Freie Universität Berlin

Bachelorarbeit Faculty of Computer Science at the Freie Universität Berlin

Research Group Software Engineering

Public Cloud Based Infrastructure for Telecommunication Cloud Control Panel in an Enterprise Environment

Dor Cohen
Student Number: 5138283
dcohen92@inf.fu-berlin.de

Advisor:	Mr. Samu Toimela
First Examiner:	Mr. Barry Linnert
Second Examiner:	Prof. Dr.-Ing. Jochen Schiller

Berlin, July 29, 2020

Statement of Originality

I hereby confirm that I have written the accompanying thesis by myself, without contributions from any sources other than those cited in the text and acknowledgments. This applies also to all graphics, drawings, maps and images included in the thesis.

July 29, 2020

Dor Cohen DorCohen

Statement of Gratitude

I would like to thank the people at Nokia that were involved in the process, Mr. Samu Toimela and Mr. Markku Niiranen - Nokia, Espoo Finland, for making the time at Nokia exceptionally pleasant. The support and mentoring were truly extraordinary.

I would also like to thank Mr. Barry Linnert - Freie Universitaet, Berlin Germany, for his continuous help and support in these complex times.

Contents

1 Introduction

Public cloud infrastructure has become an ubiquity in the field of web-services and web-applications. It offers increased flexibility by delegating hardware and low-level capabilities to professional vendors. It allows companies to focus on business crucial tasks instead of maintaining complex infrastructures. Notable cloud vendors such as *Amazon Web Services*[2] (AWS), *Google Cloud Platform*[7] (GCP) and *Microsoft Azure*[18] made a great leap in making setup of cloud environments easier, and more accessible.

Transition of applications from a traditional infrastructure to a cloud environment is a complex task which involves many supporting applications and deep knowledge of computer networks. Popular tools and concepts today such as *Serverless Architecture* [1] overlap with cloud architecture and are aimed to make such a transition easier and more intuitive. However, this change of architecture in an enterprise grade application demands a careful planning by professionals such as *System Engineers* and *Cloud Architects*.

This work describes the migration process of an application used to monitor the operation of *Edge Cloud* [2] applications as a part of a recent trend related to *5 Generation* (5G) mobile networks and *edge* facing devices. The application serves as a central operation and management panel, therefore increasing the observability and improving the ease of maintenance of edge cloud applications. The monitoring application was migrated from Nokia's on-premise data-center to a cloud vendor. Description of the technical details of the application itself are out of the scope of this work, and the emphasis is purely on computer networks and infrastructure related topics rather than mobile telecommunication. However, the field of mobile telecommunication has its requirements regarding aspects such as scalability, portability, reliability and high performance. We would like to show that the migration process and the used tools such as Kubernetes for *Container Orchestration* [11], Kong for Public Facing API Gatway [9], Keycloak for *Access Management* [8] and others, indeed satisfy these requirements, at least on the conceptual level, as this is not an empirical study, but rather an implementation of a *Proof of Concept* (PoC).

The aim on a scientific level is to provide another perspective on the migration process of complex systems in the mobile communication realm. This will hopefully assist in the standardization and unification of similar flows, as these are not part of enterprise grade processes at the moment.

We would like to prevent sensitive information about the application from being exposed. Therefore, throughout this work, generic terms such as *application*, *monitoring application*, *operation and management application* will be used to refer to the concrete application that was developed by Nokia.

[1]A subclass of cloud architecture where server applications being deployed as a standalone unit without worrying about the hosting environment [22]

[2]Cloud architecture which is placed closer to the consumer, considering it's geographical location and by that reducing latency and spreads wide area networks' load

2 Fundamentals

The following high-level overview provides an exposure to relevant concepts of cloud computing. It is important to stress that the field and its terminology are relatively new and dynamic, therefore, some of the basic terms are not well defined, as these mostly arise from the private enterprise sector. However, the underlying technical background of the driving technology is indeed sound and backed by an active academic research. Best effort is being made to provide a consistent and clear terminology.

2.1 Cloud Computing

The *Cloud* might appear as an ambiguous term. This concept is however well defined and describes the offering of computing resources as an internet service. [28] Cloud computing tries to abstract different parts of the computer infrastructure such as hardware, networking and operating system. This allows the service consumer such as an individual or a company to focus on the business logic implementation rather than the entire supporting infrastructure.
A precursor to the field and associated concepts (somewhat in their primitive form) were coined by *Turing Award* laureate John McCarthy's work in the field of *Time Sharing Systems*, where application may run on resources that seem independent but share the same underlying real-estate.[23]. Public cloud is a high level abstraction over public internet servers. These are provided by vendors for public use and are often split into different tiers, creating the typical cloud computing suite. [26]

2.1.1 Infrastructure as a Service - IaaS

IaaS is a cloud model described by the ability of the end-user to receive control over processing power, memory and storage, while abstracting hardware and low-level features such as networking and physical storage. Typically, the computing units are virutalized. This is usually the first level of cloud-abstraction for applications which were originally hosted on traditional servers in on-premise data-centers.[28]
From the user point of view, highly available computing resources are obtained, and guarantees such as availability and scalability are not of the user's concern. These are rather declared by every cloud vendor as part of its contract with the users. Usually, cloud orchestration solutions such as Kubernetes and OpenStack are used to manage and control the application and virtual machine provisioning, persistence storage layer and other resources.

2.1.2 Platform as a Service - PaaS

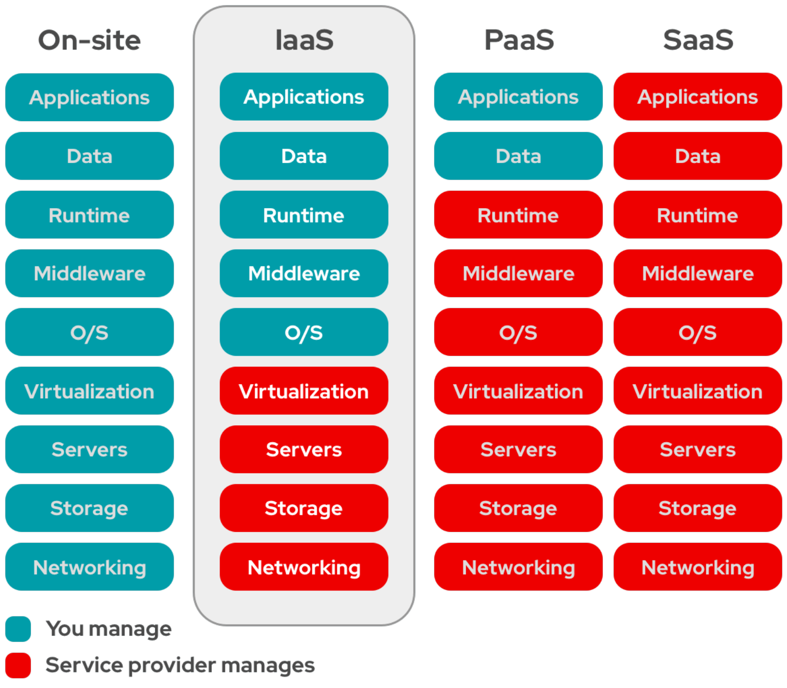
In some cases the end-user is lacking the need or ability to maintain and configure the underlying software-infrastructure such as operating systems and related softwares. PaaS offers an application-specific, pre-configured hosting platform.
The deployment of applications is being done with a relative disregard to the actual infrastructure in use by the vendor, as each application is to be observed as a self

sufficient entity and not as a process in an operating system. The platform might be responsible for storage, caching, logging and other utilities as separate applications running on arbitrary hosts in the network. [28]

Notable PaaS providers are Salesforce's Heroku, DigitalOcean. While general cloud providers such as Amazon Web Services, Microsoft Azure and Google Cloud offer specific services corresponding to Paas.

2.1.3 Software as a Service - SaaS

The typical definition for such a model is a public facing internet based application. Frequently it is characterized by a well defined purpose, subscription based usage and communication via strict set of API such as graphical, HTTP and other web protocols. [28]



Source: https://www.redhat.com/de/topics/cloud-computing/what-is-iaas

Figure 1: On-site, IaaS, PaaS, SaaS

Figure 1 shows the different cloud models by features. Each model is distinguished from the other by the services it offers to the end user. A clear distinction between the models might be challenging, as they can overlap in functionalities, e.g the popular web service *Kubernetes as a Service* is placed between IaaS and PaaS.

2.2 Containers

Historically, *Virtual Machines* were the primary method to achieve multi-environment virtualization on a single host. *Containers* allow simulation of an independent environment similarly to virtual machines, while reducing the overhead substantially, down to a mild performance drop compared to a native running application. This is possible due to the fact that hosts' resources such as the kernel and operating system libraries can be shared among containers. Although containerization features may be found in various operating systems such as Solaris and BSD, we will focus on these that are found in *Linux*[32]

Despite the fact that container technologies are considered to be *cross-platform*, and support most commercially available operating systems, the actual container *run-time* is bound to run using the linux kernel. Other operating systems cannot utilize the containerization engine without a mediator, and require an additional layer of virtualization called a *Hypervisor*, on which the containers can run. The additional virtualization layer might cause a performance impact, however, the deployment of applications in production environments is often being done on hosts running either *Windows* or specific *Linux distributions*, both are able to run containers directly[15, 24] and prevent major performance drawbacks.
Different efforts such as *Linux Container*[16] (LXC) and the *Open Container Initiative* (OCI) [19] are responsible for creating industry standards of containerization technologies.

Containers have become exceptionally valuable in the cloud environment as they offer the needed abstraction for applications orchestration.

2.2.1 Container Internals

The backbone of *Linux* containers technologies are *namespaces* and *cgroups*, which are rooted in the kernel.

Namespaces describe different aspects in which running processes can be isolated from each other, namely

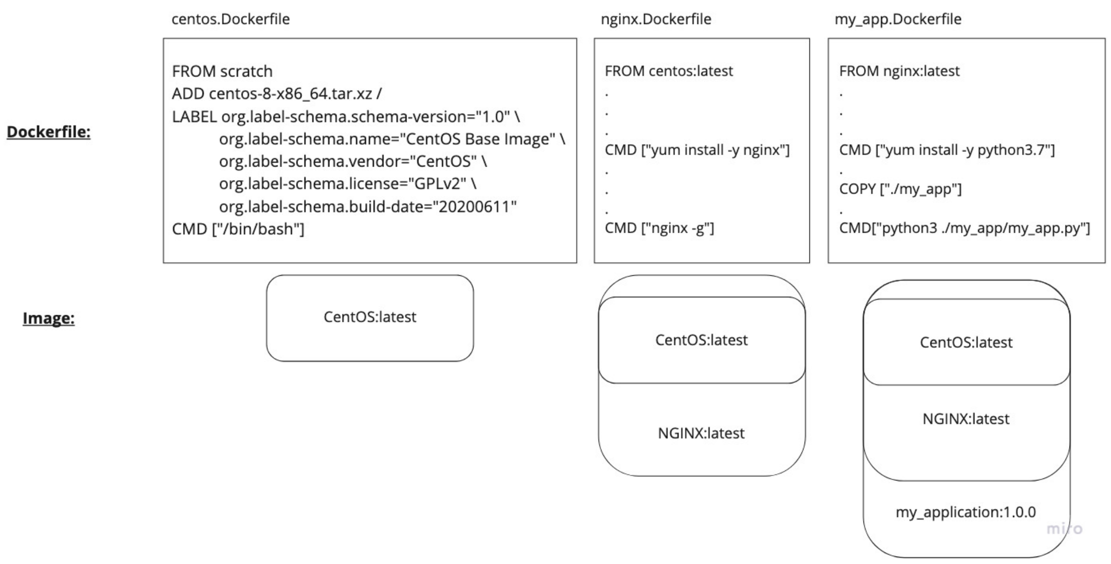
- Mount (mnt) - Selected mount points are copied from the current namespace to the newly created one.
- Process ID (pid) - Process ID table is completely separate from other namespaces.
- Network (net) - A complete and separate network stack.
- Interprocess Communication (ipc) - Prevention of interprocess-communication between namespaces.
- UTS (host and domain-name) - A UTS namespace has a different host and domain-name.
- User ID (user) - Separation of the user-id and group-id spaces between namespaces.
- Control Groups (cgroups) - Separation of different control groups.

- Time Namespace - Separation of system time between namespaces.

Control groups (Cgroups) allow resource allocation (CPU, memory, network bandwidth, disk) to predefined group of processes. The model is hierarchical, as each child-Cgroup inherits attributes from the parent-Cgroup. The control of Cgroups over different systems' resources is divided into *subsystem*, where each one represents a certain resource. [20]

2.2.2 Container Runtime - Docker

Docker [24] is a popular linux container application suite. It creates, manages and runs self-sufficient artifacts called *Images* which contain an application and its dependencies. Images are defined by a manifest called *Dockerfile*, which dictates important traits and actions like base image, container environment, initialization commands, mount points to host, file copy from host and initialization scripts.



```
FROM scratch
ADD centos-8-x86_64.tar.xz /
LABEL org.label-schema.schema-version="1.0" \
        org.label-schema.name="CentOS Base Image" \
        org.label-schema.vendor="CentOS" \
        org.label-schema.license="GPLv2" \
        org.label-schema.build-date="20200611"
CMD ["/bin/bash"]
```

```
FROM centos:latest
.
.
.
CMD ["yum install -y nginx"]
.
.
.
CMD ["nginx -g"]
```

```
FROM nginx:latest
.
.
.
CMD ["yum install -y python3.7"]
.
COPY ["./my_app"]
.
CMD["python3 ./my_app/my_app.py"]
```

Figure 2: Dockerfile and Docker image composition

Docker allows composition of containerized application by expansion of existing ones using a layering pattern. This prevents redundancy, as existing images are to be shared across others, and require exactly one replica of each, thus saving on storage and at times also on network load. In figure 2 an image is created by explicitly installing CentOS while using a *scratch* base image (the default empty image). Later this image is being used as a base image for the Nginx web server resulting in another image which lastly will create the final image, satisfying the application's dependencies. Composition of Docker images makes the run-time environments script-able and reproducible.
A running instance of a Docker image is called *Docker Container*. Communication with the container is possible via a command line interface, allowing monitoring and

dynamic modification of the running instance. [24]

The *Docker Hub* is an integral part of *Docker as a Service* and offers a centralized registry[3] for docker images, where all the configurations and dependencies are already set. This allows quick containerization of well known environments. Often the images are official, i.e released by the same entity that the created the software.

2.3 Cluster Management - Kubernetes

Modern web services are expected to be highly available and fail resistant, while providing responsiveness and low latency. [29] Software management and system management are traditionally mundane, repetitive and possibly manual tasks. Automation of these tasks has been a crucial point in the last decades, especially in large companies.

Kubernetes is a cluster management system developed by Google. It allows a complete description of the behavior of complex clustered applications as human-readable files called *manifests*. [17]. These describe application configurations, secrets management, replication and scaling strategies, monitoring, failure handling strategies and persistent storage patterns.
More specifically, Kubernetes can be described as container orchestration software, as it assumes an underlying registry of ready-made images as containerized applications, and is un-opinionated towards the specific containerization technology.
A *Kubenetes* cluster consists of *Worker Nodes* and a *Control Plane* (Master Nodes), which in turn consist lower-level applications called *Kubernetes Components* that are strictly split in order to satisfy different Kubernetes features. In general, they run, schedule, store and interact with internal and external interfaces.

In the following sub-section we will review these components.

2.3.1 Control Plane Components

- kube-apiserver (API Server) - The apiserver exposes the Kubernetes API, allowing admin communication to the cluster in order to validate and configure various Kubernetes workloads. This is the main gateway for both external and internal configuration, and changes during run-time.

- etcd (Cluster Data Database) - A distributed database responsible for storing cluster related data such as cluster health and state, name and other metadata related to the nodes and running application.

- kube-scheduler (Application Scheduler) - As the name suggests, it assigns newly created workloads to available computing resource (Worker Nodes) while taking into account constraints and requirements.

[3]hosting platform for artifacts, in this case Docker images

- kube-controller-manager (Cluster Controller) - The controller is the brain of the cluster. It watches the cluster's behavior, manages Pods replication, API access tokens and accounts for kubernetes namespaces.
- cloud-controller-manager (Cloud Provider Driver) - The controller manages the cloud vendor integration.

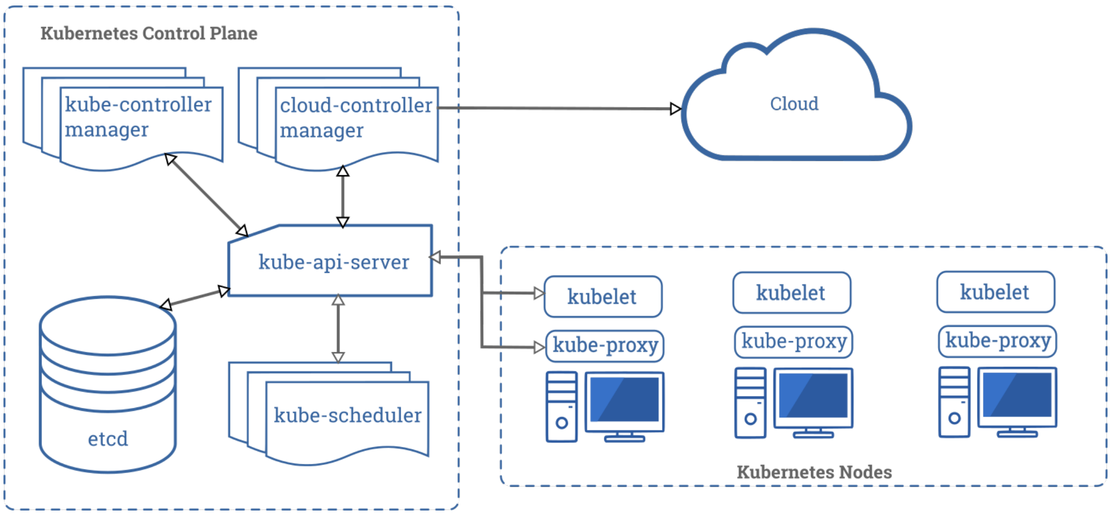
[30]

2.3.2 Worker Node Components

Each of the following components exists and operates inside every single worker node in the cluster.

- kubelet - A deamon which watches and manages individual containers according to the Pod's manifest.
- kube-proxy - The proxy component manages network rules and network sessions to the worker node from inside and outside of the cluster
- Container Runtime - A software responsible for running containers. Kubernetes supports a multitude of runtimes, such as *Docker*, *containerd* and *CRI-O*

[30]



Source: `https://kubernetes.io/docs/concepts/overview/components`

Figure 3: Overview of Kubernetes components

Figure 3 shows a high level overview of the interaction and inter-operation of different Kubernetes components. These are grouped by the type of the node to which they belong (Master or Worker).

2.3.3 Kubernetes Objects

Kubernetes defines certain abstractions in order to represent the state of the cluster. Each one is expressively describing a concrete need of any cluster-based application.

- Pod - A *Pod* is the most basic deployable unit of a Kubernetes application. It encapsulates a set of one or more highly coupled containers, ensuring that these will run on the same remote host. A Pod would usually posses a unique, cluster-internal IP address and is analogous to a virtual host.
- Service - An abstraction which defines Pod access over a network, either internal or external.
- Volume - The persistence layer of a Pod. Pods are ephemeral, and the information held during run-time is volatile by default. The Volume object provides the Pods various persistence options.
- Namespace - Supporting separation of different bounded contexts[4] on the same cluster.
- Deployment - A Kubernetes object representing a single real-life deployment by providing declarative updates for Pods and ReplicaSets. The properties describe the desired state of an application, e.g number of replicas.
- ReplicaSet - A ReplicaSet is a Kubernetes controller object which maintains the desired state of the application.
- StatefulSet - While Deployments and ReplicaSets declaratively define behavior of stateless Pods, StatefulSet defines the behavior of stateful Pods set, where each Pod is unique and not an arbitrary replica. These are utilized for instance in session-aware applications.
- Job - A Job describes an ad-hoc Pod that is maintained until task completion.
- DaemonSet - In a Kubernetes environment Pods are being scheduled to worker nodes by the kube-scheduler. By default there is no control over the scheduling pattern of a Pod to a worker node. A DaemonSet defines the degree of affinity of Pods to worker nodes.

[31]

2.3.4 Kubernetes Networking

Kubernetes internal networking mechanism is not supplied as part of the core features, but rather only defines an interface and guidelines for users to create their own networking solutions. In the past, these requirements were specific to the Kubernetes realm, but the convergence of needs from other container- and container orchestration solutions led to a well defined standard to be formed, the *Container Network Interface* (CNI), which demands the following.[4]

[4]Bounded Context is a term describing the separation of applications' business-domains into cohesive groups

- Pods can communicate with every other pod, either on the same physical machine or a different one without *Network Address Translation* (NAT). This means that each pod sees itself with the same cluster-internal IP address as every other Pod in the cluster, e.g when Pod 1 sends a packet to Pod 2, the source address that Pod 2 sees should be the same as Pod 1 sees in his local network.
- Any Kubernetes object (agent) on the node can communicate to all Pods on that node

By following these requirements, every networking solution for Kubernetes will abstract any physical or virtual machine in the network, so the cluster can be seen as a single computing resource, or alternatively, multiple ones connected in such away which does not require address-to-address mapping such as NAT. A wide selection of Kubernetes networking solutions exists today, and each one puts an emphasis on a specific aspect, such as ease of use, customizability and interoperability with other networks. Notable network plugins are Calico, Flannel, Weave and AWS CNI.

Understanding the inter-pod networking will assist clarifying more complex features, such as load-balancing and dynamic routing.
As described earlier, the Kubernetes service object is a Kubernetes component which dictates the internal and external access to pods inside the cluster. Each exposed pod has an attached service object. By default the cluster is completely isolated from external networks and allowing communication only between internal applications, effectively, a private network.
Kubernetes supplies different *Service Types* in order to allow external communication:

ClusterIP
This is the simplest service type attached to a Pod. Normally this is an indication that the Pod of the running applications is only accessible from within the cluster. Kubernetes allows to proxy external requests to it by using the proxy command.

NodePort
Exposes a specific port on each Node leading to a single Pod. This allows "routing" of external requests to internal applications by exposing each Node's IP address. Any node in the cluster will forward the incoming request to another node in the cluster whenever it does not posses a replica of the application's Pod.[13]

LoadBalancer
This service type generates a creation of a load balancer with a unique IP by most cloud providers. This allows any request to be directed to the correct node port, while the load balancer spreads the traffic across the Pods on different working nodes. A significant caveat of this service type is the unique IP assignment to each service, thus increasing costs and complexity. The LoadBalancer service type uses the Node Port service type in order to communicate to the node directly.

Ingress
The Ingress object provides a scalable and reliable solution for external exposure of

clusters' pods. It allows a server gateway to act as a reverse proxy, while abstracting multiple Kubernetes services under the same IP (and not different IPs like the Load-Balancer Service Type) and providing routing functionalities for external requests to multiplex to the correct service. Usually it is being done by HTTP host and/or URL path routing, so that each path represents a different service. [12]
Notable Ingress solutions are usually HTTP servers such as Nginx, Caddy or cloud provider specific such as AWS Elastic Load Balancer. In order for these to work in a Kubernetes environment, an *Integrator* must be implemented according to Kubernetes specifications. This is called an *Ingress-Controller*.

2.4 Load balancing

Load balancing describes the distribution of tasks and jobs across a pool of computing resources. It allows modern server infrastructures to respond properly to the amount of network load at a given moment. Load-balancers can be either hardware- or software-defined, with layer 4 (transport) or layer 7 (application) balancing capabilities and various distribution algorithms support.[27]
We will discuss specific in-depth features and behavior of the selected solution as part of the main project in chapters 3 and 4. Detailed load-balancers information exceeds the scope of this work.

3 Motivation

Mobile operators often deploy and maintain a high number of edge servers. The highly distributed, available and security critical nature often forces them to create *in-house* solutions. The adoption of software in traditional hardware realms allows great flexibility in network design and operation, e.g *Software Defined Network* (SDN). High observability and thorough monitoring are key in providing maintenance of highly available systems. Unfortunately, operation and management panels are primarily static and are not well suited to track on and act upon modern mobile communication infrastructures.
Nokia has developed a unified control panel for operation and management of mobile operators applications and infrastructures. In this chapter the general requirements from such an application will be listed, and in chapter 4 and 5 we will show how these are met by our work.

3.1 Security and Privacy

Mobile operators enterprises comply to the highest level of network security and privacy. They are responsible for reliable, secure and fail resistant transmission of data. Computer networks are convoluted, require constant maintenance and must be observed and watched constantly. Monitoring solutions such as the control panel belong to the private domain of operations, as the information should not be publicly accessible at any given moment unless explicitly stated. On premise data-centers might save costs and offer full control over all aspects, let alone security and confidentiality, but at the same time they force the mobile operator to upkeep the application and infrastructure, which is not a part of the business model of the enterprise.

3.2 Authentication and Authorization

While partially related to network security and privacy, authorization in large companies is a considerable concern and usually treated separately. The monitoring application is split to different functional modules. *Kong* API Gatway and *Keycloak* pair together to provide strong, industry standard access management mechanisms such as OAuth2.0 and Open Identity Connect.
The clear separation of concerns, and the usage of well defined standards allow replacement of this service in case the need arises, which is an important feature of the mobile telecommunication field. Security standards are being developed and deprecated rapidly, therefore, an affinity towards a single solution prevents an easy swap between security workflows.

3.3 Scalability and Latency

Enterprises are expected to maintain highly available and time-crucial systems, and deliver near real-time communication. Migration of applications to cloud providers enable massive scalability, this is made possible by allocation of highly dynamic virtual computing resources. Kubernetes along with Docker empower cloud service users and abstract the application over the underlying hardware and computing units.

3.4 Infrastructural Flexibility

Cloud infrastructure is expected to abstract underlying systems such as physical and virtual hosts along with storage mediums. It is recommended for applications to tolerate infrastructural changes and prevent *vendor lock-in*[5], due the fact that one cannot assume that a single server- or cloud-provider will satisfy the application's requirements for its entire lifetime, as the these might change constantly. This specific application is being distributed to different mobile operators by Nokia. Thus, it is preferable to allow the customers to select the cloud provider.

4 Migration to the Public Cloud

This chapter will describe the migration process of the application in question from an on-premise data-center to a public cloud. It will also justify the different tools selection being made in order to satisfy the requirements listed in chapter 3.

4.1 Account Permissions and Cluster Setup

Amazon Web Services was the selected cloud provider. It provides a large number of generic services, therefore creating a comprehensive solution suite with little to no

[5]The result of designing a system which is tightly coupled and dependent on a specific vendor

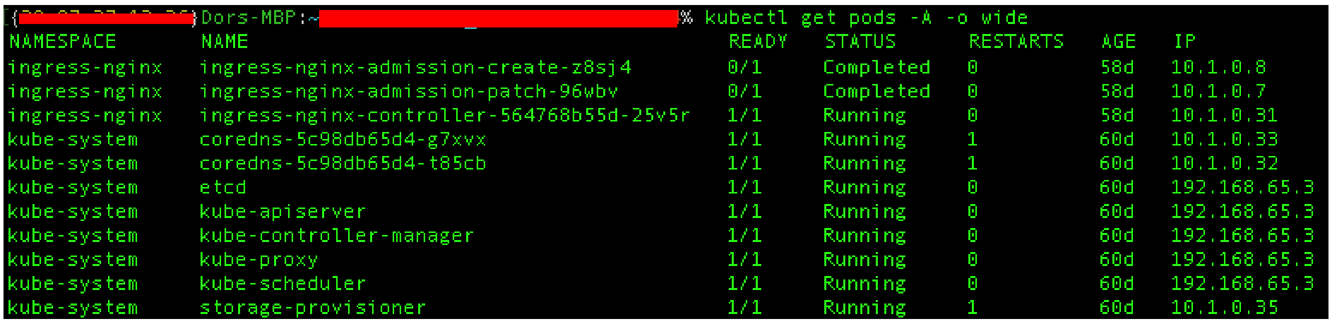
need for external ones. Some of these tools are an inherent part of AWS such as *Identity and Access Management* (IAM), along with various networking related ones such as *Virtual Private Cloud* (VPC) and subnets configurations. These are mentioned in order to show that network isolation from the public internet is a priority when using AWS. *Elastic Kubernetes Service* (EKS) [1] was selected to set-up and configure the Kubernetes cluster on the cloud. *Identity and Access Management* (AWS IAM) [3] allows enterprizes to enforce their permission and boundaries for user and application access. Proper access permissions must be carefully defined in order to provide the application with the minimal set of services allowed without hindering any. Precise *User Roles*, *Policy* and *Role* permissions must be set in order to allow users and applications to use a certain service such as storage mediums, computing units and networking related services on AWS.

Most cloud vendors allow the user to choose between *privately managed* and a *vendor managed* Kubernetes clusters. In a privately managed cluster the customer is responsible for computing resource creation, operating system installations and configurations, Kubernetes runtime installation and management, Kubernetes related dependencies installation such as *etcd* for cluster data storage or *Flannel* as cluster networking overlay. All of these tasks require deep knowledge and experience. In a vendor managed Kubernetes cluster, the complex configuration of worker nodes and cluster is being done by the cloud vendor. In the usual case, the setup will be done against a simple and pre-defined graphical interface which hides the actual complexity of Kubernetes cluster creation and management.
In this exercise the vendor managed cluster was selected. The process of a properly configured cluster creation can be a venture, while vendor-manged clusters provide a certain degree of configuration resilience. A transition between the two is possible in the future in case the need arises, e.g cost-reduction and finer control over the cluster.

The vendor-managed cluster can be made either by command line interface directives or by the AWS web console graphical interface. Although both methods are viable, the web interface prompts the entire initial configuration options such as VPC and subnets selection, along with firewall and access mode (public or private). In a later stage, further cluster creation and management can be automated using the command line interface.
This action will result in creation of the *Kubernetes Control Panel* (Master Nodes) which is managed by AWS and is out of the user's reach. The internal components and responsibilities of the Kubernetes Control Panel are described in chapter 2.

Master nodes perform solely cluster management tasks, and unlike worker nodes, they are provisioned implicitly by AWS EKS. Worker nodes' creation process resembles the cluster creation, as computing resources can be created using the web interface or command line interface. Both ways will result in properly configured and cluster-attached Elastic Cloud Computing resources (EC2), which are essentially virtual computers allocated by AWS. Other alternatives for EC2 such as *Fargate* [6] were considered, but Fargate support for our current use-case was not supported at the

[6]Fargate is a server-less system which decouples the application from any underlying dependencies and allows to treat the application without maintaining any surrounding environment



```
[[redacted]]Dors-MBP:~[redacted]% kubectl get pods -A -o wide
NAMESPACE       NAME                                        READY   STATUS      RESTARTS   AGE   IP
ingress-nginx   ingress-nginx-admission-create-z8sj4        0/1     Completed   0          58d   10.1.0.8
ingress-nginx   ingress-nginx-admission-patch-96wbv         0/1     Completed   0          58d   10.1.0.7
ingress-nginx   ingress-nginx-controller-564768b55d-25v5r   1/1     Running     0          58d   10.1.0.31
kube-system     coredns-5c98db65d4-g7xvx                    1/1     Running     1          60d   10.1.0.33
kube-system     coredns-5c98db65d4-t85cb                    1/1     Running     1          60d   10.1.0.32
kube-system     etcd                                        1/1     Running     0          60d   192.168.65.3
kube-system     kube-apiserver                              1/1     Running     0          60d   192.168.65.3
kube-system     kube-controller-manager                     1/1     Running     0          60d   192.168.65.3
kube-system     kube-proxy                                  1/1     Running     0          60d   192.168.65.3
kube-system     kube-scheduler                              1/1     Running     0          60d   192.168.65.3
kube-system     storage-provisioner                         1/1     Running     1          60d   10.1.0.35
```

Figure 4: Examplary Cluster Pods

time of writing this this work.
Figure 4 shows exemplary output after cluster creation. Each row represents a Pod in the system with additional information such as the namespace it belongs to, and internal IP address in the cluster. Pods under the kube-system namespace are managed by AWS and are running on master nodes, while Pods under other namespaces are in the control of the user. In this case we can observe Nginx Pods which function as a gateway to any application in the cluster.

4.2 Routing and Load Balancing

The application requires dynamic adaption to a changing network load. Kubernetes networking model was described in chapter 2 in order to understand its support of load balancing and robust routing. Complex cloud infrastructure promotes the approach of horizontal scaling, where machines and application-replicas can be created arbitrarily in order to provide seamless operation under heavy load. Kubernetes is capable of scaling the application as needed. It is however not responsible to spread the network traffic across the worker nodes of the cluster without an external dedicated element. This is the duty of the load-balancer.
AWS Elastic Load Balancer was selected as the network load balancer of choice, as it offers easy integration, and quick deployment and modification. Each request sent to the cluster is being piped to the application in the following manner.

4.2.1 First Layer - AWS Elastic Load Balancer

The creation of a load balancer is being triggered by specifying proper annotations in the Kubernetes configuration manifests. This network load balancer is a typical reverse proxy and operates on layer 4 of the OSI model, i.e transport layer. It will forward any network packets (TCP, UDP) to the next routing layer, while distributing the load across the available worker nodes purely by the network load threshold (raw amount of packets) and not by any other metadata the packets might carry. The Elastic Load Balancer is aware of every node in the cluster, offers high availability and periodical health checks of each one. This means that any change in the network e.g a crash fault or a new node spawn will be reacted accordingly.[6]

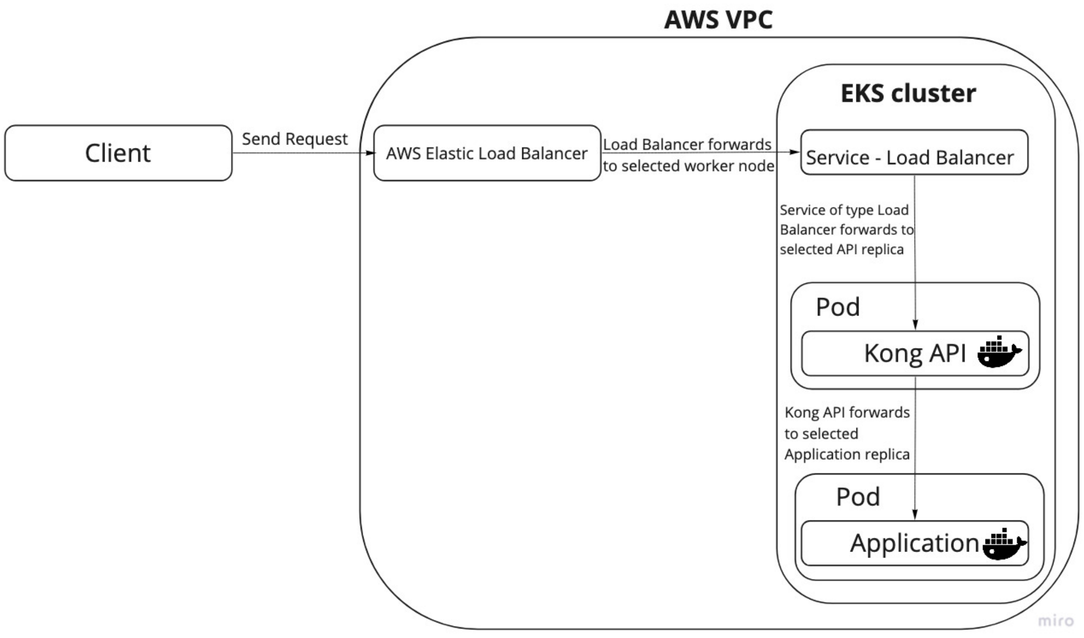

AWS VPC
EKS cluster
Client
Send Request
AWS Elastic Load Balancer
Load Balancer forwards to selected worker node
Service - Load Balancer
Service of type Load Balancer forwards to selected API replica
Pod
Kong API
Kong API forwards to selected Application replica
Pod
Application
miro

Figure 5: Simplified Request Flow with Load Balancing

4.2.2 Second layer - Service Object

The Kubernetes service object grants internal and external network capeabilites to the pod. In this application the service object is of type *LoadBalancer*, which is a requirement for interoperability with AWS Elastic Load Balancer. Therefore, it binds the external load balancing to an internal one, and bridges the abstraction between nodes and pods. It utilizes a plain round-robin load-balancing by default.

4.2.3 Third layer - Kong API Gatway

The API Gatway provides a robust entry-point to our application. It is responsible for operations which are out of the scope of the application itself such as rate limiting, TLS termination and most importantly proper routing of each request to the corresponding application in the cluster. The Kong gateway is being integrated into the Kubernetes cluster by the *kong-kubernetes-controller*, allowing Kong to route by the configurations found in the Ingress objects. The ingress objects will define the entire routing scheme for the gateway to execute, usually by forwarding based on the path value in the URL, i.e level 7 routing and load balancing. Kong will perform load balancing between the replicas of each Pod, attempting to maintain an even load to each instance.

Figure 5 presents the flow of each request into the cluster. Each request reaches the AWS load-balancer, then being forwarded into the cluster. In the cluster the request will be forwarded by the Kubernetes internal load balancer service to the pod containing an instance of Kong API gateway. Lastly it will be forwarded to the pod of the monitoring application itself. The routing flow might appear complex. However, it is an essential trade-off in such systems, as we gain scalability, robustness, and easy deployment of service-oriented systems at the cost of internal network simplicity.

4.3 Access Management

Access management systems control the flow of information and validate the identity of the client [7], but also ensure that this client will be exposed only to allowed resources [8] [25]. Despite the fact that this topic is highly coupled with web- and browser-security, it will be assumed that these are given, and we will only discuss the authentication and authorization flow in this specific implementation.

The generic nature of access management flows caused an industry standardization resulting in creation of protocols such as OAuth/OAuth2, JSON Web Token, SAML, FIDO and more complex ones such as Open ID [21]. Keycloak by Red Hat was utilized as an access management service in this project, running as a separate server in the same cluster, so it is completely isolated from the business logic in the core application server.

Keycloak supports *Single Sign On* (SSO), allowing centralized access management over a multitude of internal services of an enterprize. It also allows *Lightweight Directory Access Protocol* (LDAP) and *Active Directory* for integration with existing user management services and databases. Other features such as *Social Login* and custom login pages themes can be relevant for future uses.

4.3.1 Integration with Kong API

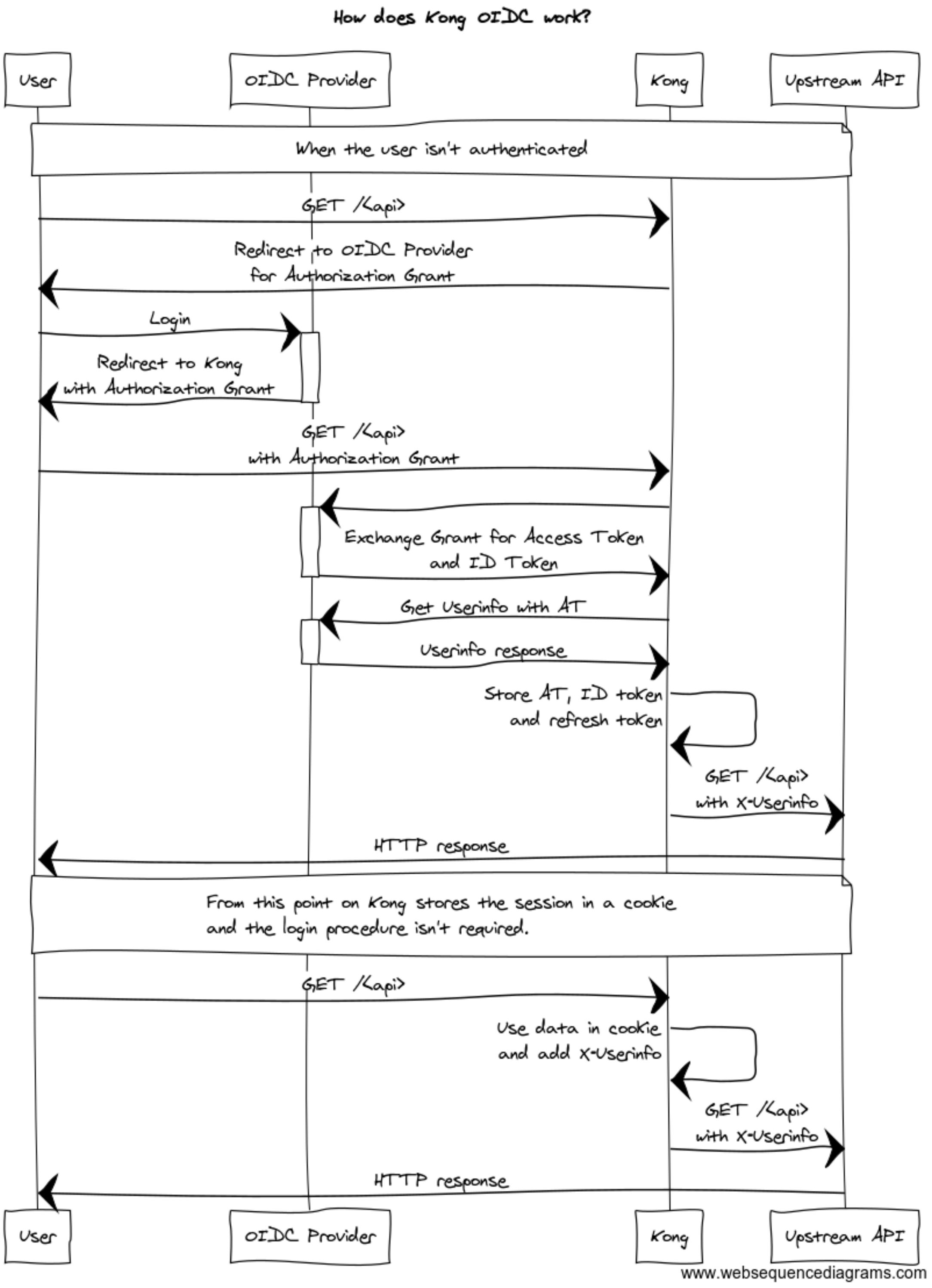
In the last section it was carefully described, how Kong API integrates and interoperates with the cluster network. Keycloak must be integrated in this process in order to allow consistent access management flow and to prevent security pitfalls. The highly extensible format of Kong allows Open ID Connect clients such as Keycloak to be integrated in the usual flow by dedicated plug-ins. Nokia maintains its own open-source Kong plug-in, making its usage and support easy and native [10]. The flow diagram at Figure 6 describes the authentication and authorization protocol, and messages flow between the client and the "Upstream API" i.e our core application.

4.4 Cloud Storage Backend

Traditional web applications are predictable and simple in terms of permanent storage management. In most cases the internal data-flow consists solely of a direct connection between the server-application and the (possibly remote) database-server. This results in a static and simple behavior of the system.

In a Kubernetes based applications, the topology and behavior of the system changes

[7] Authentication
[8] Authorization


How does Kong OIDC work?
User
OIDC Provider
Kong
Upstream API
When the user isn't authenticated
GET /<api>
Redirect to OIDC Provider for Authorization Grant
Login
Redirect to Kong with Authorization Grant
GET /<api> with Authorization Grant
Exchange Grant for Access Token and ID Token
Get Userinfo with AT
Userinfo response
Store AT, ID token and refresh token
GET /<api> with X-Userinfo
HTTP response
From this point on Kong stores the session in a cookie and the login procedure isn't required.
GET /<api>
Use data in cookie and add X-Userinfo
GET /<api> with X-Userinfo
HTTP response
www.websequencediagrams.com

Source: https://github.com/nokia/kong-oidc

Figure 6: Message exchange in Kong-Open ID Connect environment

drastically. Every operable entity in the system is being abstracted in order to provide portability, fault tolerance and scalability. This affects the persistence options of the entire system, since trivial approaches such as a single database server for each service in the cluster would not suffice. In this chapter, we will describe the problem arising from a naive implementation, and a more suitable storage scheme will be reviewed and described.

4.4.1 A Naive Approach - hostPath

Kubernetes supports a wide variety of persistence options, from volatile to permanent[14]. The application was initially using a persistence option called *hostPath*, where a mount point inside the containers was pointing to an actual storage space on the hosting worker node. This option, despite being discouraged by the Kubernetes community did satisfy the requirements in a simple infrastructure, such as a single replica or a single node environments.
When running a multi-node and multi-replicas application, one cannot guarantee that information available on a node in a certain moment will be available later, as the service might already be re-scheduled to a different node. This can happen whenever a container, a pod or a node crashes and the container must be rescheduled. Or when the cluster is operating on a multi-replica mode, scheduling identical replicas of a service to worker nodes, arbitrarily.
Figure 7 illustrates the problem with such permanent storage solution. Pod2 or Node2 crashes, resulting in the kube-controller-manager noticing that the number of wanted replicas - one, does not match the number of current replicas - zero, and will allow the *kube-scheduler* to re-schedule Pod2 to Node1. This leads to a situation where all the information that was persisted on Node2 is not longer accessible from Pod2

4.4.2 A Sustainable Approach - Storage Backend and Dynamic Provisioning

Kubernetes is responsible for application management but not for its persisted data. Yet, it allows integration with external *block-* or *file storage* solutions via compliance with *Cluster Storage Interface* (CSI), so the data persistence layer can be managed in a separate system which is independent of the application's lifetime. From version 1.13 on Kubernetes officially supports *CSI* [5]. Cloud providers often implement this interface, allowing a seamless connection between any Kubernetes cluster and their cloud storage backends. Despite the variety of CSI compliant storage mediums, Elastic Block Store, a native AWS service was selected. It offers an easy deployment and configuration for EC2 based applications such as this one.

Kubernetes defines special objects to allow integration with storage solutions (regardless of cloud based or others), namely *StorageClass*, *PersistentVolume* and *PersistentVolumeClaim*[14]. In a trivial Kubernetes cluster, the storage options will be defined by a PersistentVolume object, describing requirements such as capacity and medium type (e.g hard-drive or solid-state-drive). These storage mediums can be later "claimed" by the application via the PersistentVolumeClaim object. The issue arises from such a flow is the lack of flexibility whenever the storage requirements change. The solution

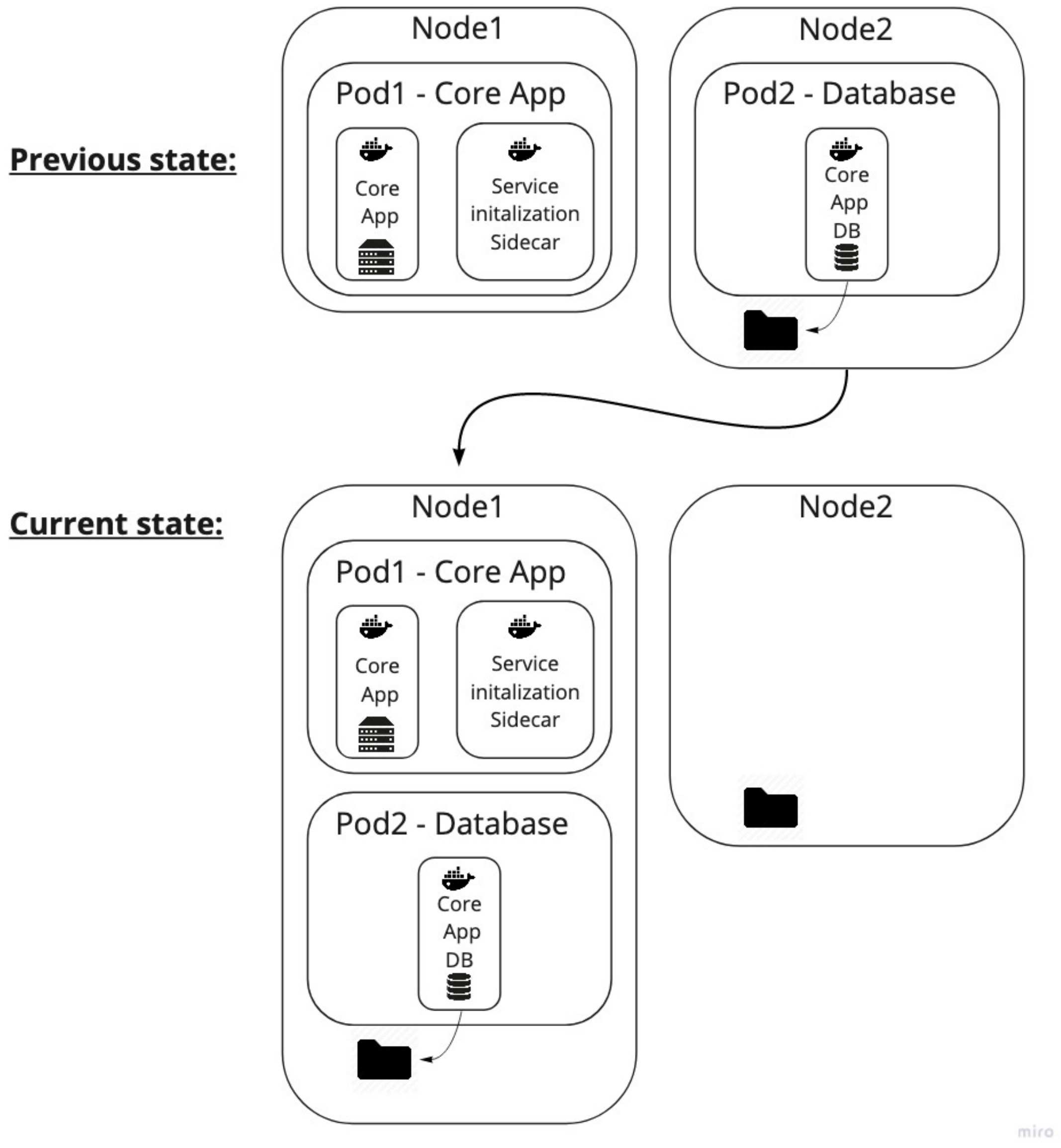


Figure 7: Rescheduling a Pod with containers in a multi-node environment

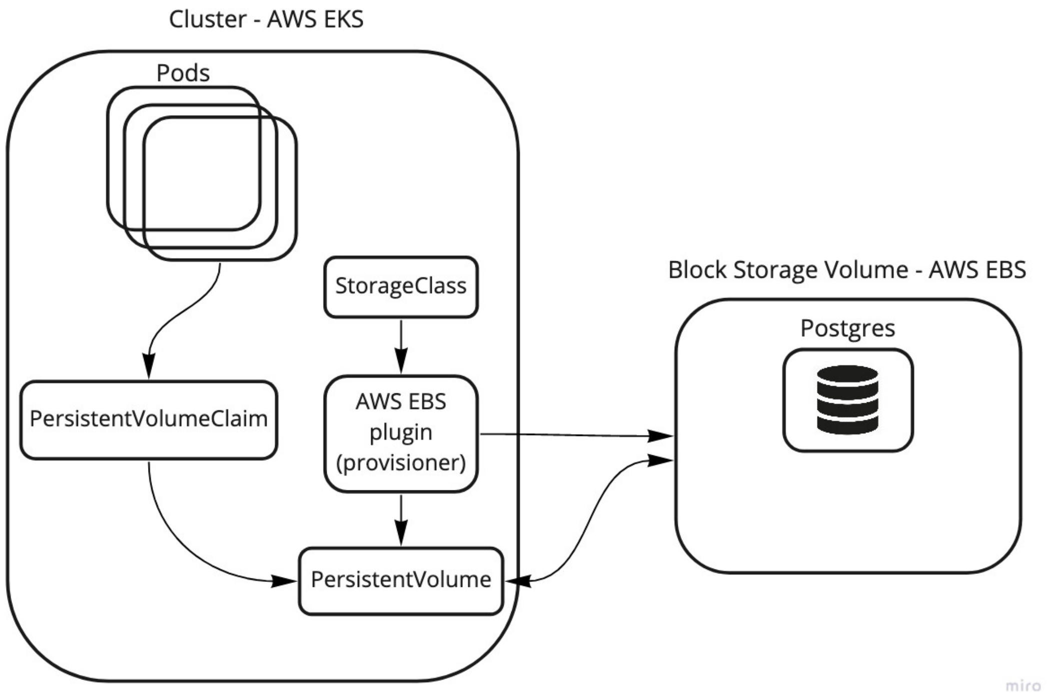


Figure 8: Dynamic provisioning of AWS Elastic Block Storage

to this problem is to allow the application to require the suitable storage whenever needed. The system administrator can set up a StorageClass which defines an entire pool of storage mediums, and by doing so allowing *Dynamic Provisioning* i.e storage allocation as needed.

Figure 8 illustrates the components involved in the dynamic provisioning of persistent volumes on AWS EBS. The server admin defines a StorageClass type object which triggers a provision of an AWS Elastic Block Storage volume by the AWS EBS Plugin and a PersistentVolume. Later, the application developer requests storage resources by defining PersistentVolumeClaims that refer to the auto-generated PersistentVolume.

5 Conclusion and Evaluation

Research and development in large enterprises might be a challenging task. The amount of employees, divisions and stakeholders is immense and dictates its ways of operation. From a technical perspective, such environment affects the tools, software, access permissions and general development procedures, which allow to raise the overall professionalism at the cost of agility and speed.

In chapter 3 we described the requirements expected from the application, and in chapter 4 the detailed implementation and considerations made. These provide sufficient information for a proper reflection.

The application is deployed on AWS in a *Virtual Private Cloud*, providing a complete network isolation. Every virtual host in the private network is explicitly exposed when needed via an integrated firewall. AWS *IAM* is the AWS integrated *Access Management* service, which allows to enforce access rules to the network. Keycloak, the application access management service was entirely migrated. As mentioned, it offers an enterprise grade authorization and authentication, complying to standard such as OAuth2.0 and Open ID Connect. Keycloak is built with cloud compatibility in mind, which promoted its initial selection. The application is containerized and runs on a Kubernetes cluster managed by EKS. This provides a large scalability potential, as there are no hard limitations on the amount of worker nodes that can be run. Kubetnetes orchestrates these nodes, scaling out and in, while restarting resources when required. This allows to maintain acceptable latency and connections threshold under significant load. Kubernetes cluster is portable, as its entire structure is defined by configuration files. A migration to a similar cloud provider should be a relatively simple task. This happens because the architecture is in fact "layered" and each underlying layer is agnostic (to some degree) to the wrapping one, and is in principle independent.
We can see that the requirements stated earlier regarding *Secutiry and Privacy*, *Authentication and Authorization*, *Scalability and Latency* and *Infrastructural Flexibility* are considered as part of the solution and are indeed met to a certain degree.

Considering the scope and requirements of the project, some aspects are left to be desired and should be researched in the future.

The network and message flow scheme is complex (see section 4.3), and involves a fair number of operating parts. These flows might be simplified with more powerful network overlays, which might reduce the steps of message passing inside the private network.
Kubernetes integration with other AWS services such as storage and load-balancing is seemingly streamline. This premise might be correct for the larger cloud providers (AWS, Google Cloud Platform, Microsoft Azure). A flexible and vendor independent solutions to the load balancing and storage backend components might improve portability between cloud vendors and allow better adoption by companies interested in the product but rely on novelty cloud solutions.
The application is designed to be deployed and operated with an arbitrary number of worker nodes and services replicas. However, it was not properly tested beyond two worker nodes and a single replica of each internal service (Kong, Keycloak and core application). This raises a concern regarding the horizontal scalability potential of the application.
A proper method to test the application infrastructure should be proposed and researched, as large migrations involve changes in many infrastructure components. It is desirable to assure that it functions as expected. In this work the testing was manual and not at all automated, which puts the reliability aspect in question. This can be introduced by a *Continuous Integration* and *Continuous Delivery* pipelines which will set up real testing environments for both the application and the infrastructure. For

obvious reasons, this part was as well omitted from the work as it requires specific knowledge and intensive resources.

Overall, we were able to migrate a large software from on-premise data-center to a public cloud provider in a desirable manner which fulfills the initial requirements, and allows integration with other cloud providers in the future. Moreover, the infrastructural tools that were used for the on-premise revision such as Kubernetes, Kong and Keycloak were not compromised. These were adopted fully to the cloud and kept the same predictable application-flow as before.

Bibliography

[1] Amazon elastic kubernetes service. https://aws.amazon.com/eks/ visited on 28/06/2020.

[2] Amazon web services. https://aws.amazon.com/ visited on 31/05/2020.

[3] Aws identity and access management (iam). https://aws.amazon.com/iam/ visited on 28/06/2020.

[4] Cluster networking. https://kubernetes.io/docs/concepts/cluster-administration/networking visited on 31/05/2020.

[5] Container storage interface (csi) for kubernetes ga. https://kubernetes.io/blog/2019/01/15/container-storage-interface-ga visited on 31/05/2020.

[6] Elastic load balancing features. https://aws.amazon.com/elasticloadbalancing/features/ visited on 31/05/2020.

[7] Google cloud platform. https://cloud.google.com/ visited on 31/05/2020.

[8] Keycloak. https://www.keycloak.org/ visited on 31/05/2020.

[9] Kong. https://konghq.com/ visited on 31/05/2020.

[10] Kong oidc plugin. https://github.com/nokia/kong-oidc visited on 20/06/2020.

[11] Kubernetes. https://kubernetes.io/ visited on 31/05/2020.

[12] Kubernetes ingress. https://kubernetes.io/docs/concepts/services-networking/ingress-controllers visited on 31/05/2020.

[13] Kubernetes service - nodeport. https://kubernetes.io/docs/concepts/services-networking/service/#nodeport visited on 31/05/2020.

[14] Kubernetes storage. https://kubernetes.io/docs/concepts/storage/ visited on 28/06/2020.

[15] Linux kernel in wsl 2. https://docs.microsoft.com/en-us/windows/wsl/wsl2-about#linux-kernel-in-wsl-2 visited on 22/06/2020.

[16] Lxc. https://linuxcontainers.org/#LXC visited on 31/05/2020.

[17] Managing resources. https://kubernetes.io/docs/concepts/cluster-administration/manage-deployment visited on 22/07/2020.

[18] Microsoft azure. https://azure.microsoft.com visited on 31/05/2020.

[19] Open container initiative. https://opencontainers.org/ visited on 31/05/2020.

[20] Overview of containers in red hat systems. 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#linux_containers_architecture visited on 22/04/2020.

[21] Owasp authentication cheat sheet. https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html visited on 10/06/2020.

[22] Serverless architecture. https://martinfowler.com/articles/serverless.html visited on 31/05/2020.

[23] Reminiscences on the theory of time-sharing, March 1989. http://aiweb.techfak.uni-bielefeld.de/content/bworld-robot-control-software/.

[24] Docker documentation, 04 2020. https://docs.docker.com/engine/.

[25] David J. Wetherall Andrew S. Tanenbaum. *Computer Networks 5th Edition*, chapter 8, page 827. Pearson Prentice Hall, 1 edition, 2006.

[26] Ed. C. Yang, Ed. SY. Pan. The standards on a cloud service framework and protocol for construction, migration, deployment,and publishing of internet-oriented scalable web software systems in non-programming mode. Internet-Draft 4180, RFC Editor, 10 2005. https://tools.ietf.org/id/draft-yangcan-core-web-software-built-in-cloud-02.html/.

[27] Shahbaz Afzal G. Kavitha. An updated performance comparison of virtual machines and linux containers. *Journal of Cloud Computing*, 2019.

[28] Justin Garrison and Kris Nova. *Cloud Native Infrastructure*, chapter 1, page 22. O'reilly Media, Inc., 1 edition, 07 2018.

[29] Justin Garrison and Kris Nova. *Cloud Native Infrastructure*, chapter 1, page 26. O'reilly Media, Inc., 1 edition, 07 2018.

[30] Marko Luksa. *Kubernetes in Action*, chapter 1, pages 18–19. Hanning, 1 edition, 2018.

[31] Marko Luksa. *Kubernetes in Action*, chapter 1, pages 55–309. Hanning, 1 edition, 2018.

[32] Ram Rajamony Juan Rubio Wes Felter, Alexandre Ferreira. An updated performance comparison of virtual machines and linux containers. *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172, 2015.