ca-cert hash>
```

[4]https://github.com/kubernetes/kubernetes/issues/53533
[5]https://kubernetes.io/docs/setup/independent/install-kubeadm/
[6]https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm-init/

Calico[7] is installed in the Kubernetes cluster. Calico provides simple, scalable and secure virtual networking that works well with all major public cloud service providers and private clusters as well. From Kubernetes version 1.11, CoreDNS[8] is installed by default and serves as the DNS server for the Kubernetes cluster. CoreDNS, Calico etcd, Calico controller are installed in the master node and rest of Calico containers are installed in the worker nodes. Kubernetes dashboard[9] is deployed in the master; it provides a web-based user interface to monitor the status of the cluster and its components. Once the necessary components are installed, the following commands are run to check the status of the pods and nodes in the cluster.

```
$ kubectl get pods --all-namespaces
$ kubectl get nodes
```

A status of 'running' for all the pods indicates that the pods are created, deployed and running successfully. The status of 'ready' with the nodes indicates that the nodes have joined the cluster and are running. Now the Kubernetes cluster is ready for application deployment.

## 5.2 Deploying PrivX with Kubernetes

Once the microservices are available in Docker containers, we deploy them with Kubernetes. The overall system architecture of the PrivX deployment model is shown in Figure 5.1. The *master node* contains the core Kubernetes and Calico pods. It also serves as the NFS server. The *worker nodes* contain the PrivX pods, kubelet[10] (node agent), proxy and Calico pods. Calico provides the overlay network for the cluster and enables inter pod communication.

### 5.2.1 Database as a service

We create the Postgres database as a container service in our implementation. Containers are short lived and stateless thus inherently lack persistent storage. Hence, Kubernetes persistent volumes[11] are used to persist the database

[7]https://www.projectcalico.org/calico-networking-for-kubernetes/
[8]https://coredns.io/
[9]https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/
[10]https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/
[11]https://kubernetes.io/docs/concepts/storage/persistent-volumes/

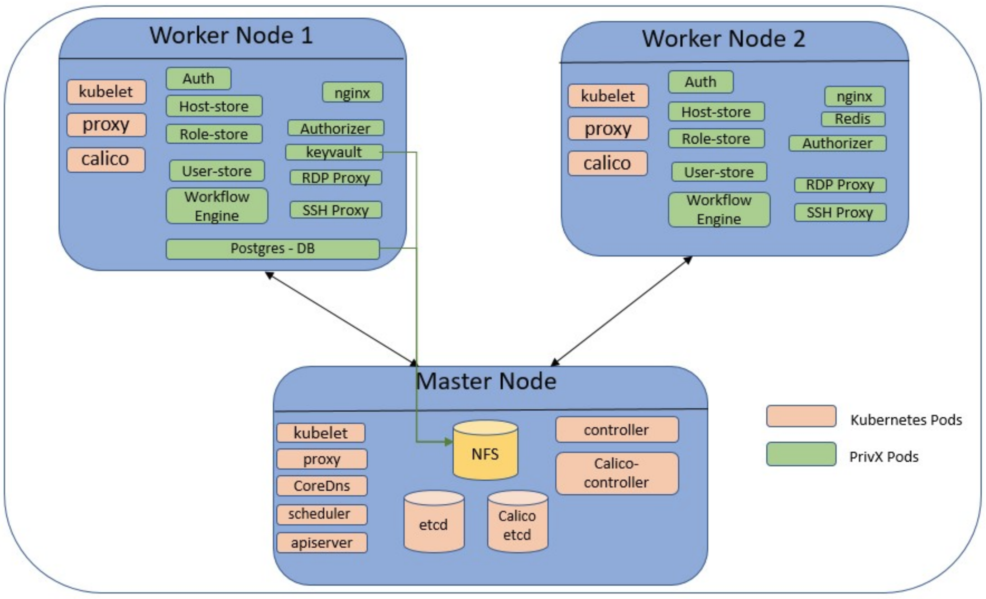

Figure 5.1: PrivX orchestrated with Kubernetes.

storage. `PersistentVolume` and `PersistentVolumeClaim` API resources are used to implement the persistent storage requirements. A PersistentVolume (PV) is a storage in the cluster that is provisioned by the system administrator. The storage could be NFS or a cloud specific storage system. The life cycle of PV is independent of any pods that use it. A PersistentVolume-Claim (PVC) is a request for storage by a service (pod). PVCs consumes PV resources. A PV for database storage is created using the NFS server (master node) as shown below.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs-pvc
spec:
  capacity:
    storage: 25Gi
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain
  nfs:
    path: /var/nfs/database
    server: <NFS server ip>
```

Listing 5.1: Persistent volume specification.

The PV definition indicates the kind as 'PersistentVolume' and specifies the storage capacity and specifies that the storage is of type NFS along with the path and NFS server IP. The PV created is consumed by a PVC as defined below.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: nfs-pvc
spec:
  accessModes:
  - ReadWriteMany
  resources:
     requests:
       storage: 25Gi
```

Listing 5.2: Persistent volume claim specification.

The PVC is linked to the PV based on the size specified in both the definitions. The status of the PV and PVC can be checked using the following commands. A status of 'bound' indicates that the persistent storage have been created and linked successfully.

```
  $ kubectl get pv
  $ kubectl get pvc
```

The Postgres database pod has a service and deployment definitions as indicated below. It can be seen that the PVC created is consumed by the deployment specification. The persistent storage is mounted to the database data directory.

```
apiVersion: v1
kind: Service
metadata:
  labels:
    service: postgres-nfs
  name: postgres-nfs
spec:
  ports:
  - name: "5432"
    port: 5432
    targetPort: 5432
  selector:
    service: postgres-nfs
  type: NodePort
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: postgres-nfs
spec:
  replicas: 1
  template:
    metadata:
      labels:
        service: postgres-nfs
    spec:
      containers:
      - env:
        image: launcher.gcr.io/google/postgresql9
        name: postgres-nfs
        volumeMounts:
        - name: pg-data
          mountPath: "/var/lib/postgresql/data"
        ports:
        - containerPort: 5432
        resources: {}
      volumes:
        - name: pg-data
          persistentVolumeClaim:
            claimName: nfs-pvc
      restartPolicy: Always
status: {}
```

Listing 5.3: Postgres service and deployment specification.

### 5.2.2 PrivX services orchestrated

As seen in Figure 5.1, we create the microservices in PrivX as a service in Kubernetes along with a corresponding deployment specification. Next, we create a Kubernetes secret [26] object to hold the image registry secrets. This secret is used during image pull. Kubernetes secret objects minimize the risk of accidental exposure and provide a streamlined mechanism to handle secrets. An example podspec for the 'role-store' service is shown below.

```
apiVersion: v1
kind: Service
metadata:
  labels:
    service: role-store
  name: role-store
spec:
  ports:
  - name: "8081"
    port: 8081
    targetPort: 8081
  selector:
    service: role-store
  type: NodePort
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: role-store
spec:
  replicas: 2
  template:
    metadata:
      labels:
        service: role-store
    spec:
      containers:
      - env:
          - name: SSH_LOG_LEVEL
            value: INFO
        image: <GitLab image registry path>
        name: role-store
        volumeMounts:
        - name: keyvault
          mountPath: "/opt/privx/keyvault"
        imagePullPolicy: Always
      imagePullSecrets:
        - name: gitlab-registry
      volumes:
        - name: keyvault
          persistentVolumeClaim:
            claimName: nfs-keyvault
      restartPolicy: Always
```

Listing 5.4: Role-store service and deployment specification

The `Service` specification exposes the ports and creates a service name. The `Deployment` specification mentions the number of replicas, the environment variables for the service, image path and the volume mounts. Additionally, the deployment specification mentions the Kubernetes secret object to be used during image pull. A similar podspec is created for all the services in PrivX and deployed in the Kubernetes cluster. We created a total of 31 Kubernetes objects (12 services, 12 deployments, 3 PVs, 3 PVcs and 1 secret) for deploying PrivX in our prototype system.

## 5.3 Additional software realized

As part of the thesis, the following steps were accomplished.

- *Automating the process of building Docker images* for all the services in PrivX, and pushing them to a private Docker image registry in GitLab.
- *Kubernetes cluster* creation is automated. The newly developed script takes care of the following.
  - Minimum system requirement checks for creating the cluster.
  - Creating the pod overlay network with Calico.
  - Creating the Kubernetes dashboard (Web-UI).
  - Creating the NFS server (The master node acts as the NFS server).
- *PrivX deployment* is a automated with script, that performs the following actions.

  - Creating and deploying the required PV and PVCs and verifying that their status is 'bound'.
  - Creating and deploying the PrivX services using the podspec (yaml definition).

- *Application log aggregation tool* is created to collect the logs from different services. Aggregating the individual microservices logs is challenging due to their distributed nature. The tool masks any secrets before collecting to ensure that no user secrets are collected and logged. Additionally, the tool runs several system commands to collect PrivX system information such as the hardware, software, networking, CPU and memory usage. The tool also collects additional configuration files, system logs and other information that are useful for debugging purposes.

- *A smart scaling tool* is created to scale the services (pods) based on predefined criterion and rules. The tool checks for predefined conditions, and once the condition is met, it scales the necessary pods. For example, a predefined rule could be set to scale the 'ssh-proxy' service when the number of active SSH connections exceed a certain limit. Similarly, when the number of connections reduce, the tool scales down the service after ensuring that there are no active connections from the given 'ssh-proxy' instance.

- *Host-store* microservice is implemented, which acts as place holder for all hosts (both on the cloud and on-premise) in the organizations infrastructure. Automatic scanning and updating of hosts across cloud service providers such as Google, Azure, Amazon and OpenStack is implemented. This ensures that PrivX has up-to-date information regarding the hosts. Each host defines a set of services offered by it such as SSH or RDP and contains a list of target principals (user accounts in the hosts). This enables connecting to the target host based on the user roles. Other host related meta-data is also stored which help in filtering and searching for specific hosts. Services like ssh-proxy and rdp-proxy communicate with the host-store to resolve a particular host for connection establishment. The microservice also implements the licensing restrictions i.e., number of active hosts that can be present in the PrivX system.

## 5.4 Evaluation and results

We evaluate the service recovery time for the core PrivX services. As discussed earlier, PrivX has a watchdog service in the current RPM deployment model to monitor and restart the failed services in PrivX. The prototype system created was used for evaluating the service recovery time in the Kubernetes deployment model. For the RPM deployment model, a CentOS VM with the system environment as described in the Table 5.2 is used. The host machine has a 2.60GHz 8-core processor, 16 GB RAM and runs Ubuntu 16.04 LTS.

| | |
|---|---|
| CPU | 1 Core |
| RAM | 4096 MB |
| Operating System | CentOS Linux 7 |

Table 5.2: Configuration of CentOS virtual machine

Basic deployment model of PrivX consists of 8 services - auth, authorizer, host-store, keyvault, user-store, role-store, ssh-proxy and workflow-engine. These 8 services are considered for our evaluation and experiments. We then measure the the time taken by a service to recover after it has been brought down. We refer to this as service recovery time in our experiments. The corresponding service pod for each service is brought down and the time is measured until the pod is in 'running' status again in the Kubernetes deployment. In the RPM deployment the corresponding service (process) is killed and the time is measured until the service status indicates as 'active'. 10 rounds of experiments are conducted for each service in their respective deployment models. The average service recovery time is calculated for each service in the two deployment models. The results of the experiment is summarized in Table 5.3.

The results indicate that there is no significant difference in the service recovery time between the two deployment models. The RPM based deployment model has a slightly better performance in terms of service recovery time.However, in absolute terms, the recovery time in case Kubernetes is also very low. Network latency might also have an effect on the Kubernetes deployment model as the images might not be available in the node and might need to be pulled from registry while recreating a replica.

It should be noted that in Kubernetes deployment model recreating the failed services involves recreating the service and deploying it on a worker node whereas in RPM deployment model it is restarting the failed service locally. Network latency has an effect on the Kubernetes deployment model.

| **Service** | **Mean recovery time in Kubernetes deployment - prototype system (ms)** | **Mean recovery time in RPM deployment(ms)** |
|---|---|---|
| auth | 0.0048151 | 0.0012466 |
| authorizer | 0.0035281 | 0.0012466 |
| host-store | 0.0047840 | 0.0011211 |
| keyvault | 0.0045735 | 0.0012251 |
| user-store | 0.0044905 | 0.0011564 |
| role-store | 0.0041297 | 0.0012825 |
| ssh-proxy | 0.0049788 | 0.0013479 |
| workflow-engine | 0.0069978 | 0.0011637 |

Table 5.3: Mean service recovery time in Kubernetes and RPM deployment models

The RPM deployment model has only one instance of the service running when compared to a minimum of 2 instances (replicas) of each service running in the Kubernetes deployment model. This results in the service being unavailable during the recovery time mentioned in the Table 5.3 for the RPM deployment model. Whereas, in Kubernetes deployment model, due to horizontal scaling and multiple replicas available for each service, the given service is always available and the service recovery time (for a single replica) may only cause a momentary performance bottleneck for the corresponding service during the service recovery time. A momentary performance degradation is acceptable for PrivX when compared to service unavailability which results in erroneous application behavior. Let us consider the scenario of the 'auth' service being unavailable during its recovery time in the RPM deployment model. Any access request made during this time would fail in this model since the service is not available for authenticating the users access request. This is not the case with the Kubernetes deployment model, where the service is always available and only a sudden surge in the number of access requests would result in a slightly delayed response. Considering the results and the above discussion it can be seen that PrivX deployed with Kubernetes offers a more resilient deployment model.

# Chapter 6

# Conclusion

Over the years, access management has evolved from a gatekeeper to enabler of business growth by enabling organizations to get their jobs done securely, efficiently and cost effectively [9], eventually becoming a key tool for risk and asset management in organizations. PrivX is an access management software developed by SSH Communications Security Oyj. In this thesis, we containerized PrivX and deployed it with Kubernetes. The main goals of this work is to setup a custom container orchestration environment that would enable PrivX to be deployed both on-premise and on public cloud service providers. We carried out a feature comparison of container orchestration frameworks: Kubernetes, Docker Swarm and Nomad. We evaluated these features with respect to PrivX requirements and in terms of support from open source community. We found that Kubernetes is the most suitable option for PrivX deployment.

Next, we containerized the microservices in PrivX. The database was also created as a service and encapsulated in a Docker container with persistent storage. We then created an automated custom Kubernetes cluster that was used to deploy the Dockerized microservices of PrivX. The scalability of PrivX in our prototype system (Kubernetes deployment model) was tested with the 'smart scaling tool' that was created. It was observed that PrivX could scale-out horizontally when the requirement to handle more traffic was there and scale-down seamlessly when the traffic reduced with the 'smart scaling tool'. Additionally, the watchdog service is not required in the Kubernetes model as it is natively available in Kubernetes. Kubernetes natively provides high-availability of services based on the replica count mentioned in the deployment specification of each service. This eliminated the need for complex set-up scripts used in the RPM deployment model. Even though, the cumulative image sizes of the individual services (dockerized) is greater than the single RPM distribution, the services are self contained and can be deployed across a cluster or any Docker container supported framework. Next we evaluated the service recovery time for each of the services in PrivX both in RPM deployment model and in the prototype Kubernetes deployment model. The results indicates that there is no significant difference between the two models. The services were unavailable during their recovery time in the RPM deployment model, whereas, the services were always available in the Kubernetes deployment due to their higher replica count. Finally, we found that Kubernetes helps in easy and rapid deployment of PrivX and provides a unified solution for both on-premise and cloud deployment models.

The concepts presented in this thesis can be extended in the following ways.

- The proof of concept auto-scaling tool can be extended as a dedicated microservice in PrivX. The rules can be made configurable and analytics could be collected and used to define custom scaling models for each customer. Auto scaling of worker nodes would also enable distribution of work among the nodes i.e., reallocating some pods from overloaded nodes.
- We used two worker nodes and a single master node in our implementation. This can be extended to a cluster comprising more master and worker nodes to provide more processing power to support large-scale deployments. Moreover, a comparison between on-premise Kubernetes deployment and cloud native Kubernetes deployment models could be made to assess the performance of PrivX. Deploying PrivX on a hybrid cloud environment [49] would be an interesting future use case.
- A study on deploying PrivX on IoT devices is an interesting topic for future research. IoT devices collect large amounts of data and any unauthorized access to them would lead to massive privacy breach [48]. PrivX deployed in an IoT ecosystem will enable streamlined access management and protect them from misuse. PrivX provides a browser based Linux shell that could be used to deploy, configure and monitor the IoT devices that inherently lack any display or keyboard.

# Bibliography

[1] Access control models. https://pdfs.semanticscholar.org/dd05/c09a0772a3beeb3db675f8b07cf80e539e6d.pdf. (Accessed : 2018-09-07).

[2] Architecture. https://www.nomadproject.io/docs/internals/architecture.html. Accessed: 2018-06-15.

[3] CASE STUDY: adform. https://kubernetes.io/case-studies/adform/. Accessed: 2018-09-10.

[4] Chapter 1. introduction to control groups (cgroups). https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01. Accessed: 2018-06-01.

[5] Chapter 1.2. linux containers architecture. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/7/html/overview_of_containers_in_red_hat_systems/introduction_to_linux_containers. Accessed: 2018-06-01.

[6] Docker Engine. https://docs.docker.com/engine/docker-overview/#docker-engine. Accessed: 2018-06-20.

[7] Docker swarm github. https://github.com/docker/swarm. Accessed: 2018-07-22.

[8] Enabling microservices with containers and orchestration docker, mesos, and kubernetes explained. https://www.mongodb.com/containers-and-orchestration-explained. Accessed: 2018-07-03.

[9] The evolution of identity and access management. http://media.govtech.net/CA_Resource_Center/evolution_of_iam_wp.pdf. (Accessed : 2018-09-12).

[10] Experiences from failing with microservices. https://www.infoq.com/news/2014/08/failing-microservices. Accessed: 2018-06-25.

[11] How swarm works. https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/. (Accessed : 2018-06-26).

[12] Images and layers. https://docs.docker.com/storage/storagedriver/#images-and-layers. Accessed: 2018-05-20.

[13] Introduction to nomad. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes. Accessed: 2018-06-15.

[14] Kubernetes Cloud Controller Manager. https://kubernetes.io/docs/tasks/administer-cluster/running-cloud-controller/. Accessed: 2018-09-10.

[15] Kubernetes cloud controller manager - kubernetes. https://kubernetes.io/docs/tasks/administer-cluster/running-cloud-controller/. Accessed: 2018-04-25.

[16] Kubernetes components. https://kubernetes.io/docs/concepts/overview/components/. Accessed: 2018-04-23.

[17] Kubernetes github. https://github.com/kubernetes/kubernetes. Accessed: 2018-07-22.

[18] Microservices architecture, containers and docker. https://www.ibm.com/developerworks/community/blogs/1ba56fe3-efad-432f-a1ab-58ba3910b073/entry/microservices_architecture_containers_and_docker?lang=en. Accessed: 2018-07-01.

[19] Microservices Decoded. https://dzone.com/articles/scalable-cloud-computing-with-microservices. Accessed: 2018-06-21.

[20] Microservices: Five Architectural Constraints. https://www.nirmata.com/2015/02/02/microservices-five-architectural-constraints/. Accessed: 2018-07-06.

[21] Multi-Stage Docker Build. https://docs.docker.com/develop/develop-images/multistage-build. Accessed: 2018-06-20.

[22] Nomad github. https://github.com/hashicorp/nomad. Accessed: 2018-07-22.

[23] Privx. https://www.ssh.com/products/privx/. Accessed: 2018-06-29.

[24] Privx admin manual. https://help.ssh.com/helpdesk/attachments/36010481405. Accessed: 2018-07-22.

[25] Privx datasheet. `https://info.ssh.com/privx_datasheet`. Accessed: 2018-07-22.

[26] Secrets. `https://kubernetes.io/docs/concepts/configuration/secret/`. Accessed: 2018-09-01.

[27] What is a container. `https://www.docker.com/what-container`. Accessed: 2018-07-01.

[28] What is kubernetes? `https://www.nomadproject.io/intro/index.html`. Accessed: 2018-06-05.

[29] What is privileged access management (pam) software? `https://www.gartner.com/reviews/market/privileged-access-management-solutions`. Accessed: 2018-06-25.

[30] Anderson, C. Docker [software engineering]. *IEEE Software 32*, 3 (2015), 102–c3.

[31] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., et al. A view of cloud computing. *Communications of the ACM 53*, 4 (2010), 50–58.

[32] Balalaie, A., Heydarnoori, A., and Jamshidi, P. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software 33*, 3 (2016), 42–52.

[33] Balzan, D. J. B. Distributed computing framework based on software containers for heterogeneous embedded devices.

[34] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *ACM SIGOPS operating systems review* (2003), vol. 37, ACM, pp. 164–177.

[35] Bass, L., Weber, I., and Zhu, L. *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015.

[36] Bennett, K. H., and Rajlich, V. T. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering* (2000), ACM, pp. 73–87.

[37] Bernstein, D. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing 1*, 3 (2014), 81–84.

[38] Bhardwaj, S., Jain, L., and Jain, S. Cloud computing: A study of infrastructure as a service (iaas). *International Journal of engineering and information Technology 2*, 1 (2010), 60–63.

[39] Boettiger, C. An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review 49*, 1 (2015), 71–79.

[40] Bui, T. Analysis of docker security. *arXiv preprint arXiv:1501.02967* (2015).

[41] Bunyard, M. What is next-generation privileged account management? `https://securityintelligence.com/what-is-next-generation-privileged-account-management/`. Accessed: 2018-06-25.

[42] Butt, A. B., Hillyard, P. B., and Su, J. Certificate-based authentication system for heterogeneous environments, June 22 2004. US Patent 6,754,829.

[43] Deinhart, K., Gligor, V., Lingenfelder, C., and Lorenz, S. Method and system for advanced role-based access control in distributed and centralized computer systems, June 8 1999. US Patent 5,911,143.

[44] Dmitry, N., and Manfred, S.-S. On micro-services architecture. *International Journal of Open Information Technologies 2*, 9 (2014).

[45] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. Microservices: yesterday, today, and tomorrow. *arXiv preprint arXiv:1606.04036* (2016).

[46] Ebert, C., Gallardo, G., Hernantes, J., and Serrano, N. Devops. *IEEE Software 33*, 3 (2016), 94–100.

[47] Fowler, M., and Lewis, J. Microservices. `https://martinfowler.com/articles/microservices.html`. Accessed: 2018-06-05.

[48] Fremantle, P., Aziz, B., Kopecký, J., and Scott, P. Federated identity and access management for the internet of things. In *Secure Internet of Things (SIoT), 2014 International Workshop on* (2014), IEEE, pp. 10–17.

[49] Goyal, S. Public vs private vs hybrid vs community-cloud computing: a critical review. *International Journal of Computer Network and Information Security 6*, 3 (2014), 20.

[50] Graziano, C. D. A performance analysis of xen and kvm hypervisors for hosting the xen worlds project.

[51] Grobauer, B., Walloschek, T., and Stocker, E. Understanding cloud computing vulnerabilities. *IEEE Security & Privacy 9*, 2 (2011), 50–57.

[52] Herbsleb, J. D., and Moitra, D. Global software development. *IEEE software 18*, 2 (2001), 16–20.

[53] Hightower, K., Burns, B., and Beda, J. *Kubernetes: Up and Running: Dive Into the Future of Infrastructure.* " O'Reilly Media, Inc.", 2017.

[54] Hur, J., and Noh, D. K. Attribute-based access control with efficient revocation in data outsourcing systems. *IEEE Transactions on Parallel and Distributed Systems 22*, 7 (2011), 1214–1221.

[55] Jaramillo, D., Nguyen, D. V., and Smart, R. Leveraging microservices architecture by using docker technology. In *SoutheastCon, 2016* (2016), IEEE, pp. 1–5.

[56] Javed, A., et al. Container-based iot sensor node on raspberry pi and the kubernetes cluster framework.

[57] Joshi, S. Organization & cultural impact of microservices architecture. In *International Conference on Applied Human Factors and Ergonomics* (2017), Springer, pp. 89–95.

[58] Krause, L. *Microservices: Patterns and Applications: Designing fine-grained services by applying patterns.* El Autor, 2015.

[59] Lewis, J., and Fowler, M. Microservices: a definition of this new architectural term. *Mars* (2014).

[60] Melo, M. F. Developing microservices for paas with spring and cloud foundry, 2014.

[61] Nerur, S., Mahapatra, R., and Mangalaraj, G. Challenges of migrating to agile methodologies. *Communications of the ACM 48*, 5 (2005), 72–78.

[62] Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J., and Josuttis, N. M. Microservices in practice, part 2: Service integration and sustainability. *IEEE Software 34*, 2 (2017), 97–104.

[63] Popek, G. J., and Goldberg, R. P. Formal requirements for virtualizable third generation architectures. *Communications of the ACM 17*, 7 (1974), 412–421.

[64] Rensin, D. K. Kubernetes-scheduling the future at cloud scale.

[65] Sandhu, R. S., Coyne, E. J., Feinstein, H. L., and Youman, C. E. Role-based access control models. *Computer 29*, 2 (1996), 38–47.

[66] Schwaber, K., and Beedle, M. *Agile software development with Scrum*, vol. 1. Prentice Hall Upper Saddle River, 2002.

[67] Sharma, P., Chaufournier, L., Shenoy, P., and Tay, Y. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference* (2016), ACM, p. 1.

[68] Singleton, A. The economics of microservices. *IEEE Cloud Computing 3*, 5 (2016), 16–20.

[69] Sirisha, A., and Kumari, G. G. Api access control in cloud using the role based access control model. In *Trendz in Information Sciences & Computing (TISC), 2010* (2010), IEEE, pp. 135–137.

[70] Srinivasan, M. K., Sarukesi, K., Rodrigues, P., Manoj, M. S., and Revathy, P. State-of-the-art cloud computing security taxonomies: a classification of security challenges in the present cloud computing environment. In *Proceedings of the international conference on advances in computing, communications and informatics* (2012), ACM, pp. 470–476.

[71] Tosatto, A., Ruiu, P., and Attanasio, A. Container-based orchestration in cloud: state of the art and challenges. In *Complex, Intelligent, and Software Intensive Systems (CISIS), 2015 Ninth International Conference on* (2015), IEEE, pp. 70–75.

[72] Truyen, E., Van Landuyt, D., Reniers, V., Rafique, A., Lagaisse, B., and Joosen, W. Towards a container-based architecture for multi-tenant saas applications. In *Proceedings of the 15th International Workshop on Adaptive and Reflective Middleware* (2016), ACM, p. 6.

[73] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at google

with borg. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM, p. 18.

[74] Xavier, M. G., Neves, M. V., Rossi, F. D., Ferreto, T. C., Lange, T., and De Rose, C. A. Performance evaluation of container-based virtualization for high performance computing environments. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on* (2013), IEEE, pp. 233–240.

[75] Xu, X. From cloud computing to cloud manufacturing. *Robotics and computer-integrated manufacturing 28*, 1 (2012), 75–86.