TECHNICAL UNIVERSITY OF CRETE

Fail Safe Container Scheduling in Kubernetes

Author:
Christos V. CHRISTODOULOPOULOS

Supervisor:
Euripides G.M. PETRAKIS

A thesis submitted in partial fulfillment of the requirements for the diploma of Electrical & Computer Engineer

December 2017

“Kept you waiting, huh ?”

Big Boss

Abstract

Kubernetes is a tool for facilitating deployment of multiple OS system-level virtualized applications using containers. It is capable for the management of several containerized applications and users at the same time allowing compute resources to be managed and distributed to the applications using a scheduling mechanism. The scheduling mechanism is also responsible for re-scheduling of compute resources per application based on the actual needs of each application at run-time. However, resource allocation in a typical Kubernetes environment is rather static (i.e. the maximum amount of compute resources that each application can use has to be known in advance) meaning that if the application requests more resources than the maximum, a failure scheduling event will be generated. Although solutions to the problem of automatic scaling of resources in Kubernetes are known to exist and auto scaling is supported by cloud providers such as Amazon and Google, these solutions are fully proprietary and not always generic (i.e. do not apply to all Kubernetes distributions). This is exactly the problem this work is dealing with. We propose Commodore, a mechanism that is platform independent (i.e. can work with any Kubernetes distribution including non-cloud based implementations). Commodore is capable of allocating (or de-allocating) resources to applications based on their actual demands. To show proof of concept, Commodore is implemented on a Fiware cloud platform running on Openstack. In this environment, application services are deployed on worker machines (nodes) which are realized as Virtual Machines (VMs). This way, the implementation takes advantage of the virtualization features of cloud computing (i.e. allows for definition of virtualized resources including network, cpu and memory leading to better utilization of physical resources while ensuring software security by isolating services and applications from each other). We run several experiments based on a simulated (but realistic) use case scenario. The experimental results demonstrated that Commodore responds to the increasing (or decreasing) resource demands of the application leading to significantly faster response times compared to a non-auto scaled implementation where all service requests are handled by the maximum statically pre-allocated resources.

Acknowledgements

I would like to express my gratitude to my supervisor, Prof. Euripides Petrakis, for his encouragement, his continuous guidance and support throughout my thesis. Special thanks go to Spyros Argyropoulos for his support and his valuable advises. I would also like to thank Prof. Minos Garofalakis and Prof. Stelios Sotiriadis for serving on my thesis committee. My sincerest gratitude also goes to all my friends for the memorable moments we lived together during our studies. Finally, this thesis would not have been possible without the support and encouragement of my parents. My mother Rebekka and my father Vasileios were always there whenever I need them. Every hour, every day, every moment were there and constantly support me. Thank you from the bottom of my heart for your never ending love and care. I am grateful to you...

Contents

List of Figures

Dedicated to my family…

Chapter 1

Introduction

Scalability refers to the ability of a system to continue to function well when its size (i.e., in terms of computing resources) has to change (increase or decrease) in order to respond to increasing or decreasing workloads. For example, a scalable database management system (DBMS) should be able to efficiently expand the size of the datastore as more data is added to the database, or a scalable Web application should be able to serve an increasing number of users. Scalablity can appear as a problem of infrastructure, where the lack of compute resources can decrease the performance of the system or as a problem of architecture in the case where the design of the system itself does not allow it to scale. This work focuses on the infrastructure aspect of the problem.

Back in the days, when an application was experiencing increased load, the addition of resources (vertical scaling) and a reconfiguration of the run-time environment would solve the problem. In cases where this wasn't enough, more instances of the application have to be spawned, most likely in different physical or virtual machines (horizontal scaling). The introduction of cloud computing along with virtualization technology made such operations easier and faster to be applied by providing an abstraction layer that decouples the physical hardware from the environment that runs on top of it and by supporting features like isolation and resource customization.

Nowadays, with the advent of the internet, applications can utilize hundreds or even thousands of machines. Furthermore, hardware virtualization is not a sufficient abstraction anymore, since machines can be powerful enough to host multiple applications, thus hosting each application separately on a different machine would result to under-utilization of the infrastructure and consequently to the increase of operational costs. On the other hand, hosting multiple applications on the same machine compromises application isolation, a fundamental feature of cloud-based environments. The introduction of containers [1] addresses this limitation by providing the benefits of cloud virtualization in a more light-weight manner. As illustrated in Figure 1.1 containers introduce a run-time environment which adds an abstraction layer between the (physical or virtual) machine's operating system and the application. Containers can be configured in order to host an application and

[1]https://www.docker.com/what-container

can be deployed to any machine since they package the application as a self-contained entity with no external dependencies.

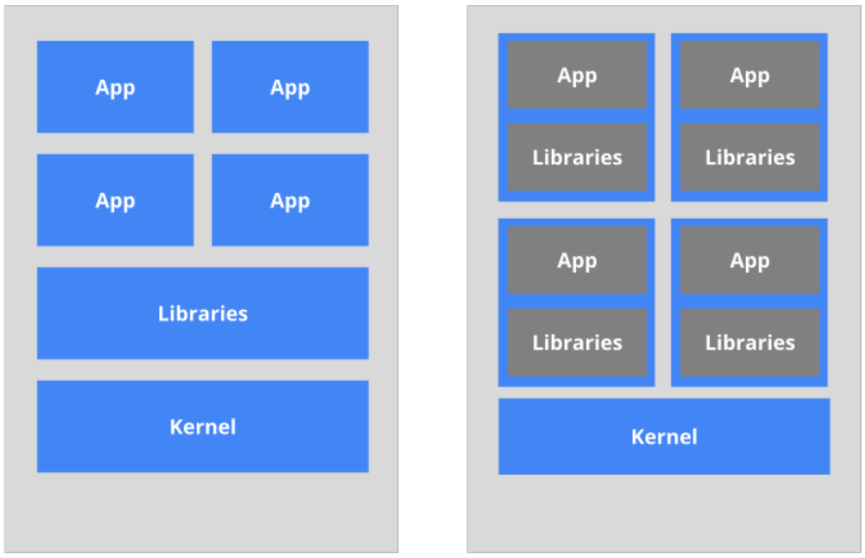


FIGURE 1.1: Multiple applications on a host (left) versus multiple containerized applications on a host (right).

Despite their benefits to application development and deployment, they do not solve the problem of vertical scalability (i.e. cannot scale up automatically to a bigger or smaller size) nor the problem of horizontal scalability (i.e. cannot scale in size automatically by utilizing more compute resources such as memory or cpu) according to workload demands. The introduction of container orchestration and management tools, like Kubernetes[2], automate the deployment, scaling and management of containerized applications.

1.1 Problem Definition

Kubernetes provides the means for describing, deploying and managing containerized applications. Each application can run in many copies. Applications run in clusters of nodes (each node is typically deployed as a virtual machine). Each node runs clusters of containers referred to as pods; each pod is deployed as a separate Docker environment running one or more containers. After providing that description, Kubernetes resumes responsibility for scheduling and managing each application and their replicas inside each cluster. Kubernetes can also be configured to balance the traffic across clusters.

Consider a two node [3] Kubernetes cluster with an application deployed as illustrated in Figure 1.2. The nodes are identical in terms of compute resources. The application is deployed in 3 replicas (pods [4]) running in these two nodes

[2]https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/
[3]A physical or virtual machine
[4]https://kubernetes.io/docs/concepts/workloads/pods/pod/

(i.e., one copy of application in Node1 and two copies of the same application in Node2). Each pod consumes half of the compute resources of each node. Now consider the following scenario:

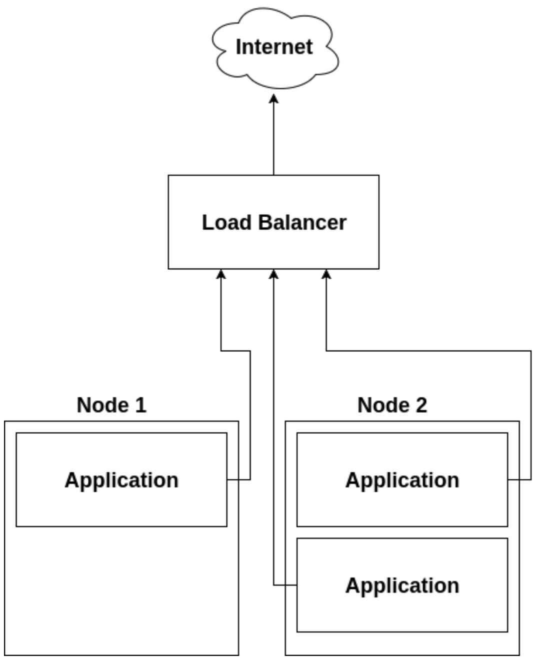


FIGURE 1.2: Example deployment with three replicas.

1. At peak traffic hours, the incoming traffic increases and the application experiences heavy load resulting in longer response times as the three replicas can't cope with that amount of load.
2. The system administrator observes the derating in performance and decides to increase the number of the replicas of the application from three to five in order reduce the work load per running application.
3. The system administrator instructs Kubernetes to schedule two new replicas of the application in the existing nodes.
4. Kubernetes tries to schedule the two new replicas to the cluster. The first replica is scheduled correctly but the second cannot since the cluster has no available resources left as Figure 1.3 shows.
5. Kubernetes emits an event that a replica couldn't be scheduled on any of the existing nodes and doesn't take any further action.

As the above example shows, when a change of state occurs in Kubernetes (in our case application scaling) regardless of how it was initiated (in our case by the system administrator), it doesn't necessarily mean that it can be satisfied. In the above example, the reason the operation failed was the lack of compute resources. So, a mechanism that is capable of providing the additional resources, when needed, is mandatory. This is the problem this work is dealing with that is, the problem of infrastructure scalability on a virtualized cloud environment managed by Kubernetes.

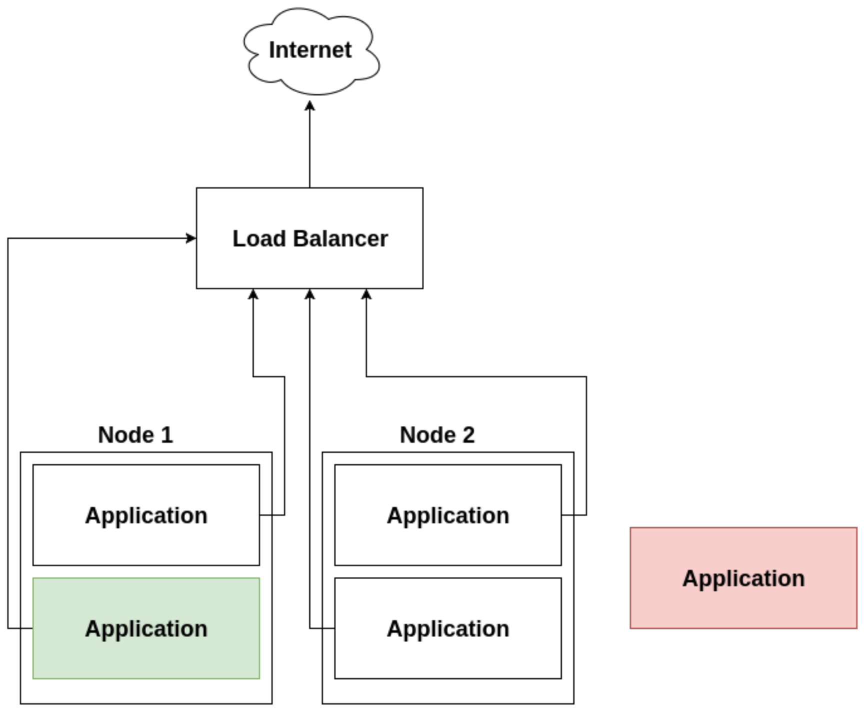


FIGURE 1.3: Example deployment with five replicas.

1.2 Proposed Solution

In this work we design and implement a service that is capable of provisioning and de-provisioning compute resources in a Kubernetes cluster automatically (i.e. without user intervention), by taking into consideration the cluster state and load.

Continuing our previous example scenario but with such service in place:

1. The system receives the event emitted from Kubernetes regarding the scheduling failure.
2. The service contacts Kubernetes about the compute resources that are needed in order to deploy the failing replica correctly.
3. The service inquires the infrastructure provider (cloud provider in our case) for a (virtual) machine with the compute resources that Kubernetes needs.
4. The new machine is initialized and binds to the cluster.
5. Kubernetes schedules the new replica of the application to run in the new machine as Figure 1.4 shows.

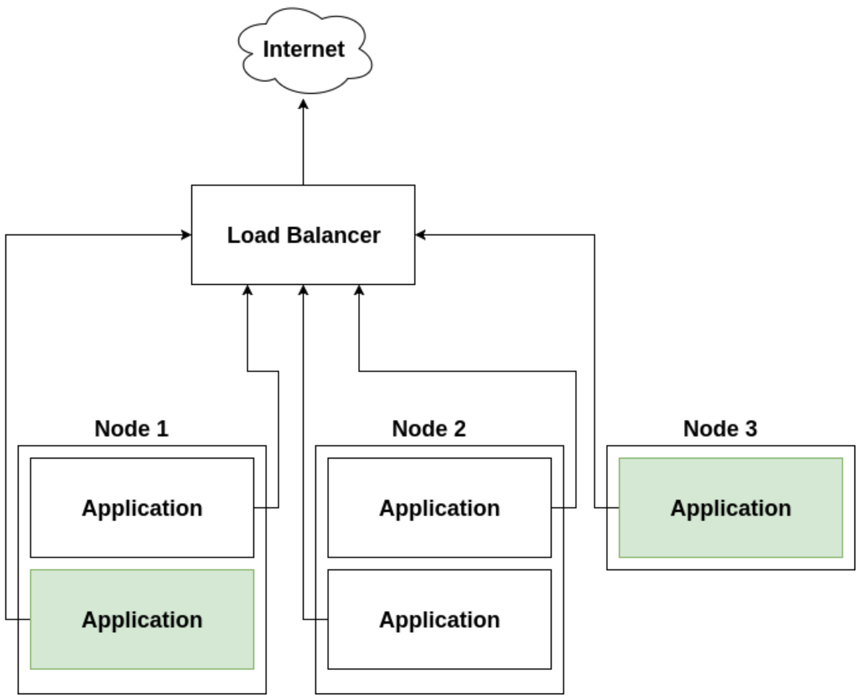


FIGURE 1.4: Example deployment with five replicas scheduled.

1.3 Summary of Contributions

This thesis introduces Commodore. Commodore is a complement system to Kubernetes and provides Kubernetes with infrastructure scalability: allocation of additional compute resources (as Kubernetes nodes) for dealing with increased workloads. Similarly, Commodore, is capable of dealing with decreasing computational demands in which case, unused nodes are freed. The way of doing so is by dynamically provisioning and de-provisioning resources based on the cluster load. Resources are provisioned in the context of virtual machines, i.e. additional Kubernetes nodes. In other words, Commodore adds or removes worker nodes (virtual machines) on a Kubernetes cluster based on the resources that the deployed applications need.

Existing solutions to autoscaling by Amazon and Google are either proprietary (Amazon) or bound to specific infrastructure (Google, Amazon). Commodore is, to the best of our knowledge, the fist solution to the problem of autoscaling which is both, platform independent and open source (working on the Kubernetes distribution available on Github [5]. Commodore is also compatible with any Kubernetes distribution since it uses standardized API calls in order to communicate with the cluster.

Commodore decisions for infrastructure scaling depend upon the scheduling failure events received from Kubernetes management service (in response to pod resource requests) rather than on actual resource utilization measurements as Amazon's solution does. This is a design decision that can be

[5]https://github.com/kubernetes/kubernetes

changed in later implementations of Commodore. The rationale behind this decision is to focus (at this stage) on the service aspects of autoscaling in Kubernetes rather than on elaborate decision making mechanisms that apart from resource utilization need to take decision on optimal placement (or re-placement) of workload among nodes or pods. This is rather complicated task that needs careful consideration and is outside the scope of this thesis.

1.4 Thesis Outline

This thesis is structured as follows:

- Chapter 2 provides a brief overview of the technologies used for the implementation of the system. These include advances in open source cloud technologies (OpenStack), containerized application deployment (Docker), container orchestration tools (Kubernetes) and No-SQL databases (Cassandra).
- Chapter 3 presents research which is most relevant to the problem being solved.
- Chapter 4 presents an architectural overview of our proposed system and its internal services.
- Chapter 5 explains the implementation details for each sub module of our system.
- Chapter 6 discusses how a typical application can be deployed on a Kubernetes cluster and presents the evaluation process we followed for testing the performance of Commodore. The experimental results demonstrate advantages and improved performance compared to the existing Kubernetes set-up (with no scale support).
- Chapter 7 discusses contributions and key features of this work along with issues for improvements and future work.

Chapter 2

Background

This chapter presents a brief overview of the standards and technologies used in this thesis. Section 2.1 gives an overview of the OpenStack cloud infrastructure and presents some of the core components. Section 2.2 describes the container specification, an operating system-level virtualization technology, as well as an one of the most adopted implementations of this field. Section 2.3 describes Kubernetes i.e. a container orchestration tool that our work is based on, and finally section 2.4 describes the database we chose for our implementation. Parts of this text are from [2], [3] and [1].

2.1 Cloud

The cloud has risen in popularity and function in the past few years. Storing data and consuming computing resources on a third party's hardware reduces the overhead of operations by keeping the number of people and owned assets low. For a small company, this could be an opportunity to expand operations, whereas for a large company, this could help to streamline costs. The cloud not only abstracts the management of the hardware that an end user consumes, it also creates an on-demand provisioning capability that was previously not available to consumers. Traditionally, provisioning new hardware or virtualized hardware was a fairly manual process that would often lead to a backlog of requests, thus stigmatizing this way of provisioning resources as a slow process.

The cloud grew in popularity mostly as a public offering in the form of services accessible to anyone on the Internet and operated by a third party. This paradigm has implications for how data is handled and stored and requires a link that travels over the public Internet for a company to access the resources they are using. These implications translate into questions of security for some use cases. As the adoption of the public cloud increased in demand, a private cloud was birthed as a response to addressing these security implications. A private cloud is a cloud platform operated without a public connection, inside a private network. By operating a private cloud, the speed of on-demand visualization and provisioning could be achieved without the risk of operating over the Internet, paying for some kind of private connection to a third party, or the concern of private data being stored by a third-party provider.

2.1.1 OpenStack

OpenStack is a cloud platform which began as a joint project between NASA and Rackspace. It was originally intended to be an open source alternative that has compatibility with the Amazon Elastic Compute Cloud (EC2) cloud offering. Today, OpenStack has become a key player in the cloud platform industry. It is in its seventh year of release, and it continues to grow and gain adoption both in its open source community and the enterprise market.

OpenStack has a very modular design, and because of this design, there are lots of moving parts. It's overwhelming to start walking through installing and using OpenStack without understanding the internal architecture of the components that make up OpenStack. In this chapter, we'll look at these components. Each component in OpenStack manages a different resource that can be virtualized for the end user. Separating each of the resources that can be virtualized into separate components makes the OpenStack architecture very modular. If a particular service or resource provided by a component is not required, then the component is optional to an OpenStack deployment. Let's start by outlining some simple categories to group these services into.

2.1.2 OpenStack architecture

Logically, the components of OpenStack can be divided into three groups:

- Control
- Network
- Compute

The control tier runs the Application Programming Interfaces (API) services, web interface, database, and message bus. The network tier runs network service agents for networking, and the compute node is the virtualization hypervisor. It has services and agents to handle virtual machines. All of the components use a database and/or a message bus. The database can be MySQL, MariaDB, or PostgreSQL. The most popular message buses are RabbitMQ, Qpid, and ActiveMQ. For smaller deployments, the database and messaging services usually run on the control node, but they could have their own nodes if required.

Now that a base logical architecture of OpenStack is defined, let's look at what components make up this basic architecture. To do that, we'll first touch on the web interface and then work towards collecting the resources necessary to launch an instance. Finally, we will look at what components are available to add resources to a launched instance.

Dashboard

The OpenStack dashboard is the web interface component provided with OpenStack. Sometimes the terms dashboard and Horizon used interchangeably. Technically, they are not the same thing. The team that develops the web interface maintains both the dashboard interface and the Horizon framework that the dashboard uses.

The dashboard cannot do anything that the API cannot do. All the actions that are taken through the dashboard result in calls to the API to complete the task requested by the end user. Next, we will discuss both the dashboard and the underlying components that the dashboard makes calls to when creating OpenStack resources.

Keystone

Keystone is the identity management component. The first thing that needs to happen while connecting to an OpenStack deployment is authentication. In its most basic installation, Keystone will manage tenants, users, and roles and be a catalog of services and endpoints for all the components in the running cluster.

Everything in OpenStack must exist in a tenant. A tenant is simply a grouping of objects. Users, instances, and networks are examples of objects. They cannot exist outside of a tenant. Another name for a tenant is project. On the command line, the term tenant is used. In the web interface, the term project is used.

Users must be granted a role in a tenant. A user cannot log in to the cluster unless they are members of a tenant. Even the administrator has a tenant. Even the users the OpenStack components use to communicate with each other have to be members of a tenant to be able to authenticate.

Keystone also keeps a catalog of services and endpoints of each of the OpenStack components in the cluster. This is advantageous because all of the components have different API endpoints. By registering them all with Keystone, an end user only needs to know the address of the Keystone server to interact with the cluster. When a call is made to connect to a component other than Keystone, the call will first have to be authenticated, so Keystone will be contacted regardless.

Within the communication to Keystone, the client also asks Keystone for the address of the component the user intended to connect to. This makes managing the endpoints easier. If all the endpoints were distributed to the end users, then it would be a complex process to distribute a change in one of the endpoints to all of the end users. By keeping the catalog of services and endpoints in Keystone, a change is easily distributed to end users as new requests are made to connect to the components.

By default, Keystone uses username/password authentication to request a token and Public Key Infrastructure (PKI) tokens for subsequent requests. The token has a user's roles and tenants encoded into it. All the components in the cluster can use the information in the token to verify the user and the user's access. Keystone can also be integrated into other common authentication systems instead of relying on the username and password authentication provided by Keystone.

Glance

Glance is the image management component. Once we're authenticated, there are a few resources that need to be available for an instance to launch. Before a server is useful, it needs to have an operating system installed on it. This is a boilerplate task that cloud computing has streamlined by creating a registry of pre-installed disk images to boot from. Glance serves as this registry within an OpenStack deployment. In preparation for an instance to launch, a copy of a selected Glance image is first cached to the compute node where the instance is being launched. Then, a copy is made to the ephemeral disk location of the new instance. Subsequent instances launched on the same compute node using the same disk image will use the cached copy of the Glance image.

The images stored in Glance are sometimes called sealed-disk images. These images are disk images that have had the operating system installed but have had things such as Secure Shell (SSH) host key, and network device MAC addresses removed. This makes the disk images generic, so they can be reused and launched repeatedly without the running copies conflicting with each other. To do this, the host-specific information is provided or generated at boot. The provided information is passed in through a post-boot configuration facility called cloud-init.

The images can also be customized for special purposes beyond a base operating system install. If there was a specific purpose for which an instance would be launched many times, then some of the repetitive configuration tasks could be performed ahead of time and built into the disk image. For example, if a disk image was intended to be used to build a cluster of web servers, it would make sense to install a web server package on the disk image before it was used to launch an instance. It would save time and bandwidth to do it once before it is registered with Glance instead of doing this package installation and configuration over and over each time a web server instance is booted.

Neutron

Neutron is the network management component. With Keystone, we're authenticated, and from Glance, a disk image will be provided. The next resource required for launch is a virtual network. When an OpenStack deployment is using Neutron, it means that each of your tenants can create virtual isolated networks. Each of these isolated networks can be connected to virtual routers to create routes between the virtual networks. A virtual router can have an external gateway connected to it, and external access can be given to each instance by associating a floating IP on an external network with an instance. Neutron then puts all configuration in place to route the traffic sent to the floating IP address through these virtual network resources into a launched instance. This is also called Networking as a Service (NaaS). NaaS is the capability to provide networks and network resources on demand via software.

By default, the OpenStack distribution we will install uses Open vSwitch to orchestrate the underlying virtualized networking infrastructure. Open vSwitch is a virtual managed switch. As long as the nodes in your cluster have simple connectivity to each other, Open vSwitch can be the infrastructure configured to isolate the virtual networks for the tenants in OpenStack. There are also many vendor plugins that would allow you to replace Open vSwitch with a physical managed switch to handle the virtual networks. Neutron even has the capability to use multiple plugins to manage multiple network appliances. As an example, Open vSwitch and a vendor's appliance could be used in parallel to manage virtual networks in an OpenStack deployment. This is a great example of how OpenStack is built to provide flexibility and choice to its users.

Nova

Nova is the instance management component. An authenticated user who has access to a Glance image and has created a network for an instance to live on is almost ready to tie all of this together and launch an instance. OpenStack will allow you to import your own SSH key pair or generate one to use. When the instance is launched, the public key is placed in the authorized_keys file so that a password-less SSH connection can be made to the running instance.

Before that SSH connection can be made, the security groups have to be opened to allow the connection to be made. A security group is a firewall at the cloud infrastructure layer. The OpenStack distribution we'll use will have a default security group with rules to allow instances to communicate with each other within the same security group, but rules will have to be added for Internet Control Message Protocol (ICMP), SSH, and other connections to be made from outside the security group.

Once there's an image, network, key pair, and security group available, an instance can be launched. The resource's identifiers are provided to Nova, and Nova looks at what resources are being used on which hypervisors, and schedules the instance to spawn on a compute node. The compute node gets the Glance image, creates the virtual network devices, and boots the instance. During the boot, cloud-init should run and connect to the metadata service. The metadata service provides the SSH public key needed for SSH login to the instance and, if provided, any post-boot configuration that needs to happen. This could be anything from a simple shell script to an invocation of a configuration management engine.

Cinder

Cinder is the block storage management component. Volumes can be created and attached to instances. Cinder also handles snapshots. Snapshots can be taken of the block volumes or of instances. Instances can also use these snapshots as a boot source.

There is an extensive collection of storage backends that can be configured as the backing store for Cinder volumes and snapshots. By default, Logical Volume Manager (LVM) is configured. GlusterFS and Ceph are two popular software-based storage solutions. There are also many plugins for hardware appliances.

Swift

Swift is the object storage management component. Object storage is a simple content-only storage system. Files are stored without the metadata that a block filesystem has. These are simply containers and files. The files are simply content. Swift has two layers as part of its deployment: the proxy and the storage engine. The proxy is the API layer. It's the service that the end user communicates with. The proxy is configured to talk to the storage engine on the user's behalf. By default, the storage engine is the Swift storage engine. It's able to do software-based storage distribution and replication. GlusterFS and Ceph are also popular storage backends for Swift. They have similar distribution and replication capabilities to those of Swift storage.

Ceilometer

Ceilometer is the telemetry component. It collects resource measurements and is able to monitor the cluster. Ceilometer was originally designed as a metering system for billing users. As it was being built, there was a realization that it would be useful for more than just billing and turned into a general-purpose telemetry system.

Ceilometer meters measure the resources being used in an OpenStack deployment. When Ceilometer reads a meter, it's called a sample. These samples get recorded on a regular basis. A collection of samples is called a statistic. Telemetry statistics will give insights into how the resources of an OpenStack deployment are being used. The samples can also be used for alarms. Alarms are nothing but monitors that watch for a certain criterion to be met. These alarms were originally designed for Heat autoscaling.

Heat

Heat is the orchestration component which enables launching multiple instances that are intended to work together. In orchestration, there is a file, known as a template, used to define what will be launched. In this template, there can also be ordering or dependencies set up between the instances. Data that needs to be passed between the instances for configuration can also be defined in these templates. Heat is also compatible with AWS CloudFormation templates and implements additional features in addition to the AWS CloudFormation template language.

To use Heat, one of these templates is written to define a set of instances that needs to be launched. When a template launches a collection of instances, it's called a stack. When a stack is spawned, the ordering and dependencies, shared conflagration data, and post-boot configuration are coordinated via Heat. Heat is not configuration management. It is orchestration. It is intended to coordinate launching the instances, passing configuration data, and executing simple post-boot configuration. A very common post-boot configuration task is invoking an actual configuration management engine to execute more complex post-boot configuration.

2.2 Containers

Containers have a long history in computing. Unlike hypervisor virtualization, where one or more independent machines run virtually on physical hardware via an intermediation layer, containers run in user space on top of an operating system's kernel. As a result, container virtualization is often called operating system-level virtualization. Container technology allows multiple isolated user space instances to run on a single host.

Containers have been deployed in a variety of use cases. They are popular for hyperscale deployments of multi-tenant services, for lightweight sandboxing, and, despite concerns about their security, as process isolation environments. Indeed, one of the more common examples of a container is a chroot jail, which creates an isolated directory environment for running processes. Attackers, if they breach the running process in the jail, then find themselves trapped in this environment and unable to further compromise a host.

More recent container technologies have included OpenVZ, Solaris Zones, and Linux containers like lxc. Using these more recent technologies, containers can now look like full-blown hosts on their own right rather than just execution environments. In the case of Docker[1], having modern Linux kernel features, such as control groups and namespaces, means that containers can have strong isolation, their own network and storage stacks, as well as resource management capabilities to allow friendly co-existence of multiple containers on a host.

Containers are generally considered a lean technology because they require limited overhead. Unlike traditional virtualization or paravirtualization technologies, they do not require an emulation layer or a hypervisor layer to run and instead use the operating system's normal system call interface. This reduces the overhead required to run containers and can allow a greater density of containers to run on a host. Despite their history containers haven't achieved large-scale adoption. A large part of this can be laid at the feet of their complexity: containers can be complex, hard to set up, and difficult to manage and automate. Docker aims to change that.

2.2.1 Docker

Docker is an open-source engine that automates the deployment of applications into containers. It was written by the team at Docker, Inc (formerly dotCloud Inc, an early player in the PAAS market), and released under the Apache 2.0 license. Docker adds an application deployment engine on top of a virtualized container execution environment. It is designed to provide a lightweight and fast environment in which to run a code as well as an efficient workflow to get that code from e.g a laptop to a test environment and then into production. Docker is incredibly simple.

Docker is fast: an application can be Dockerized in minutes. Docker relies on a copy-on-write model so that making changes to an application is also incredibly fast: only what the user needs to change gets changed. The user can then create containers running applications. Most Docker containers take less than a second to launch. Removing the overhead of the hypervisor also means containers are highly performant so that more of them can be packed into hosts making the best possible use of compute resources.

With Docker, Developers care about their applications running inside containers, and Operations cares about managing the containers. Docker is designed to enhance consistency by ensuring the environment in which developers write code matches the environments into which your applications are deployed.

Docker aims to reduce the cycle time between code being written and code being tested, deployed, and used. It aims to make applications portable,

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

easy to build, and easy to collaborate on. Docker also encourages service-oriented and microservices architectures. Docker recommends that each container run a single application or process. This promotes a distributed application model where an application or service is represented by a series of inter-connected containers. This makes it easy to distribute, scale, debug and introspect your applications.

2.2.2 Docker Components

The core components that compose Docker are:

- The Docker client and server, also called the Docker Engine.
- Docker Images
- Registries
- Docker Containers

Docker client and server

Docker is a client-server application. The Docker client talks to the Docker server or daemon, which, in turn, does all the work. Sometimes the Docker daemon is referred to as Docker Engine. Docker ships with a command line client binary, *docker*, as well as a full RESTful API to interact with the daemon. Docker daemon and client can run on the same host or connect a local Docker client to a remote daemon running on another host.

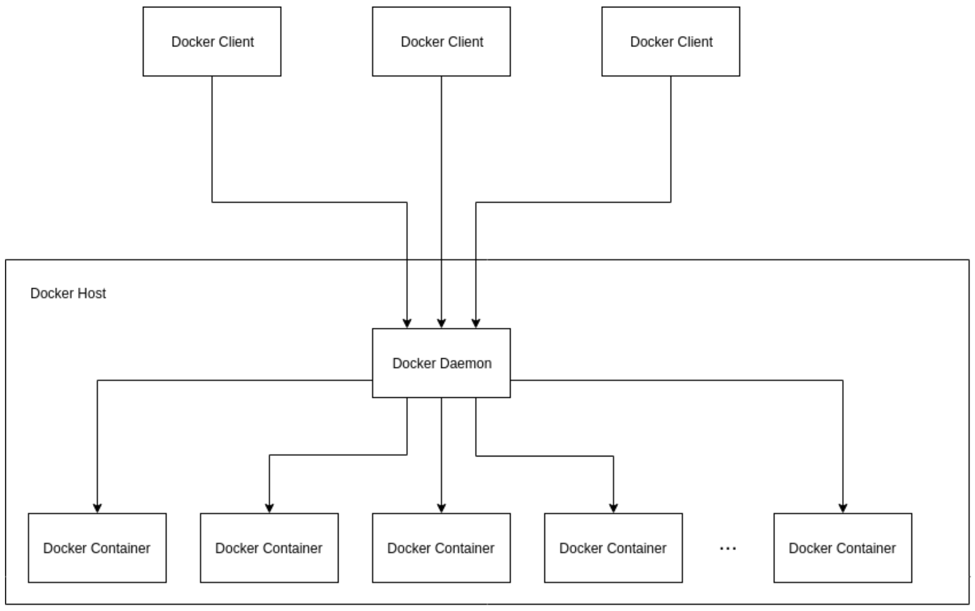


FIGURE 2.1: Docker architecture.

Docker Images

Images are the building blocks of the Docker world. Containers are launched from images. Images are the "build" part of Docker's life cycle. They are a layered format, using Union file systems, that are built step-by-step using a series of instructions. For example:

- Add a file.
- Run a command.
- Open a port.

Images are considered to be the "source code" for containers. They are highly portable and can be shared, stored, and updated.

Registries

Docker stores the images in registries. There are two types of registries: public and private. Docker, Inc., operates the public registry for images, called the Docker Hub. A user creates an account on the Docker Hub and uses it to share and store images. Docker Hub also contains, at last count, over 10,000 images that other people have built and shared. Images can also be stored privately on Docker Hub. These images might include source code or other proprietary information that is stored securely or shared with other members of a team or organization. A user can also run his own private registry. This allows to store images behind a firewall, which may be a requirement for some organizations.

Containers

Docker helps a user build and deploy containers inside of which applications and services are packed. As mentioned before, containers are launched from images and can contain one or more running processes. One can think about images as the building or packing aspect of Docker and the containers as the running or execution aspect of Docker.

A Docker container is:

- An image format.
- A set of standard operations.
- An execution environment.

Docker borrows the concept of the standard shipping container, used to transport goods globally, as a model for its containers. But instead of shipping goods, Docker containers ship software. Each container contains a software image – its "cargo" – and, like its physical counterpart, allows a set of operations to be performed. For example, it can be created, started, stopped, restarted, and destroyed. Like a shipping container, Docker doesn't care about the contents of the container when performing these actions; for example, whether a container is a web server, a database, or an application server. Each container is loaded the same as any other container.

Docker also doesn't care where containers are shipped from: they can be built on a laptop, upload to a registry, then download to a physical or virtual server, test, deploy to a cluster of a dozen Amazon EC2 hosts, and run. Like a normal shipping container, it is interchangeable, stackable, portable, and as generic as possible. With Docker, a user can quickly build an application server, a message bus, a utility appliance, a CI test bed for an application, or one of a thousand other possible applications, services, and tools. It can build local, self-contained test environments or replicate complex application stacks for production or development purposes. The possible use cases are endless.

2.2.3 Docker's technical components

Docker can be run on any x64 host running a modern Linux kernel. It has low overhead and can be used on servers, desktops, or laptops. It includes:

- A native Linux container format that Docker calls *libcontainer*.
- **Linux kernel namespaces**, which provide isolation for filesystems, processes, and networks.
- Filesystem isolation: each container is its own root filesystem.
- Process isolation: each container runs in its own process environment.
- Network isolation: separate virtual interfaces and IP addressing between containers.
- Resource isolation and grouping: resources like CPU and memory are allocated individually to each Docker container using the *cgroups*, or control groups, kernel feature.
- **Copy-on-write**: filesystems are created with copy-on-write, meaning they are layered and fast and require limited disk usage.
- Logging: STDOUT , STDERR and STDIN from the container are collected, logged, and available for analysis or trouble-shooting.
- Interactive shell: You can create a pseudo-tty and attach to STDIN to provide an interactive shell to your container.

2.3 Kubernetes

Kubernetes[2] is an open source automation framework for deploying, managing, and scaling applications. It is the essence of an internal Google project known as Borg, infused with the lessons learned from over a decade of experience managing applications with Borg (and other internal frameworks) at scale.

Kubernetes centers around a common API for deploying all types of software ranging from web applications, batch jobs, and databases. This common API is based on a declarative set of APIs and cluster configuration objects that allow you to express a desired state for your cluster. Rather than manually deploying applications to specific servers, you describe the number of application instances that must be running at a given time. Kubernetes will perform the necessary actions to enforce the desired state. For example, if you declare 5 instances of your web application must be running at all times, and one of the nodes running an instance of the web application fails, Kubernetes will automatically reschedule the application on to another node. In addition to application scheduling, Kubernetes helps automate application configuration in the form of service discovery and secrets. Kubernetes keeps a global view of the entire cluster, which means once applications are deployed Kubernetes has the ability to track them, even in the event they are rescheduled due to node failure. This service information is exposed to other apps through environment variables and DNS, making it easy for both cluster native and traditional applications to locate and communicate with other services running within the cluster. Kubernetes also provides a set of APIs that allows for custom deployment workflows such as rolling updates, canary deploys, and blue-green deployments.

2.3.1 Kubernetes Design Overview

Kubernetes aims to decouple applications from machines by leveraging the foundations of distributed computing and application containers. At a high level Kubernetes sits on top of a cluster of machines and provides an abstraction of a single machine. Some of the basic concepts of Kubernetes are listed below.

Clusters

Clusters are the set of compute, storage, and network resources where pods are deployed, managed, and scaled. Clusters are made of nodes connected via a "flat" network, in which each node and pods in nodes can communicate with each other. A typical Kubernetes cluster size ranges from 1 - 200 nodes,

[2] https://kubernetes.io/

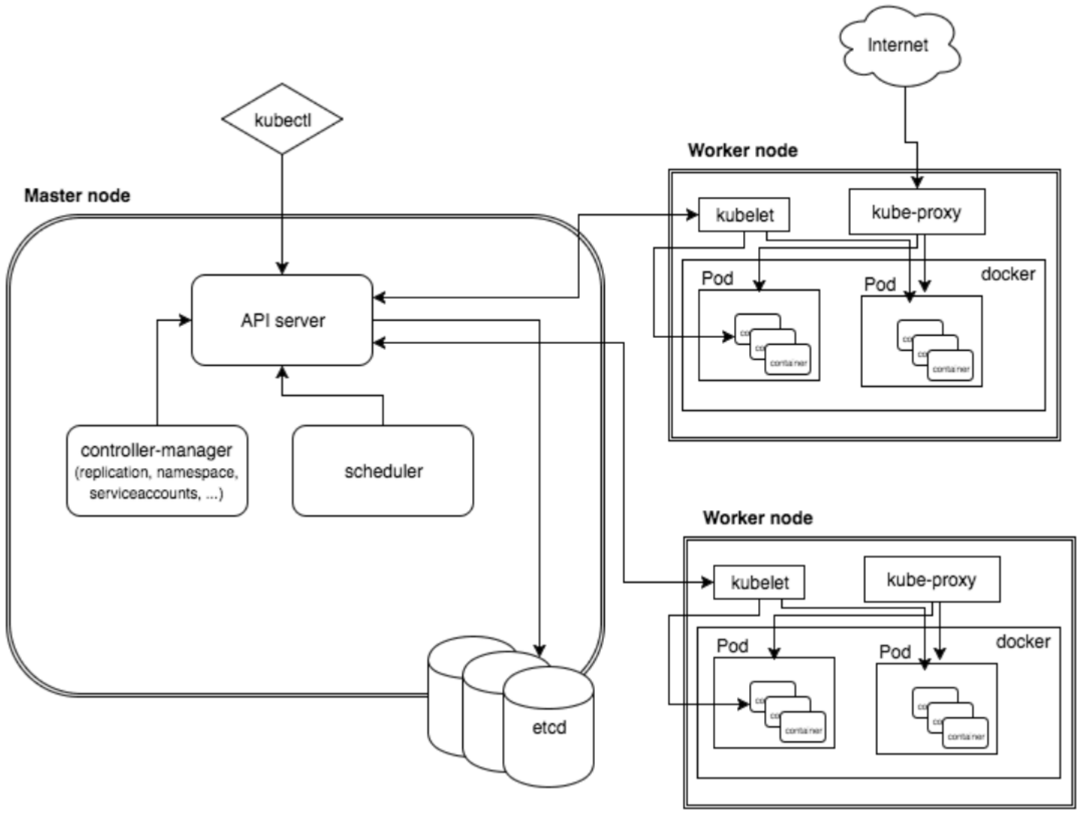


FIGURE 2.2: Kubernetes architecture.

and it's common to have more than one Kubernetes cluster in a given data center based on node count and service level agreements.

Pods

Pods are a co-located group of application containers that share volumes and a networking stack. Pods are the smallest units that can be deployed within a Kubernetes cluster. They can be deployed individually but long running applications, such as web services, should be deployed and managed by a replication controller.

Replication Controllers

Replication Controllers ensure a specific number of pods, based on a template, running at any given time. Replication Controllers manage pods based on labels and status updates.

Services

Services deliver cluster wide service discovery and basic load balancing by providing a persistent name, address, or port for pods with a common set of labels.

Labels

Labels are key/value pairs that are attached to objects, such as pods. Labels are intended to be used to specify identifying attributes of objects that are meaningful and relevant to users, but do not directly imply semantics to the core system. Labels can be used to organize and to select subsets of objects. Labels can be attached to objects at creation time and subsequently added and modified at any time. Each object can have a set of key/value labels defined. Each Key must be unique for a given object

2.3.2 The Kubernetes Control Plane - Master Node

The control plane is made up of a collection of components that work together to provide a unified view of the cluster.

etcd

etcd is a distributed, consistent key-value store for shared configuration and service discovery, with a focus on being simple, secure, fast, and reliable. etcd uses the Raft consensus algorithm[3] to achieve fault-tolerance and high-availability. etcd provides the ability to "watch" for changes, which allows for fast coordination between Kubernetes components. All persistent cluster state is stored in etcd.

Kubernetes API Server

The API server is responsible for serving the Kubernetes API and proxying cluster components such as the Kubernetes web UI. The API server exposes a REST interface that processes operations such as creating pods and services, and updating the corresponding objects in etcd. The API server is the only Kubernetes component that talks directly to etcd.

Scheduler

The scheduler watches the API server for unscheduled pods and schedules them onto healthy nodes based on resource requirements.

[3]https://raft.github.io/

Controller Manager

There are other cluster-level functions such as managing service end-points, which is handled by the endpoints controller, and node lifecycle management which is handled by the node controller. When it comes to pods, replication controllers provide the ability to scale pods across a fleet of machines, and ensure the desired number of pods are always running. Each of these controllers currently live in a single process called the Controller Manager.

2.3.3 The Kubernetes Worker Node

The Kubernetes node runs all the components necessary for running application containers and load balancing service end-points. Nodes are also responsible for reporting resource utilization and status information to the API server.

Docker

Docker, the container runtime engine, runs on every node and handles downloading and running containers. Docker is controlled locally via its API by the Kubelet.

Kubelet

Each node runs the Kubelet, which is responsible for node registration, and management of pods. The Kubelet watches the Kubernetes API server for pods to create as scheduled by the Scheduler, and pods to delete based on cluster events. The Kubelet also handles reporting resource utilization, and health status information for a specific node and the pods it's running.

Proxy

Each node also runs a simple network proxy with support for TCP and UDP stream forwarding across a set of pods as defined in the Kubernetes API.

2.3.4 Kubernetes Resource Model

When a Pod is defined (i.e. a user defines a Pod), the amount of CPU and memory (RAM) each Container needs must be specified as well. When Containers have resource requests specified, the scheduler can make better decisions about which nodes to place Pods on.

Resource types

CPU and memory are each a resource type. A resource type has a base unit. CPU is specified in units of cores, and memory is specified in units of bytes. CPU and memory are collectively referred to as compute resources, or just resources. Compute resources are measurable quantities that can be requested, allocated, and consumed.

Each Container of a Pod can specify one or more of the following:

- spec.containers[].resources.limits.cpu
- spec.containers[].resources.limits.memory
- spec.containers[].resources.requests.cpu
- spec.containers[].resources.requests.memory

Although requests and limits can only be specified on individual Containers, it is convenient to talk about Pod resource requests and limits. A Pod resource request/limit for a particular resource type is the sum of the resource requests/limits of that type for each Container in the Pod.

Meaning of CPU

Limits and requests for CPU resources are measured in cpu units. One cpu, in Kubernetes, is equivalent to:

- 1 AWS vCPU
- 1 GCP Core
- 1 Azure vCore
- 1 Hyper-thread on a bare-metal processor

Meaning of Memory

Limits and requests for memory are measured in bytes. You can express memory as a plain integer or as a fixed-point integer using one of these suffixes: E, P, T, G, M, K. You can also use the power-of-two equivalents: Ei, Pi, Ti, Gi, Mi, Ki.

How Pods with resource requests are scheduled

When you create a Pod, the Kubernetes scheduler selects a node for the Pod to run on. Each node has a maximum capacity for each of the resource types: the amount of CPU and memory it can provide for Pods. The scheduler ensures that, for each resource type, the sum of the resource requests of the scheduled Containers is less than the capacity of the node. Note that although actual memory or CPU resource usage on nodes is very low, the scheduler still refuses to place a Pod on a node if the capacity check fails. This protects against a resource shortage on a node when resource usage later increases, for example, during a daily peak in request rate.

How Pods with resource limits are run

When the kubelet starts a Container of a Pod, it passes the CPU and memory limits to the container runtime. If a Container exceeds its memory limit, it might be terminated. If it is restartable, the kubelet will restart it, as with any other type of runtime failure. If a Container exceeds its memory request, it is likely that its Pod will be evicted whenever the node runs out of memory. A Container might or might not be allowed to exceed its CPU limit for extended periods of time. However, it will not be killed for excessive CPU usage.

2.4 NoSQL databases and Apache Cassandra

Over the last few years NoSQL databases have emerged as a cutting edge alternative to the traditional relational database systems. Prior to the NoSQL disruption, organizations used to rely on traditional database systems for all kind of workloads. A primary drawback of the relational databases is that they do not scale horizontally. The introduction of NoSQL databases changed this perception about database systems. Most of the NoSQL databases follow a distributed architecture, where the system is comprised of a number of servers. As a result of that, horizontal scaling can be achieved. However, NoSQL databases have their own limitations like no ACID compliance, lack of transactional features, etc. hence they can't be considered as a straight forward replacement for traditional relational database systems.

2.4.1 Apache Cassandra

Apache Cassandra^{TM}, a top level Apache project, is a distributed database for managing large amounts of structured data across many commodity servers, while providing a highly available service and no single point of failure. Also offers capabilities like continuous availability, linear performance scaling, operational simplicity and easy data distribution across multiple nodes. Its' architecture can be described as a master-less "ring" design that is elegant, easy to setup, and easy to maintain.

It is worth mentioning that in a Cassandra cluster all nodes play an identical role; there is no concept of a master node, with all nodes communicating with each other equally. Its' built-for-scale architecture means that it is capable of handling large amounts of data and thousands of concurrent users

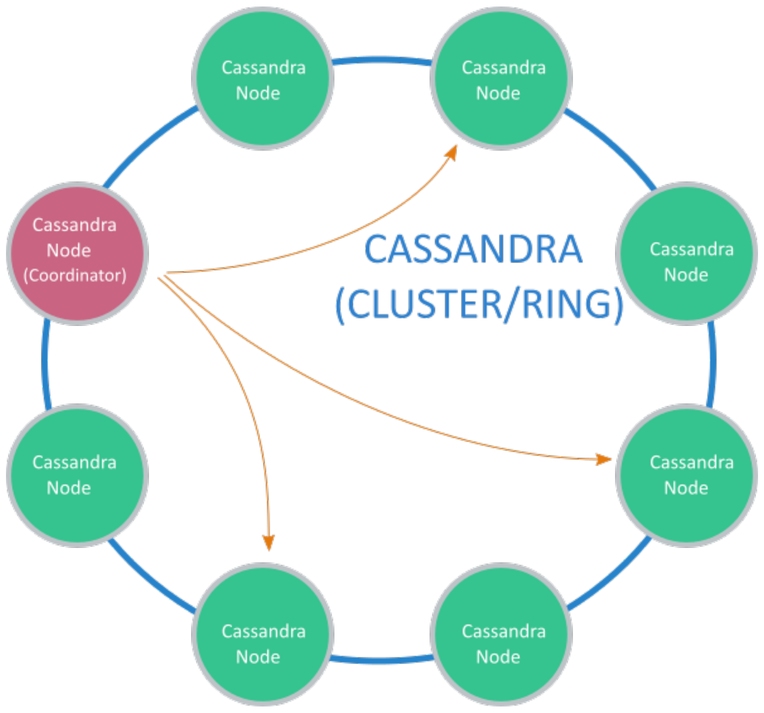


FIGURE 2.3: Cassandra Ring Architecture.

or operations per second as easily as it can manage much smaller amounts of data and user traffic. Lastly, a big key point of such architecture is that, unlike other master-slave or sharded systems, it has no single point of failure and therefore is capable of offering true continuous availability and uptime — simply by adding new nodes to an existing cluster without having to take it down.

Cassandra is a row-oriented database. Cassandra's architecture allows any authorized user to connect to any node in the cluster and access data using the CQL language. For ease of use, CQL uses a similar syntax to SQL. From the CQL perspective the database consists of tables.

Client read or write requests can be sent to any node in the cluster. When a client connects to a node with a request, that node serves as the coordinator for that particular client operation. The coordinator acts as a proxy between the client application and the nodes that own the data being requested. The coordinator determines which nodes in the ring should get the request based on how the cluster is configured. Figure 2.3 shows an example of such operation.

Chapter 3

Related Work

In this chapter, we present tools and platforms that are considered relevant to our work. For each of the systems we present their capabilities and discuss their functionality.

3.1 Azure Container Service[1]

Azure Container Service (AKS) is a container management service by Microsoft Azure that allows quick deployment of a production ready Kubernetes, DC/OS[2], or Docker Swarm[3] cluster. It runs on standard virtual machine infrastructure using a container optimized Linux image (a stripped-down version of Linux kernel with only a container run-time) as host operating system.

In order to deploy a Kubernetes cluster to AKS, a user must use Azure CLI (Command Line Interface), which is a shell specific for azure operations. First step is to create a resource group which is a template mechanism regarding resources (e.g. A resource group named *cluster node* consists of a virtual machine with 2 CPU cores and 4 GB of RAM). After that, a cluster can be created by specifying the number of machines needed as well as Kubernetes as orchestrator.

In terms of scaling, worker nodes can be added or removed from the cluster using Azure CLI. No autoscaling features are supported.

3.2 Amazon EC2 Container Service [4]

Amazon EC2 Container Service (Amazon ECS) is a container management service, provided by Amazon Web Services (AWS), that allows management

[1]https://azure.microsoft.com/en-us/services/container-service/

[2]https://dcos.io/

[3]https://docs.docker.com/engine/swarm/

[4]http://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html

of Docker containers on a cluster of Amazon Elastic Compute Cloud (Amazon EC2) instances (virtual machines). ECS also provides the ability to launch and stop containerized applications by making API calls, allows monitoring the state of the cluster from a centralized service and integrates with many familiar AWS features like Elastic Load Balancers, CloudTrail, CloudWatch etc.

The cluster setup is a single step process in which the number and flavor of EC2 instances needed for the cluster is specified. The rest of the setup process as well as the management of those instances is handled by the ECS service.

Comparing it to Kubernetes, both systems follow similar architecture and provide similar abstractions. For example a Kuberntes *pod* can be compared to an ECS *task definition* for application deployment or Kubernetes *kubelet* to ECS *container agent* for node management. One big difference is that Kubernetes is not restricted to run on any particular kind of infrastructure or a specific provider in contrast to ECS which is bound to AWS.

In terms of autoscaling features, ECS provides a mechanism which lets a user configure policies on how scaling operations have to take place. A policy consists of a set of rules and a set of actions. Rules typically refer to thresholds upon utilization metrics and actions to scaling operations. An example policy would be the following: *If cpu utilization in all cluster nodes is above x%, create a new node with y amount of compute resources and add it to the cluster*.

In ECS's case, autoscaling operations are triggered from the actual compute resources that the deployed applications consume. Based on that, ECS can move applications (tasks) across the cluster nodes in order to achieve better utilization, meaning that ECS's scheduler operates based on the actual runtime utilization of each application. Although Commodore collects utilization scores from the cluster nodes, it doesn't use them for its scaling operations. That's because Kubernetes scheduler operates based on the requested resources of each application (pod) (and not the actual utilized resources), thus scaling a Kubernetes environment based on actual utilization (and keeping the same scheduling mechanism) wouldn't produce any significant benefits.

3.3 Google Container Engine [5]

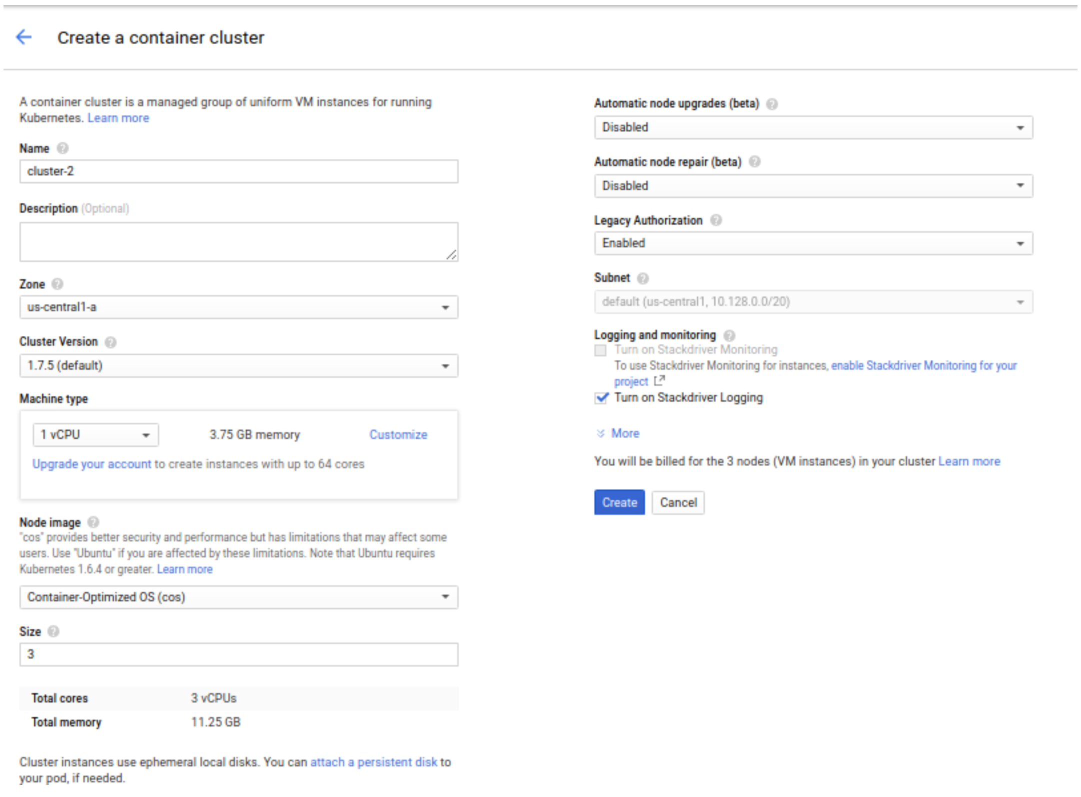
Google Container Engine (GKE) is a managed environment for deploying containerized applications. It is based on virtual machine infrastructure and uses the same compute and storage resources as GCE (Google Compute Engine). GKE uses a proprietary distribution of Kubernetes which is capable to integrate with other Google cloud services.

Figure 3.1 illustrates the cluster setup process, consisting of multiple steps in which one can specify:

[5]https://cloud.google.com/kubernetes-engine/

- the size of the cluster in terms of virtual machines
- the flavor and the deployment region of those machines
- the operating system image (Ubuntu or CoreOs) for the machines
- the Kubernetes version to be deployed

In terms of features, Google provides logging and monitoring tools (Stackdriver) which can be enabled during the setup process. It's also worth mentioning that the cluster comes pre-configured with a version of Kubernetes dashboard which gives GUI access to the cluster and also integrates with Googles' infrastructure in order to visualize live statistics regarding resource utilization (Figure 3.2).



Create a container cluster

A container cluster is a managed group of uniform VM instances for running Kubernetes. Learn more

Name: cluster-2

Description (Optional):

Zone: us-central1-a

Cluster Version: 1.7.5 (default)

Machine type: 1 vCPU — 3.75 GB memory — Customize

Upgrade your account to create instances with up to 64 cores

Node image

"cos" provides better security and performance but has limitations that may affect some users. Use "Ubuntu" if you are affected by these limitations. Note that Ubuntu requires Kubernetes 1.6.4 or greater. Learn more

Container-Optimized OS (cos)

Size: 3

Total cores	3 vCPUs
Total memory	11.25 GB

Cluster instances use ephemeral local disks. You can attach a persistent disk to your pod, if needed.

Automatic node upgrades (beta): Disabled

Automatic node repair (beta): Disabled

Legacy Authorization: Enabled

Subnet: default (us-central1, 10.128.0.0/20)

Logging and monitoring

☐ Turn on Stackdriver Monitoring

To use Stackdriver Monitoring for instances, enable Stackdriver Monitoring for your project

☑ Turn on Stackdriver Logging

You will be billed for the 3 nodes (VM instances) in your cluster Learn more

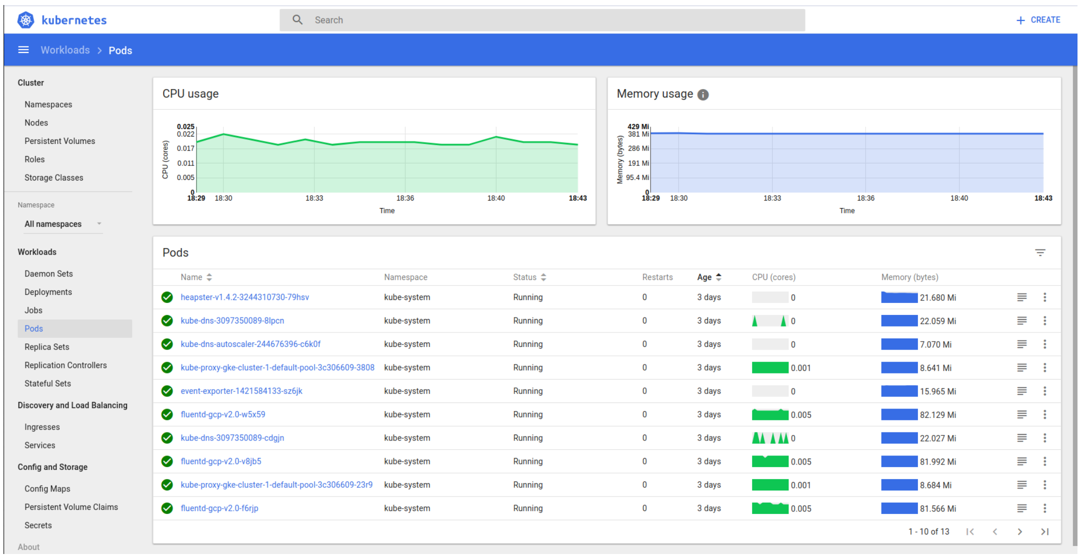
FIGURE 3.1: GKE Cluster Creation Wizard.

GKE Cluster Autoscaler [6]

Cluster Autoscaler (CA) is a mechanism, provided by GKE, which enables infrastructure scalability. CA is implemented as a Kubernetes plug-in and is part of the official Kubernetes project. At the time of this writing, CA is supported only on GKE platform.

CA automatically resizes a Kubenretes cluster based on the demands of the deployed pods. CA automatically adds new nodes to the cluster if new

[6]In order to understand terms like "node capacity" and "pod resources", please refer to subsection 2.3.4



Pods

Name	Namespace	Status	Restarts	Age	CPU (cores)	Memory (bytes)
heapster-v1.4.2-3244310730-79hsv	kube-system	Running	0	3 days	0	21.680 Mi
kube-dns-3097350089-8lpcn	kube-system	Running	0	3 days	0	22.059 Mi
kube-dns-autoscaler-244676396-c6k0f	kube-system	Running	0	3 days	0	7.070 Mi
kube-proxy-gke-cluster-1-default-pool-3c306609-3808	kube-system	Running	0	3 days	0.001	8.641 Mi
event-exporter-1421584133-sz6jk	kube-system	Running	0	3 days	0	15.965 Mi
fluentd-gcp-v2.0-w5x59	kube-system	Running	0	3 days	0.005	82.129 Mi
kube-dns-3097350089-cdgjn	kube-system	Running	0	3 days	0	22.027 Mi
fluentd-gcp-v2.0-v8jb5	kube-system	Running	0	3 days	0.005	81.992 Mi
kube-proxy-gke-cluster-1-default-pool-3c306609-23r9	kube-system	Running	0	3 days	0.001	8.684 Mi
fluentd-gcp-v2.0-f6rjp	kube-system	Running	0	3 days	0.005	81.566 Mi

FIGURE 3.2: GKE Cluster Dashboard.

pods are created and there are not enough compute resources available; conversely, if a node in the cluster is underutilized and its pods can run on other nodes, CA moves the pods and deletes the node. Unlike ECS's configurable approach (discussed in the previous section), CA scales the cluster automatically. CA's functionality is based on the amount of resources that pods request, not the amount they actively using. Essentially, CA schedules pods on nodes based on the assumption that their resource requests are accurate.

Both CA and Commodore rely on pod resource requests in order to perform scaling operations. While Commodore works in conjunction with Kubernetes scheduler (i.e. Commodore watches for failed scheduling events from Kubernetes), CA may perform additional operations inside the cluster (e.g. move pods from one node to another in order to achieve better resource utilization). Commodore respects Kubernetes scheduler decisions and just provides the resources needed in order to schedule new pods. Regarding scale down operations, CA tries to free underutilized nodes (i.e. nodes with a small number of pods) by moving their pods to other cluster nodes in order to eventually delete them. Commodore doesn't move any pods and just deletes empty nodes. One last key difference between those systems is that Commodore can operate on any infrastructure unlike CA which is supported only on GKE.

Chapter 4

Architecture

This chapter describes the architecture of Commodore system, identifies its basic components and provides an in depth analysis of its functionality.

4.1 Functional Specification

Commodore adds infrastructure scalability features to Kubernetes, meaning addition or removal of compute resources (bound in the context of virtual machines) depending on cluster state. As Figure 4.1 illustrates, Commodore is implemented as a service that runs in parallel and in cooperation with the master node of Figure 2.2 (alternatively it can be implemented within the master node), it reads the cluster state periodically and performs operations accordingly. More specifically:

- Commodore makes periodic checks to the cluster state, by requesting the latest events emitted by the scheduler, from the Kubernetes API server.
- Commodore takes action when there are failed scheduling events, meaning events emitted by the scheduler when it fails to deploy one or more pods to the cluster. Failed scheduling events are emitted when pods request more compute resources than the cluster can provide.

In the advent of failed scheduling events, a scale up operation is mandatory. Scale up operations work as follows:

- Commodore calculates a sum of the compute resources requested from the pods that failed to be scheduled[1].
- Commodore determines which (virtual machine) flavor suits better for the requested resources.
- Commodore requests from the cloud provider a new virtual machine with that flavor that matches the requested resource demands.
- Commodore binds the new machine to the cluster and the previously failing pods are deployed to this new machine.

[1]**Please refer to subsection 2.3.4 for more details**.

Otherwise, if there are no failed scheduling events, Commodore checks if a scale down operation has to take place:

- Commodore inquires Kubernetes API server regarding what pods are deployed on each node.
- If there are nodes with no pods deployed (empty nodes) Commodore removes them from the cluster.

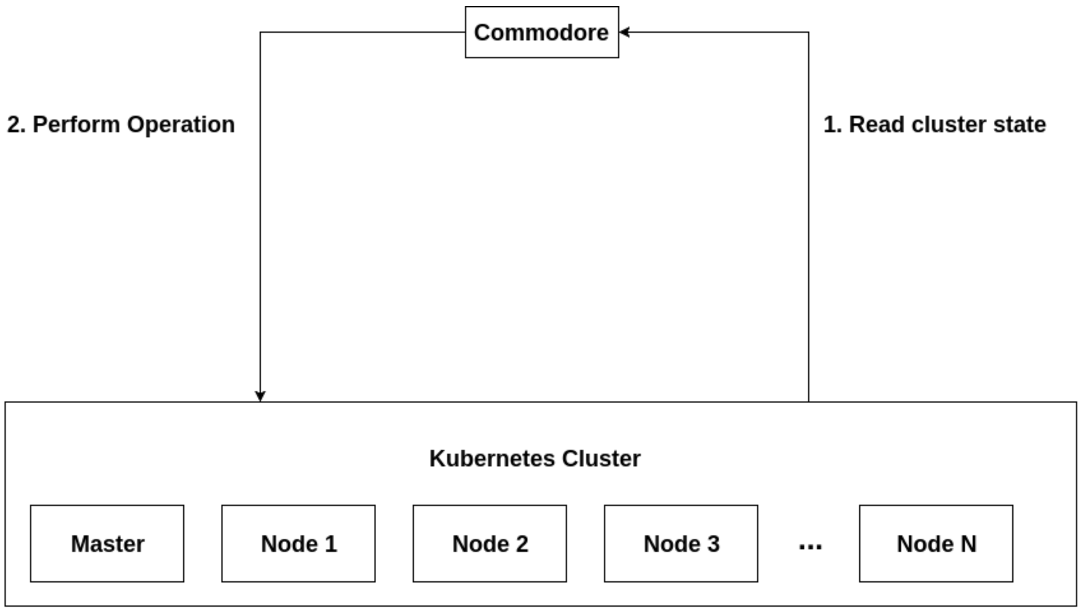


FIGURE 4.1: Commodore abstract architecture.

4.2 Commodore Architecture

Commodore follows the Service Oriented Architecture principles. Decomposing an application into services improves modularity and makes the application easier to understand, develop and test. Commodore comprises of many interconnected components (or services) as shown in Figure 4.2, the most important of them being:

- The Database.
- The Collector Service.
- The Infrastructure Manager.
- The Kubernetes Manager.
- The Rule Engine.
- The User Interface.

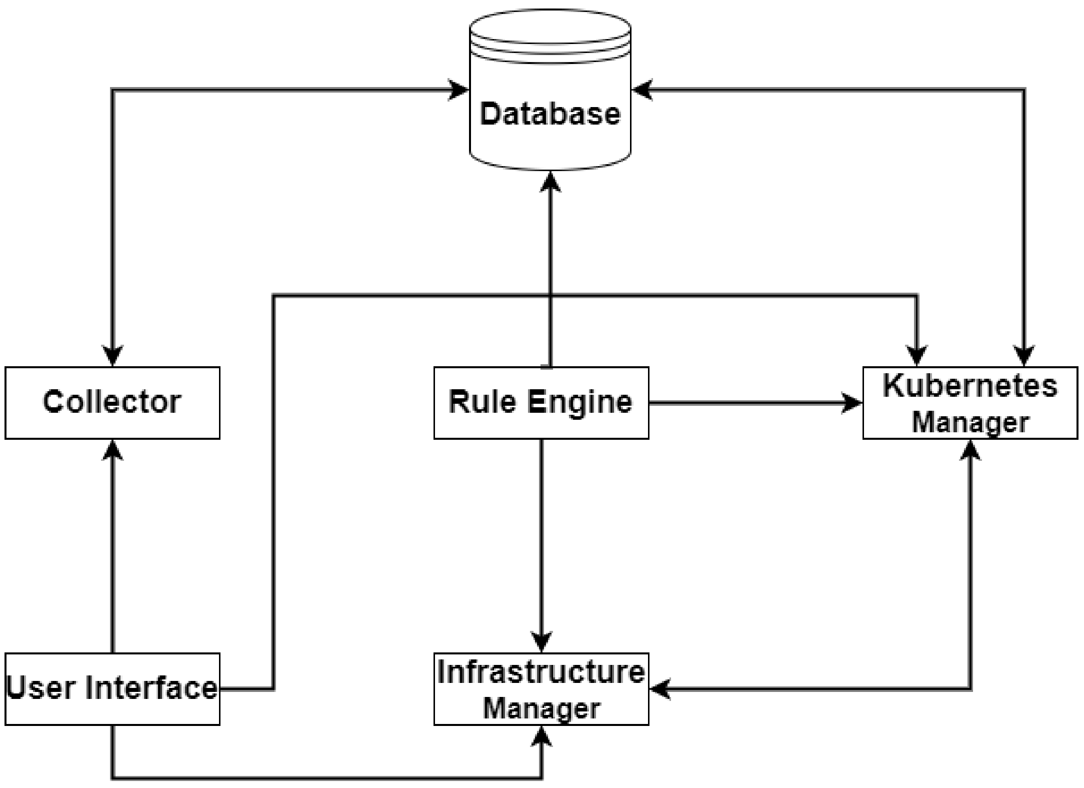


FIGURE 4.2: Commodore architecture.

4.3 Database

Commodore produces large amounts of usage data (logs) that account for utilization scores from the cluster, events from the Kubernetes API server as well as history data regarding scaling operations. All this data is stored in an Apache Cassandra (NoSQL) database which suits best the nature of semi-structured log data and of operations allowed on such data (i.e. ACID transactions are not mandatory), and also due to its scalability, SQL like features and superior performance on insert and retrieval operations.

4.4 Collector Service

Collector is the service that collects utilization scores, stores them in the database and exposes and API for accessing the database. This module consists of two components, a daemon process and a Web server. The daemon is a simple script that runs on each cluster node and at regular time intervals (i.e. every second) collects utilization scores from the node (including CPU, disk and memory usage) and reports them to the server.

The collector is implemented as a web application that exposes a set of Restful services that can be invoked in order to submit and retrieve utilization metrics. Apart from CRUD operations, the service also performs a moving

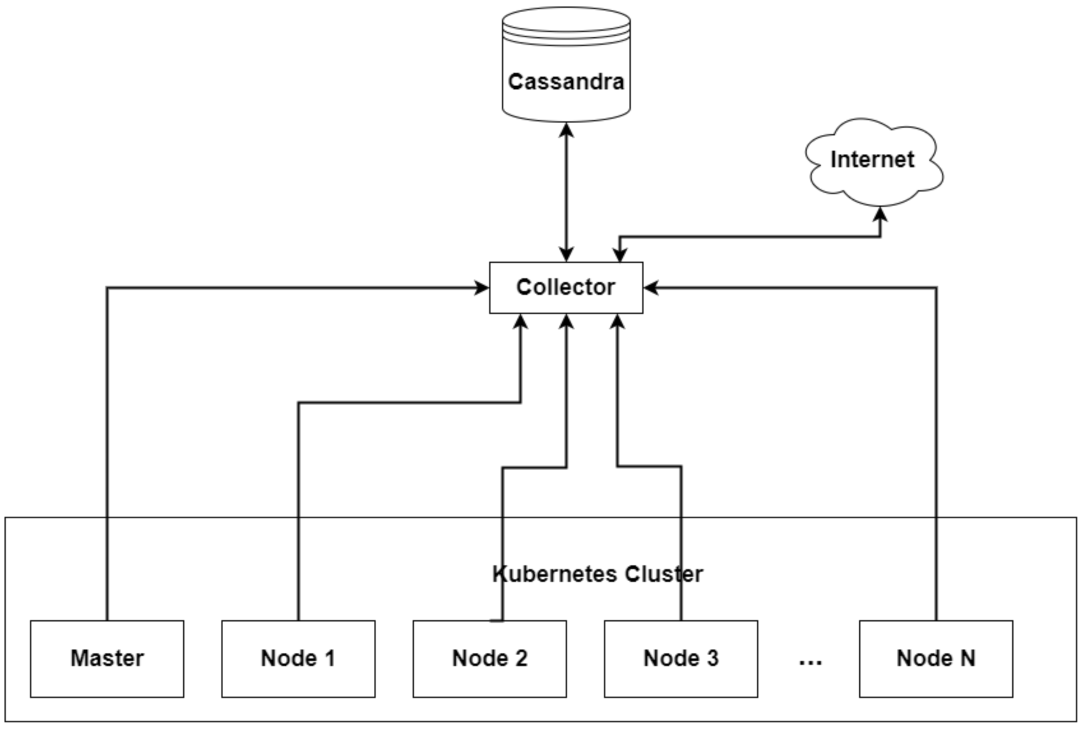


FIGURE 4.3: Collector Architecture.

average aggregation on the incoming data, which calculates the average utilization of each node per minute. This aggregated score is useful because it gives a more accurate representation of the actual utilization than a single metric does, since node utilization fluctuates greatly between two consecutive metrics.

Collector's scores are only used from the monitoring interface for visualization purposes. This is a useful feature for cluster administrators because it gives them the ability to monitor the actual cluster load without the need of extra tools. Note that collector's scores are not used for autoscaling since Kubernetes scheduler does not rely on actual utilization in order to distribute resources, as explained in subsection 2.3.4. Nevertheless, those scores may potentially power a more advanced scheduling mechanism for Kubernetes (using machine learning or other mathematical models) and in conjunction with existing Commodore features, could achieve better overall cluster utilization.

4.5 Infrastructure Manager

Infrastructure manager is the service that manages cluster's infrastructure and allows CRUD operations to be executed on infrastructure level, like addition, resize or deletion of cluster nodes (virtual machines). The infrastructure manager communicates with the cloud provider (in our case OpenStack through its API) and also exposes a set of Restful web services which can be invoked from the other Commodore services, as Figure 4.4 illustrates.

Infrastructure manager implements the following operations:

1. Listing available flavors
2. Listing nodes
3. Getting details of a specific node
4. Deleting nodes
5. Adding nodes
6. Resizing nodes

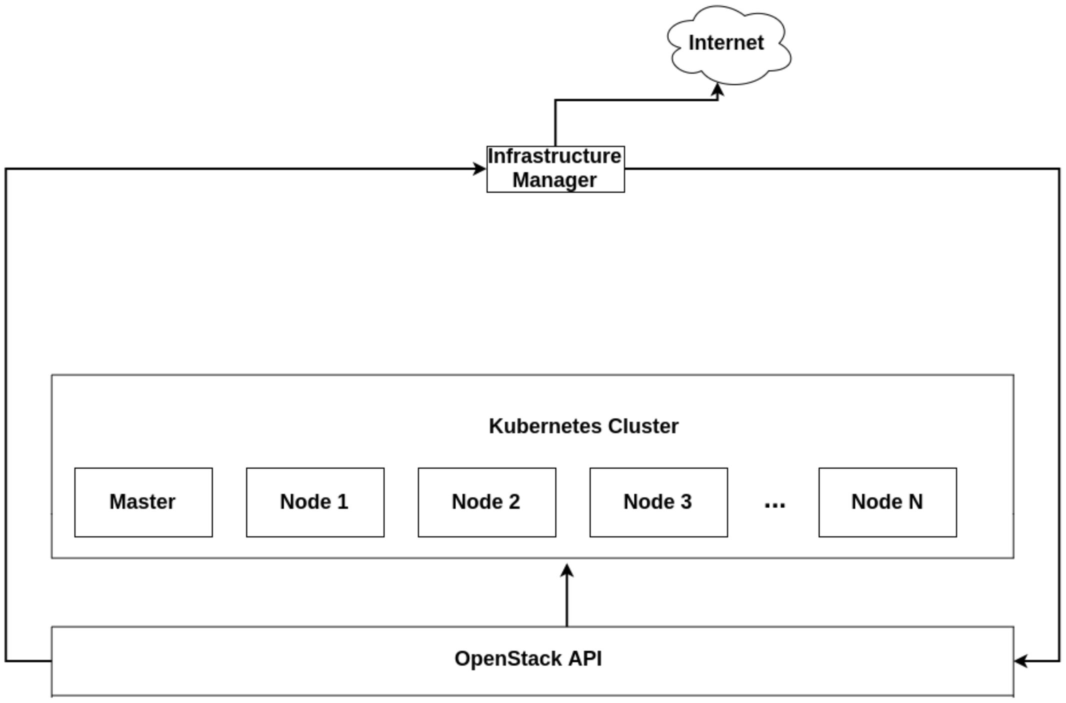


FIGURE 4.4: Infrastructure Manager Architecture.

4.6 Kubernetes Manager

Kubernetes manager is the service responsible for interacting with the Kubernetes cluster. It exposes a Restful API which allows a specific set of operations to be executed on the Kubernetes cluster, like node operations, listing pods and events. As mentioned in section 2.3, Kubernetes has a built-in API server which controls operations upon the cluster and is used by Kubernetes Manager in order to achieve its functionality. Kubernetes Manager is also responsible for logging what events have been emitted from the Kubernetes cluster as well as the pods that are deployed on each node.

Kubernetes manager implements the following operations:

1. Listing pods
2. Listing events
3. Listing nodes
4. Getting details of a specific node
5. Removing nodes from the cluster

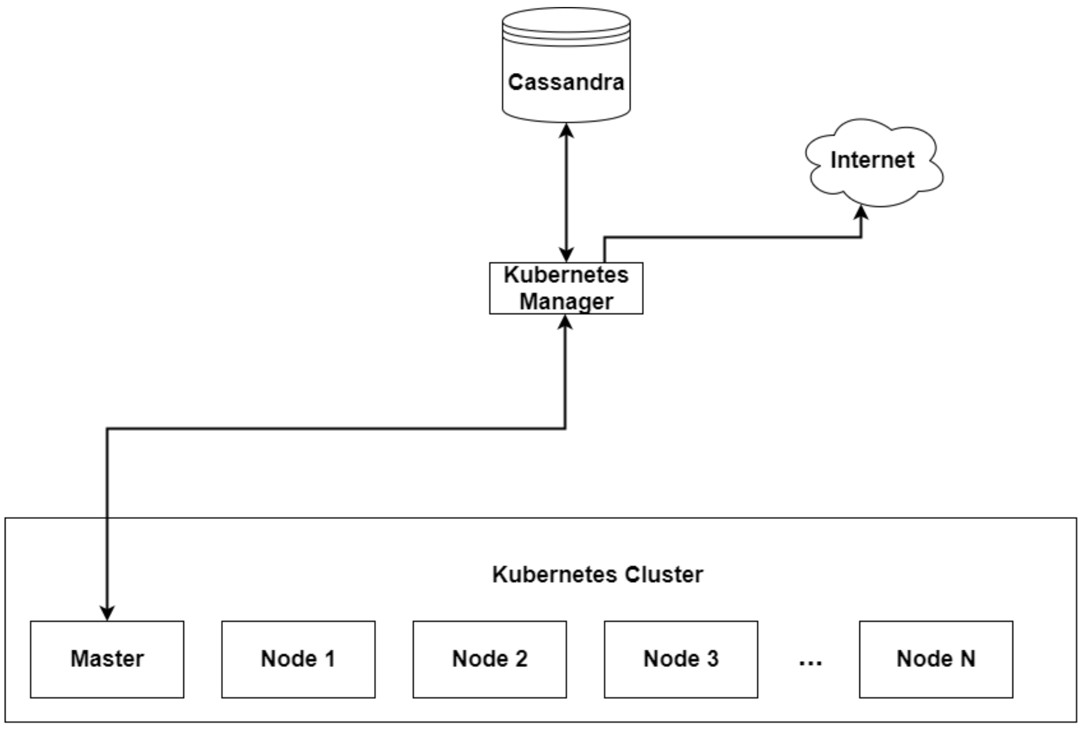


FIGURE 4.5: Kubernetes Manager Architecture.

4.7 Rule Engine

The Rule engine implements the infrastructure scalability features. It is responsible for managing the number and the flavor of the worker nodes (virtual machines) that Kubernetes utilizes. It receives information about the cluster state from Kubernetes Manager and decides whether or not an infrastructure change (scale up or down) is needed. If a change is mandatory, Rule engine initiates a sequence of operations for the purpose of adjusting the cluster (addition or deletion of machines). The operation is triggered periodically by a timeout value which is set at run-time.

Algorithm 1 illustrates the method that implements the basic logic behind the whole service. Initially, the method checks if a scale up operation is needed by calling the *scaleUp()* method (Algorithm 2). This method returns a *decision* and a *flavor*. If the *decision* is 'up', means that Kubernetes scheduler has failed to deploy an indefinite number of pods to the existing cluster nodes (that information is retrieved from Kubernetes Manager). *scaleUp()* function also

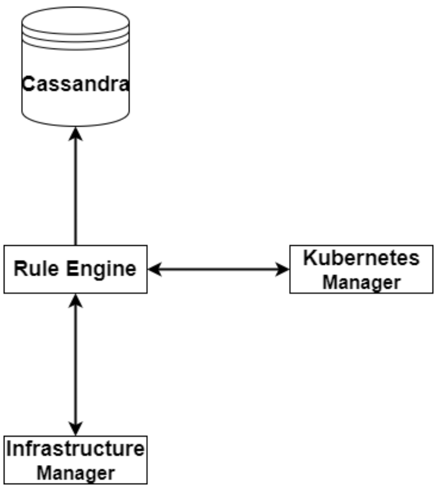


FIGURE 4.6: Rule Engine Architecture.

Algorithm 1 Timeout Loop - Rule Engine.

```
 1: procedure DECIDE
 2:     $decision, flavor \leftarrow scaleUp()$
 3:     if $decision =$ 'up' then
 4:         $infrastructureManager.addNode(flavor)$
 5:         $saveDecision(decision, flavor)$
 6:         $sleep(longTime)$
 7:     else
 8:         $decision, nodeList \leftarrow scaleDown()$
 9:         if $decision =$ 'down' then
10:             for $node$ in $nodeList$ do
11:                 $infrastructureManager.deleteNode(node)$
12:         $sleep(shortTime)$
```

calculates how much resources are needed in order to successfully deploy the above mentioned pods, by summing the requested compute resources of each pod. Based on this score, the best fit of the available flavors is returned. Then, a new node with the specific flavor is created by invoking the *addNode* function of infrastructure manager. Finally the thread sleeps. The amount of time that the thread is sleeping is relevant to the time the cloud provider needs to start a new virtual machine and initialize it.

If scale up operations are not needed, the method checks if scale down can be achieved by calling the *scaleDown()* method (Algorithm 3). This method returns a *decision* and a *nodeList*. If the *decision* is 'down', means that there are nodes in the cluster with no pods deployed to them. Those nodes are added in the *nodeList* variable. Then, iterating through the *nodeList* we invoke the *deleteNode* function of infrastructure manager for each node in order to remove them from the cluster. Finally, regardless if a scale down can be achieved or not, the thread goes to sleep. The amount of time that the thread is sleeping is one minute, since node deletion is almost instant and there is no need for the whole process to run more frequently than once in a minute.

Algorithm 2 Scale Up - Rule Engine.

```
1: procedure SCALEUP
2:     $decision \leftarrow$ 'neutral'
3:     $events \leftarrow kubernetesManager.getFailedSchedulingEvents()$
4:     if $events$ is empty then return $decision, null$
5:     $pods \leftarrow kubernetesManager.getPods(events)$
6:     if $pods$ is empty then return $decision, null$
7:     $defaultCpu \leftarrow 0.1$
8:     $defualtMemory \leftarrow 128$
9:     $reqCpu \leftarrow 0.02$
10:    $reqMemory \leftarrow 0.0$
11:    for $pod$ in $pods$ do
12:        $containers \leftarrow pod.getContainers()$
13:        for $container$ in $containers$ do
14:            if $container$ requires cpu then
15:                $reqCpu \leftarrow reqCpu + container.getCpu()$
16:            else
17:                $reqCpu \leftarrow reqCpu + defaultCpu$
18:            if $container$ requires memory then
19:                $reqMemory \leftarrow reqMemory + container.getMemory()$
20:            else
21:                $reqMemory \leftarrow reqMemory + defualtMemory$
22:    $flavors \leftarrow infrastructureManager.getFlavors()$
23:    $flavor \leftarrow findBestFit(flavors, reqCpu, reqMemory)$
24:    if $flavor$ is empty then return $decision, null$
25:    $decision \leftarrow$ 'up' return $decision, flavor$
```

Algorithm 3 Scale Down - Rule Engine.

```
1: procedure SCALEDOWN
2:     $decision \leftarrow$ 'neutral'
3:     $nodes \leftarrow infrastructureManager.getNodes()$
4:     for $node$ in $nodes$ do
5:         if $node.canbeDeleted()$ then
6:             if $kubernetesManager.isNodeRegistered(node)$ then
7:                 $pods \leftarrow kubernetesManager.getPods(node)$
8:                 if $pods$ is empty then
9:                     $nodesToDelete.add(node)$
10:    if $nodesToDelete$ is empty then return $decision, null$
11:    $decision \leftarrow$ 'down' return $decision, nodesToDelete$
```

4.8 User Interface

The last service in Commodore's architecture is the monitoring user interface, which is responsible for the visualization of various aspects of our system including:

- The health status of Commodore's services.
- The number, the flavor and the state of the various nodes that Kubernetes cluster utilizes.
- The actual utilization of each cluster node.
- The pods that are deployed on each node.

This is an optional component to our architecture since it doesn't serve any functional purpose than monitoring. The reason it was included is the ease of use that it provides in terms of monitoring multiple aspects of the system from one place.

In order to get the statistics and metrics needed, the monitoring interface invokes various web services that the rest of the components expose, as Figure 4.7 illustrates.

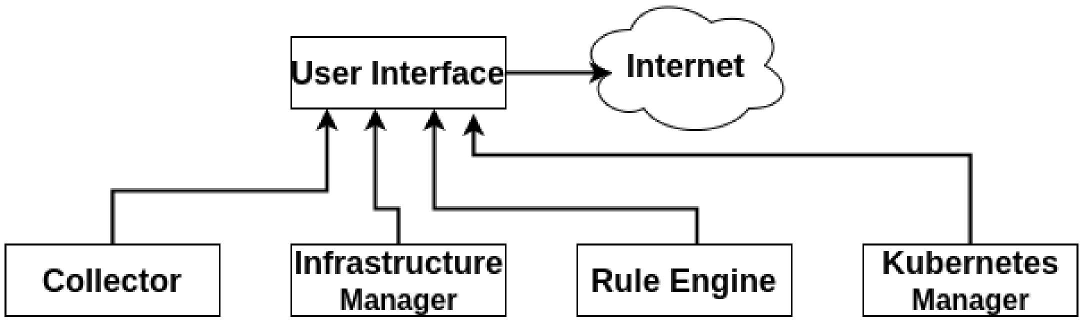


FIGURE 4.7: User interface Architecture.

Chapter 5

Implementation

This chapter describes the implementation details of Commodore components and provides details about the technical issues we stumbled upon and how we dealt with them.

5.1 General Implementation

As mentioned in chapter 4, Commodore is a system that follows the service oriented architecture which means that each key component of the system is implemented individually. Although, we could use different set of tools or languages for each one, we chose to standardize the environment for all of Commodore's services. Also, since Commodore doesn't rely on any special libraries in order to operate or any low-level access to the machine that is deployed, we decided to use well established technologies and libraries for the implementation. Our choice of tools was the Java programming language along with Spring Boot framework. We chose Spring Boot because it provides an easy way to create Restful web services and also packages a Java servlet container (in our case Apache Tomcat) inside the same executable. This makes the deployment of the individual services as easy as running a single command in the shell.

Architecturally, all of Commodore's services share the same design principals. As Figure 5.1 illustrates, each service is implemented following the multi-layer paradigm, which separates the application functionality into layers. Each layer is responsible for a specific set of operations. More specifically:

- **Data Layer:** is responsible for database operations. It consists of two packages, Repositories and Model. Repositories contain the data access objects which handle database connections, query execution etc. Model contains the domain objects of the service.
- **Logic Layer:** encapsulates the logic and functionality of the service.
- **Web Layer:** is the entry point of the service and contains the Controllers, which define the Rest endpoints as well as the Filters, which act as middleware upon those endpoints.

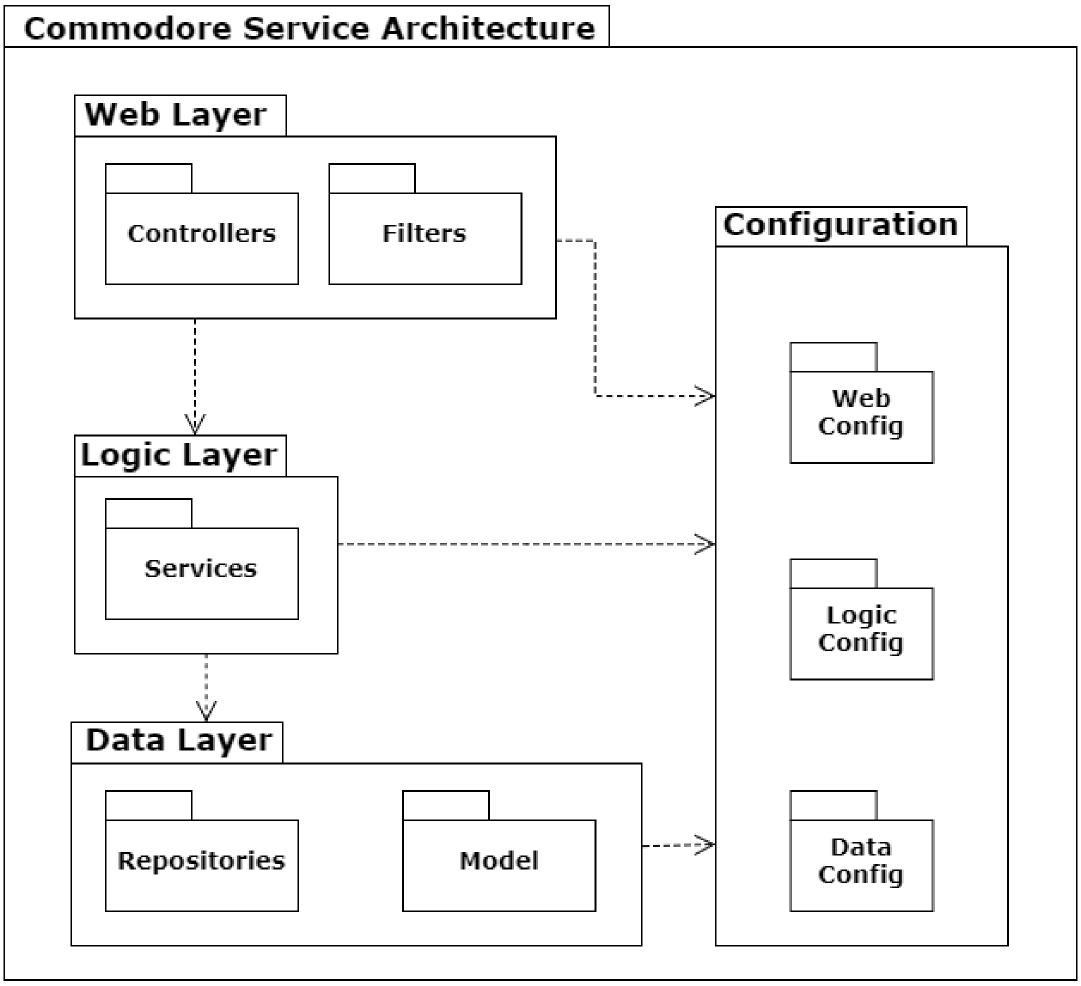


FIGURE 5.1: Commodore service internal design.

- **Configuration:** is a cross-cutting package which is used by all the above mentioned layers and contains all the necessary configuration data for each layer individually.

5.2 Collector

As mentioned in chapter 4, collector service consists of two components, the daemon process and the web server. The daemon is a simple python script that runs on each of the cluster nodes and sends utilization data, in JSON format, to the server. This script is managed by systemd and starts on machine boot. Each second, it sends a request to the web server containing the current utilization metrics. An example of such message is shown below:

LISTING 5.1: Collector - Daemon request data.

```
{
  "host": "192.168.11.12",
  "timestamp": "2017-03-06 00:10:12+0000",
  "cpu": 11.586,
  "disk": {
    "free": 183.96403,
    "total": 210.93545,
    "used": 16.19002
  },
  "memory": {
    "available": 4.35578,
    "free": 1.38603,
    "percent": 72.1,
    "total": 15.61531,
    "used": 10.84916
  }
}
```

Daemons' functionality was implemented using the python programming language because it is generally adopted by almost all Linux distributions, it has a lightweight run-time environment with a small memory footprint and is pre-installed to many distributions out of the box. The server side of the collector service is implemented as a web application (as described in section 5.1) and exposes a set of Restful services which can be invoked in order to submit and retrieve utilization data.

Figure 5.2 describes the interactions between the objects in the order they occur, when a host sends a metric request. Figure 5.3 describes the case when metric summaries are requested to be retrieved.

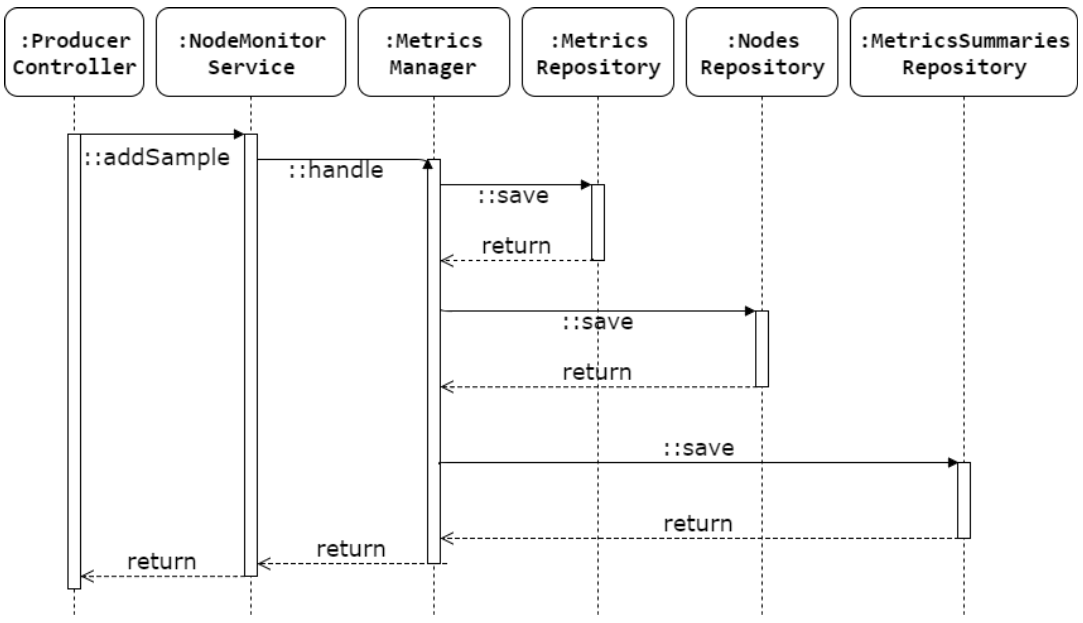


FIGURE 5.2: UML sequence diagram (daemon request) - Collector.

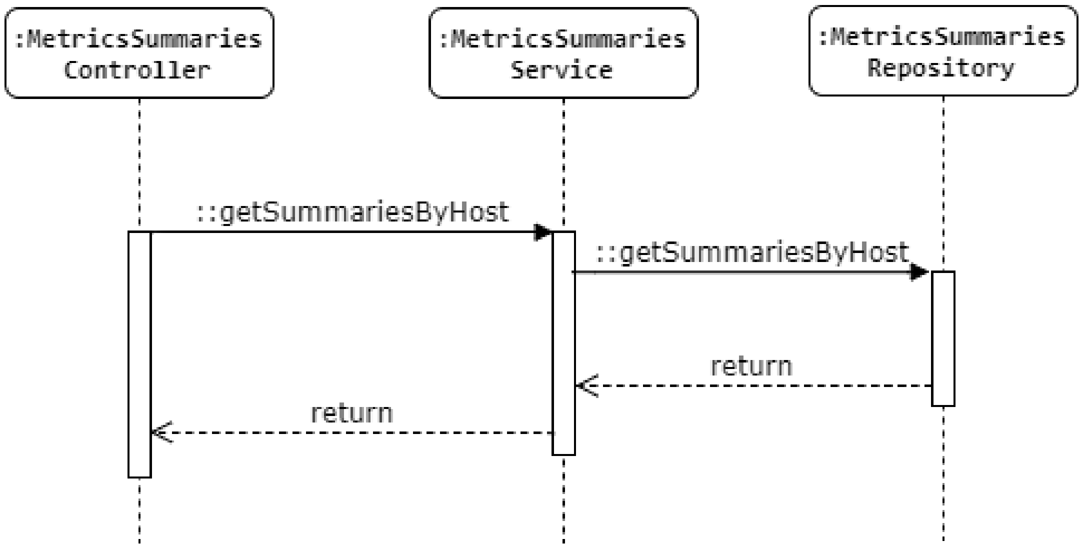

:MetricsSummaries Controller
:MetricsSummaries Service
:MetricsSummaries Repository
::getSummariesByHost
::getSummariesByHost
return
return

FIGURE 5.3: UML sequence diagram (fetch request) - Collector.

5.3 Infrastructure Manager

Infrastructure manager is the service which handles all the infrastructure level operations like listing, adding or deleting nodes. It follows the same internal architecture as the rest of Commodore's services and exposes an Restful API in order to make its functionality available to the other services. Besides OpenStack integration, our goal for this service was to make it generic enough so that it can be integrated easily with other cloud platforms like Google Cloud or AWS. For that matter, we chose to use Apache jclouds®, an open source multi-cloud toolkit for the Java platform that gives the ability to create services that are portable across clouds while allowing full control over cloud-specific features.

In terms of functionality, listing and getting details about nodes and flavors are operations that involve only the cloud provider so the implementation was fairly trivial. An example of the API request and response is shown below:

LISTING 5.2: GET /nodes

```
[
  {
    "id": "ca07da69-39c4-4151-8745-00d8f89f6c38",
    "name": "node-f488963a-de7f-41d6-b193-f37c9b47f779",
    "flavor": "m1.large",
    "status": "ACTIVE",
    "created": 1499873693000,
    "updated": 1499873701000,
    "address": "192.168.111.143",
    "canBeDeleted": true
  },
  {
    "id": "dbb14c2d-2494-4503-a11b-bf4a28fc6c25",
    "name": "kube-adm-gateway",
    "flavor": "m1.small",
    "status": "ACTIVE",
    "created": 1493599333000,
    "updated": 1495410802000,
    "address": "192.168.111.12",
    "canBeDeleted": false
  },
  {
    "id": "9300ffef-2599-4184-9524-65d829f7b6f8",
    "name": "kube-adm-master",
    "flavor": "m1.small",
    "status": "ACTIVE",
    "created": 1493598226000,
    "updated": 1495410802000,
    "address": "192.168.111.11",
    "canBeDeleted": false
  }
]
```

On the other hand, adding a node also involves Kubernetes because that's the system that will eventually utilize the new node. We also had to take into consideration the fact that adding a node is an asynchronous operation for the cloud provider. So the problem we faced, was that we couldn't know when a new node was ready to be binded to the rest of the cluster. Luckily, node initialization is a common problem that has been addressed in the past. One of the most adopted solutions is Cloud-init. Cloud-init is a set of scripts and utilities that handle early initialization of a cloud instance and is available in almost every modern cloud-enabled Linux distribution. Therefore, the solution was to create a script with a set of commands that will run after the initialization of the instance is finished. The initialization script is shown below:

LISTING 5.3: Node initialization script.

```
#!/bin/bash

hostnamectl set-hostname $(hostname -i|cut -f2 -d ' ')
systemctl restart network

sysctl -w net.bridge.bridge-nf-call-iptables=1
sysctl -w net.bridge.bridge-nf-call-ip6tables=1

kubeadm join --token <token> <kube-apiserver-endpoint>

systemctl enable monitor.service
systemctl start monitor.service
```

```
yum remove -y cloud-init

exit 0
```

The first lines of the script are changing the hostname to the internal IP address of the node in order to identify all the nodes with the same "key" in all of Commodore's modules. Then we restart the network manager in order for the changes to take affect immediately. Then we configure the container network interface bridge and finally we bind to the cluster and start the daemon of the Collector module. Since we don't need cloud-init package anymore, we uninstall it and then exiting.

As for node deletion, we had to first mark the node as unschedulable (cordon) which prevents new pods from being scheduled to that node. In order to do so, we had to invoke the Kubernetes Manager service. After this operation returns we invoke the Kubernetes Manager again to unbind the node from the cluster. Finally we delete the node from our infrastructure. It is worth mentioning that there is the capability to define a set of nodes as non deletable through configuration (e.g if there are more than one master nodes). Figure 5.4 presents the sequence diagram of the operation.

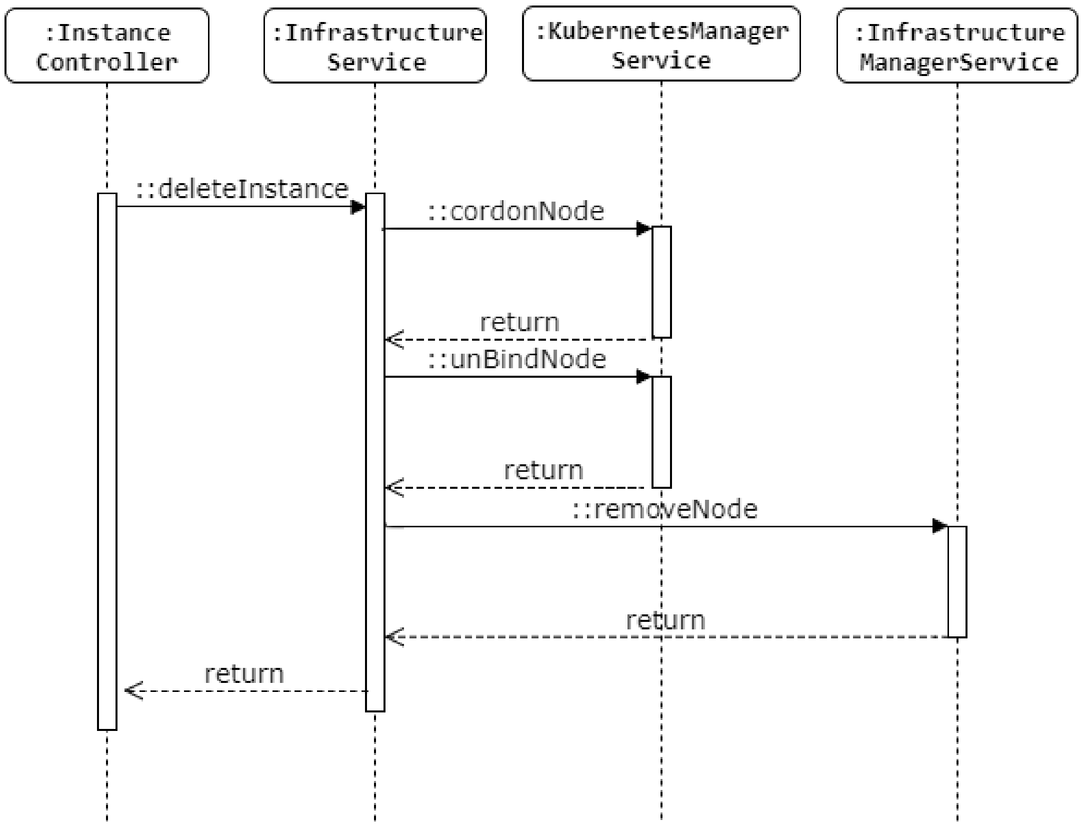


FIGURE 5.4: UML sequence diagram (delete node) - Infr. Manager.

5.4 Kubernetes Manager

Kubernetes manager integrates with Kubernetes API server in order to expose, through its API, certain operations to the rest of Commodore services including:

1. Listing nodes, pods and events
2. Getting details of a specific node
3. Removing nodes from the cluster

An example of listing the pods of a specific node is illustrated below (certain unrelated attributes are omitted from the response JSON object):

LISTING 5.4: GET /nodes/192.168.111.143/pods

```
[
  {
    "metadata": {
      "namespace": "default",
      "name": "primes-deployment-1315112784-0ww10",
      "labels": {
        "app": "primes"
      }
    },
    "kind": "Pod",
    "spec": {
      "containers": [
        {
          "image": "php-primes:v1.0",
          "name": "primes",
          "resources": {
            "requests": {
              "memory": {
                "amount": "768Mi"
              },
              "cpu": {
                "amount": "500m"
              }
            }
          },
          "ports": [
            {
              "protocol": "TCP",
              "containerPort": 80
            }
          ]
        }
      ]
    },
    "status": {
      "phase": "Running",
      "podIP": "10.32.64.3",
      "conditions": [
        {
          "lastTransitionTime": "2017-11-31T15:46:42Z",
          "type": "Ready",
          "status": "True"
        },
        ...
      ]
    }
  }
  ...
]
```

Also, Figure 5.5 shows the sequence diagram of the operation.

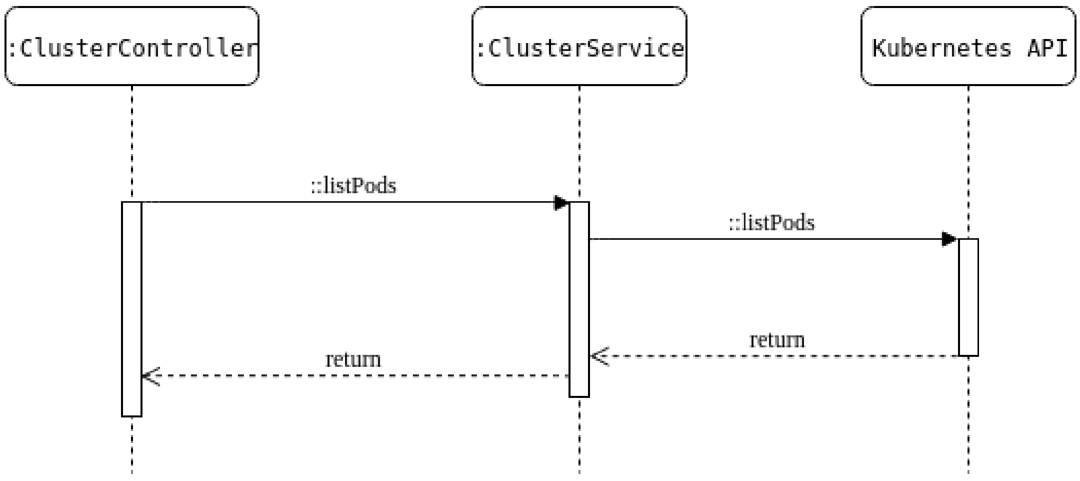


FIGURE 5.5: UML sequence diagram (get Pods) - Kubernetes Manager.

5.5 Rule Engine

Rule engine also follows the internal architecture described in section 5.1, with the only difference being the fact that it doesn't expose an API, meaning it is a typical Java console application. Nevertheless, it is the core service of Commodore since it handles the scaling operations.

As discussed in Algorithms 2 and 3, scale up and scale down operations utilize both Infrastructure and Kubernetes manager. The sequence diagrams for both operations are presented in Figures 5.6 and 5.7.

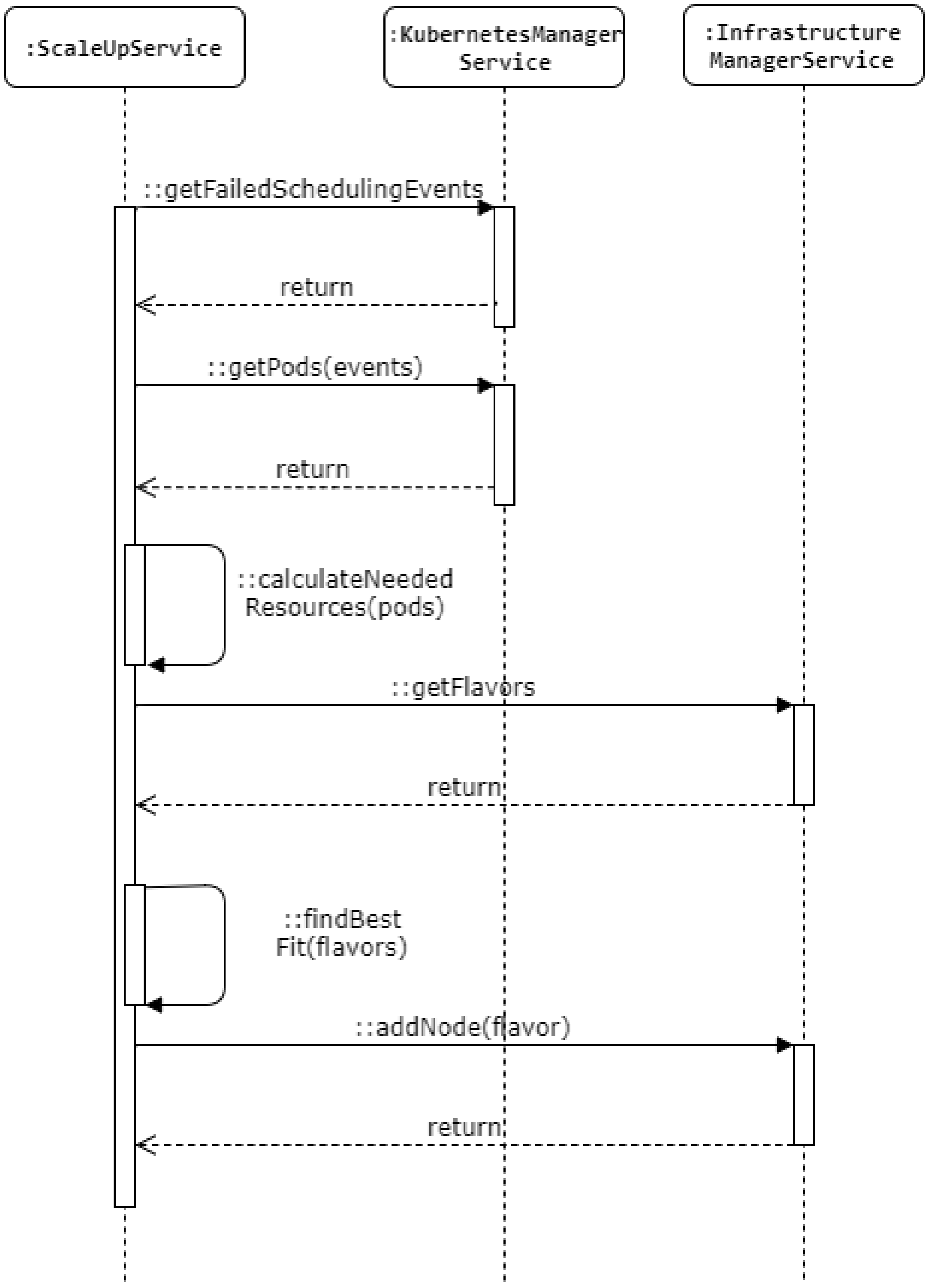

:ScaleUpService
:KubernetesManager Service
:Infrastructure ManagerService
::getFailedSchedulingEvents
return
::getPods(events)
return
::calculateNeeded Resources(pods)
::getFlavors
return
::findBest Fit(flavors)
::addNode(flavor)
return

FIGURE 5.6: UML sequence diagram (scale up) - Rule Engine.

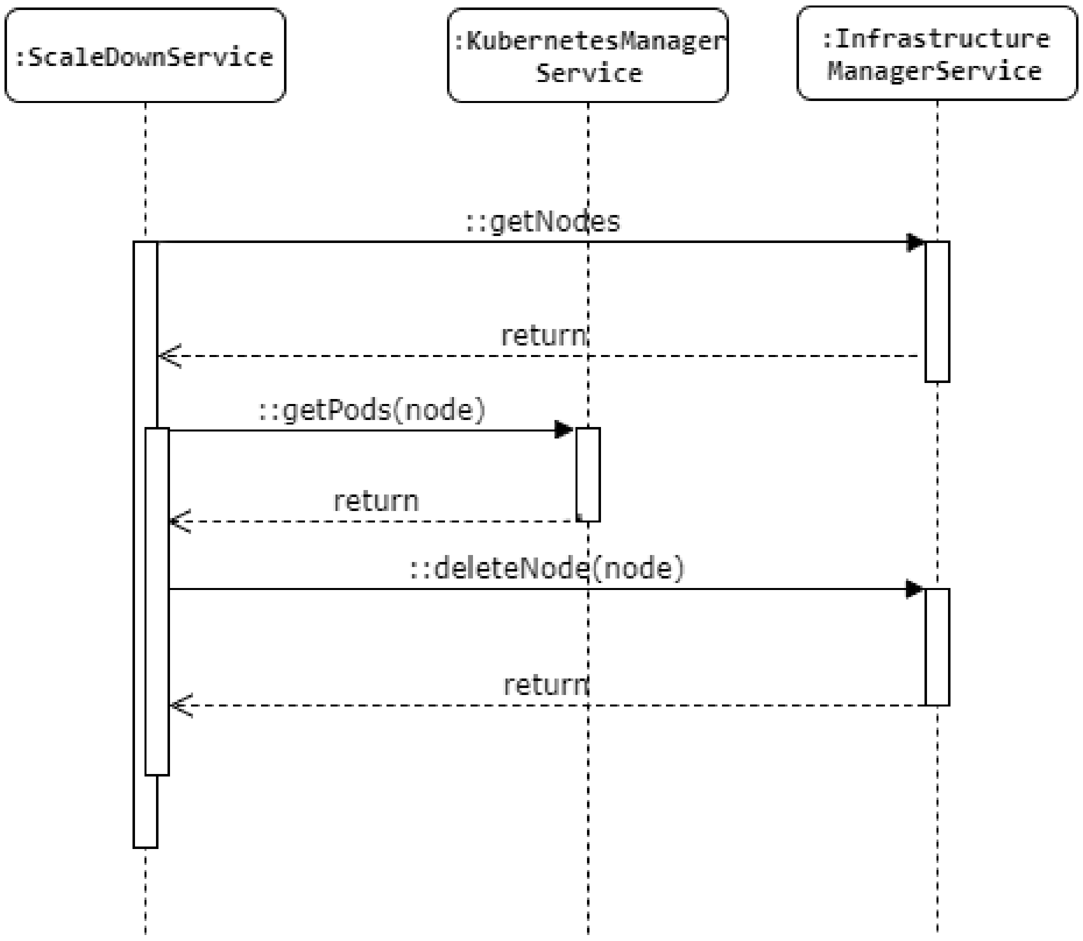


FIGURE 5.7: UML sequence diagram (scale down) - Rule Engine.

Chapter 6

Evaluation

This chapter describes the methodology we followed in order to evaluate Commodore and also explains the results from this analysis. Furthermore, it provides a detailed guide on how an application (or a service) can be deployed on a typical Kubernetes cluster.

6.1 Infrastructure and Deployment

Our first concern was to choose a cloud environment to run a Kubernetes cluster and Commodore services (deployed on Virtual Machines). We chose FIWARE[1], a cloud platform that runs on OpenStack. Our core infrastructure consists of three virtual machines, one for Commodore's services and the other two for our Kubernetes cluster (one master, one worker node). We chose Kubernetes version 1.6[2] for our system.

More specifically:

- **Commodore:** This machine hosts all of Commodore's services as well as its database. Those services are:
  1. The collector service.
  2. The Kubernetes manager.
  3. The infrastructure manager.
  4. The rule engine.
  5. The monitoring user interface.
- **Kubernetes Master:** This machine acts as Master node of the Kubernetes cluster and hosts all the services that are essential for a cluster to operate, more specifically:
  1. The etcd database.
  2. The API server.

[1]https://www.fiware.org/

[2]https://github.com/kubernetes/kubernetes/tree/release-1.6

3. The controller manager.
4. The scheduler.

All the above mentioned services are Kubernetes specific and not part of Commodore.

- **Worker Node:** This machine acts as a worker node of the cluster (meaning that applications can be deployed upon it) and consists of the following services:

  1. The kubelet.
  2. The proxy.
  3. The docker run-time.
  4. The collector daemon process.

  These services are Kubernetes specific except collector daemon process, which is collector's sub-component that reports node utilization metrics to Commodore as explained in section 4.4. Also, throughout the evaluation process, Commodore will add more worker nodes (virtual machines), as the load increases. Each new worker node will have same running services as this one.

6.2 Application and Deployment

To show proof of concept and study scalability and performance of a Kubernetes deployment with Commodore, we needed an application to be deployed on the cluster so we can scale it (up or down) and see if Commodore also scales the infrastructure accordingly. We implemented a simple application that computes all prime numbers up to 10^6.

Our goal was to measure response time as the application and the infrastructure scales, by utilizing only compute resources (CPU and RAM) and without any dependance upon bandwidth or disk.

6.2.1 The Testing Application

We wrote the application using PHP 7 as our programming language, Slim Framework in order to expose it as a Web service and Apache web server to access it. The whole application is shown at Listing 6.1

LISTING 6.1: The Testing Application.

```
<?php
use Slim\App;
use Slim\Http\Request;
use Slim\Http\Response;
```

```
require_once 'vendor/autoload.php';

$app = new App;

$app->get('/{to}',function (Request $request, Response
   $response, $args) {
   $primes = [];
   for ($i = 0; $i <= $args['to']; $i++)
      if (isPrime($i))
         $primes[] = $i;
   return $response->withJson($primes);
});

function isPrime($num) {
   if($num == 1) return false;
   if($num == 2) return true;
   if($num % 2 == 0) return false;

   $ceil = ceil(sqrt($num));
   for($i = 3; $i <= $ceil; $i = $i + 2)
      if($num % $i == 0)
         return false;
   return true;
}

$app->run();
```

Next step was to containerize the application using Docker. Luckily, since PHP and Apache is a well known stack for Web applications, there are a lot of publicly available Docker images that suite our needs. We chose to use the official image from the PHP repository. After acquiring a run-time environment we enable the rewrite module and copy our code to the container. In order to automate the procedure of building the image, we implemented a Dockerfile (Listing 6.2) which can be called and execute the above mentioned steps.

LISTING 6.2: The Dockerfile.

```
FROM php:7.0-apache
RUN a2enmod rewrite
COPY . /var/www/html/
EXPOSE 80
```

6.2.2 The Deployment

As mentioned in Chapter 2, the first step in order to deploy a containerized application to Kubernetes is to create an application pod. A pod is a group of Docker containers which compose the deployed application or service (in our example our application consists of only one container). Listing 6.3 illustrates the `Pod` object of our application.

LISTING 6.3: The Kubernetes Deployment Object.

```
kind: Pod
apiVersion: extensions/v1beta1
metadata:
  name: primes-deployment
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: primes
    spec:
      containers:
      - name: primes
        image: cchristodoulopoulos/php-primes:v1.0
        ports:
        - containerPort: 80
        resources:
          limits:
            cpu: 500m
            memory: 512Mi
          requests:
            cpu: 500m
            memory: 512Mi
```

Next step is to create a `Service` object which acts as a load balancer for our pods. A load balancer serves no purpose since there is only one replica of our pod, but throughout the evaluation process, as the number of pods increases, is essential because it will distribute the incoming load among our pods.

LISTING 6.4: The Kubernetes Service Object.

```
kind: Service
apiVersion: v1
metadata:
  name: primes-svc
  labels:
    app: primes
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 80
  selector:
    app: primes
```

Last think is to expose the `Service` object to ingress traffic. There are multiple ways of doing this, most of them are defined directly to the `Service` object (usually by changing the type of the service from NodePort to LoadBalancer) but in order for this to work you need cloud provider integration. Since we don't have that, we created an `Ingress` object which routes incoming requests to our `Service` object.

LISTING 6.5: The Kubernetes Ingress Object.

```
kind: Ingress
apiVersion: extensions/v1beta1
metadata:
  name: primes-ingress
spec:
  rules:
  - host: primes.example.com
    http:
      paths:
      - backend:
          serviceName: primes-svc
          servicePort: 80
```

At this point, the application is deployed on Kubernetes and we can start the evaluation process.

6.3 Experimental procedure

In order to find a good starting point for our tests, we begun with one instance of our pod. The average response time for the operation that computes all prime numbers until 10^6 is about 1400 milliseconds from the time we sent the request until the time we get a response. In terms of infrastructure, our cluster starts with one worker node (Node 1) which has 1 CPU core and 2 GB of RAM (small flavor).

We also concluded that in order to see if we can actually scale on such environment, we had to standardize the benchmarking workload by keeping the number of requests (n) and the number of concurrent requests (c) consistent (i.e. for each test the workload stays the same). In order to execute those requests and get the actual metrics, we used Apache Bench (ab), which is a tool for benchmarking (HTTP) servers.

Apart from the load, we also had to standardize on the resources that each pod could potentially consume, in which we concluded that half of one CPU core and 512MB of RAM (as illustrated in Listing 6.3) would be sufficient values.

Our test suite consist of five runs with increasing number (1, 2, 4, 8 and 14) of pods (we change the number of pods manually). All the experiments were executed with the same workload of incoming requests (n=500, c=4). For each run, we present a graph of the response time as well as a diagram of the current infrastructure.

Our goal for these tests is to observe and evaluate:

- how an application scales horizontally (i.e. by creating multiple replicas - pods of the application) on Kubernetes.
- how Commodore expands the infrastructure as new application pods are created.

6.3.1 One pod

Our first test was executed with one pod of our application. As Figure 6.1 shows the average response time was around 5000 milliseconds.

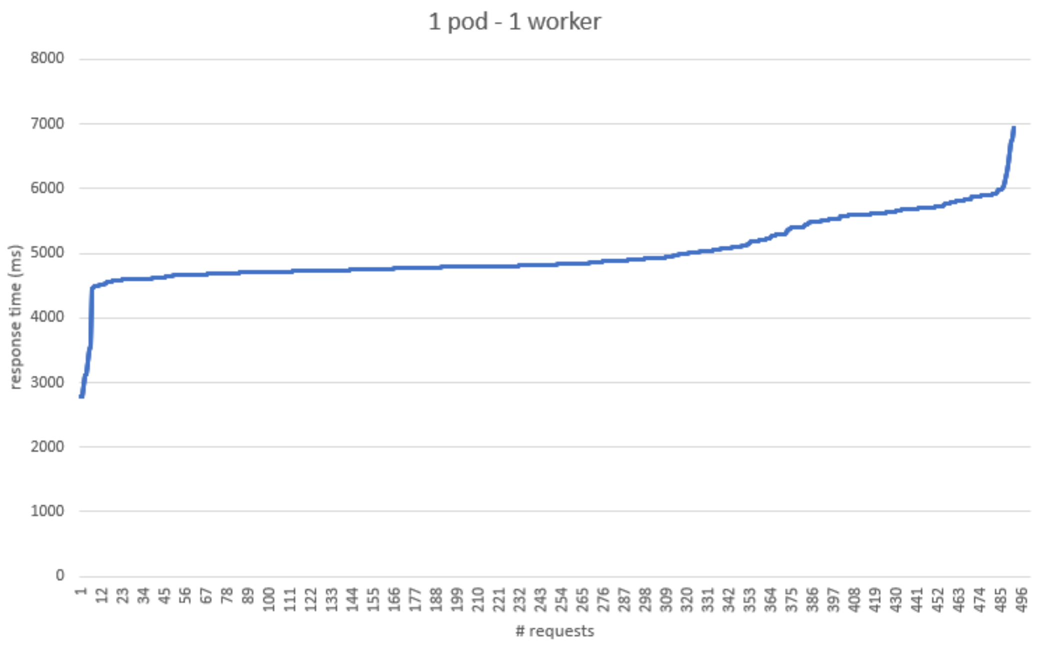


FIGURE 6.1: Response time - one pod.

Commodore didn't perform any scaling operations since our pod requested only a half of one CPU core and 512 GB of RAM. Scheduler deployed it to Node 1 which had available resources.

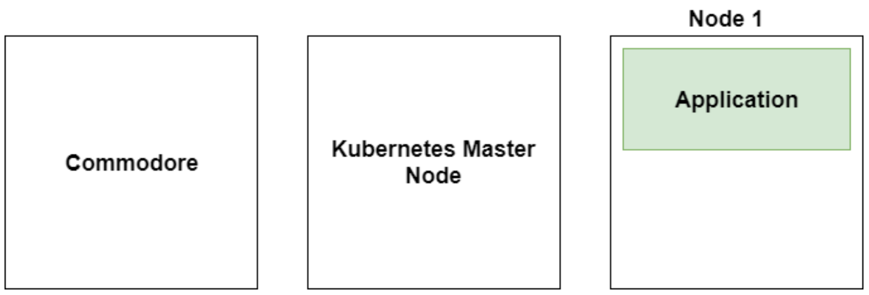


FIGURE 6.2: Infrastructure with one pod deployed

6.3.2 Two pods

Next, we scaled the application up to two pods and the average response time dropped to 2600 milliseconds, which is half of the previous test. This number makes sense because Kubernetes balanced the load between the two pods.

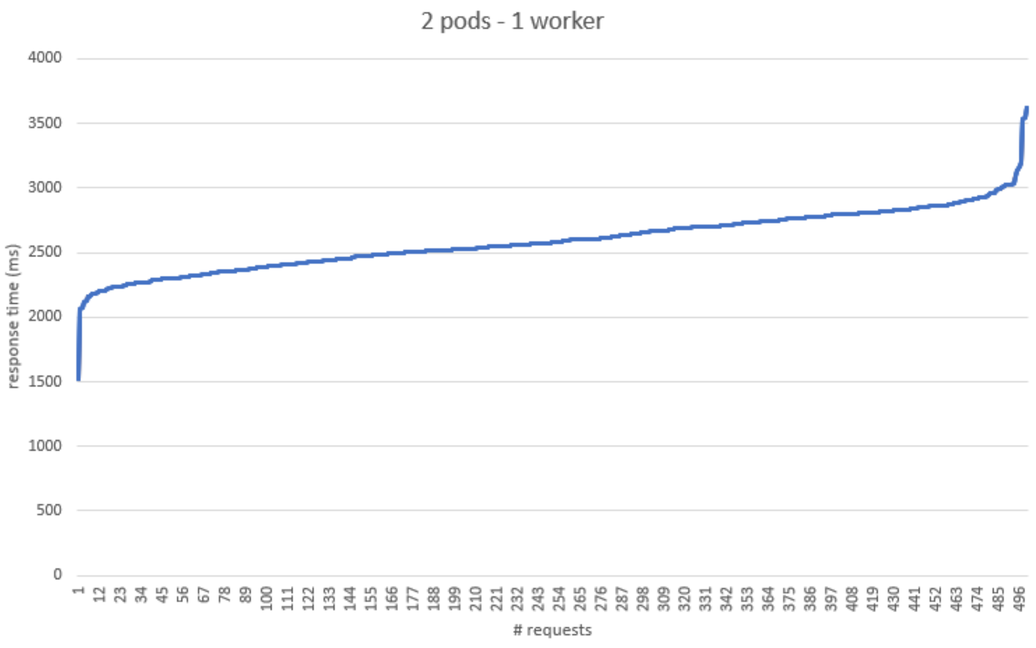


FIGURE 6.3: Response time - two pods.

Commodore still didn't perform any operation since the scheduler deployed the new pod to Node 1 successfully.

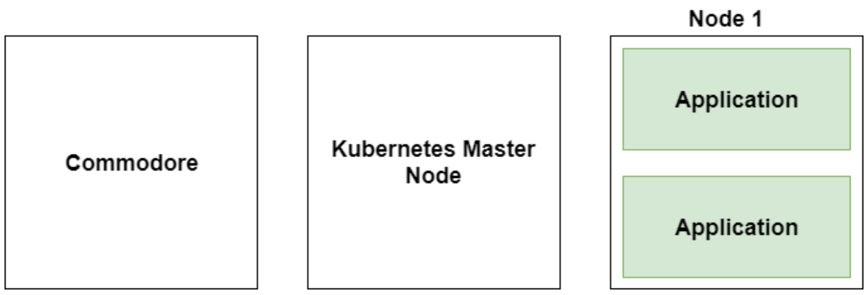


FIGURE 6.4: Infrastructure with two pods deployed

6.3.3 Four pods

Next we scaled the application up to four pods. Kubernetes scheduler couldn't deploy the two new pods to Node 1 since it had no available resources, therefore it emitted two 'FailedScheduling' events. Commodore receives the events and calculates the resources needed in order for the new pods to be scheduled and starts a new small flavor machine (the same as Node 1). The machine binds to the cluster and the scheduler deploys the two pods to the new machine. Then we run the benchmark and the average response time dropped to 1700 milliseconds.

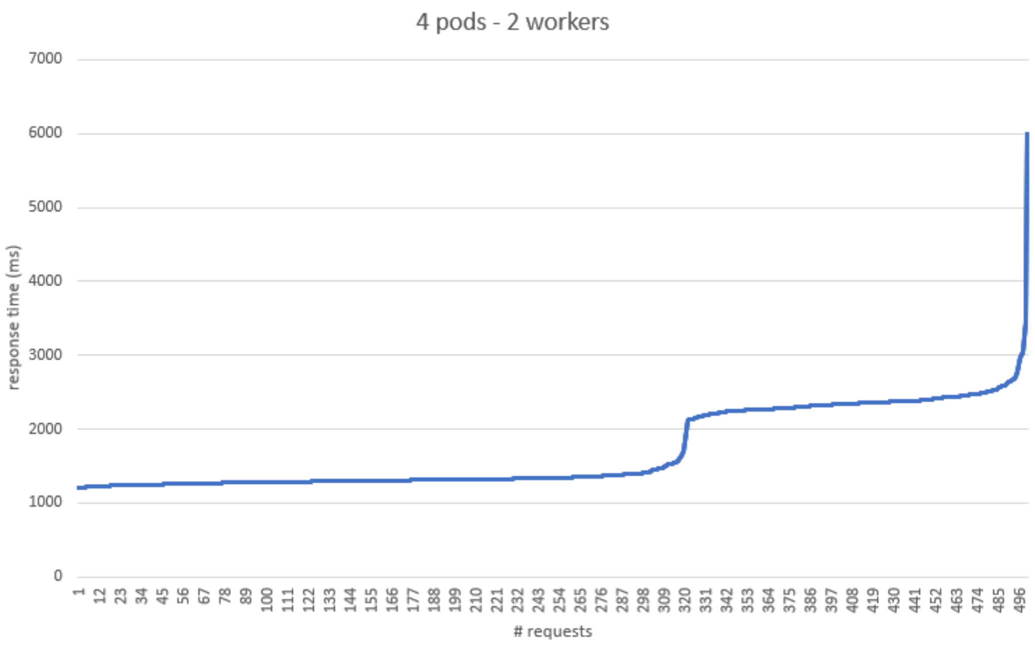


FIGURE 6.5: Response time - four pods.

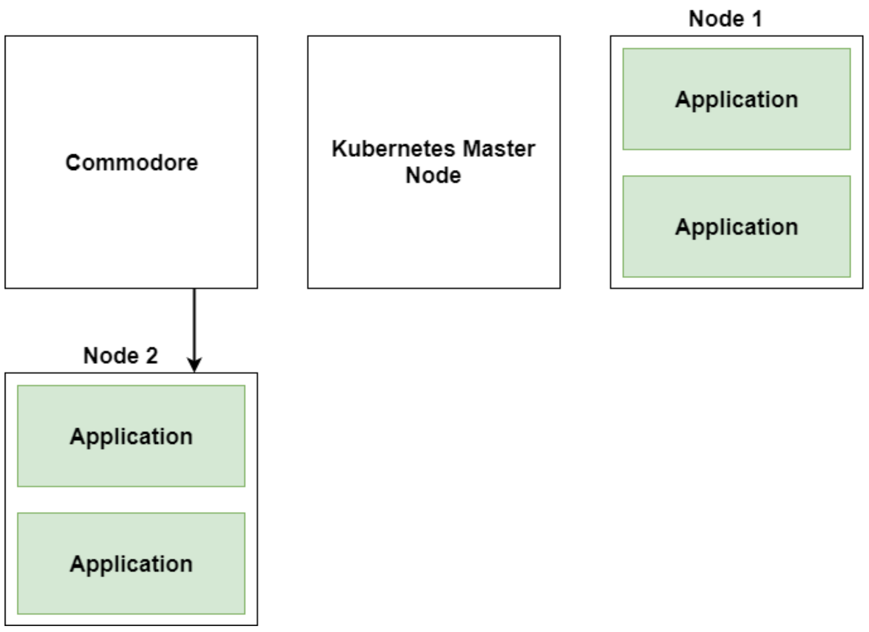


FIGURE 6.6: Infrastructure with four pods deployed

6.3.4 Eight pods

Next we scaled the application up to eight pods. Again, Kubernetes couldn't deploy the four new pods and emitted four events. Commodore received the events, calculated the sum of the requested resources and started a new medium flavor machine (2 CPU cores, 4 GB of RAM) for the new pods and Kubernetes scheduler deployed them. Then we run the benchmark and the average response time was 1500 milliseconds.

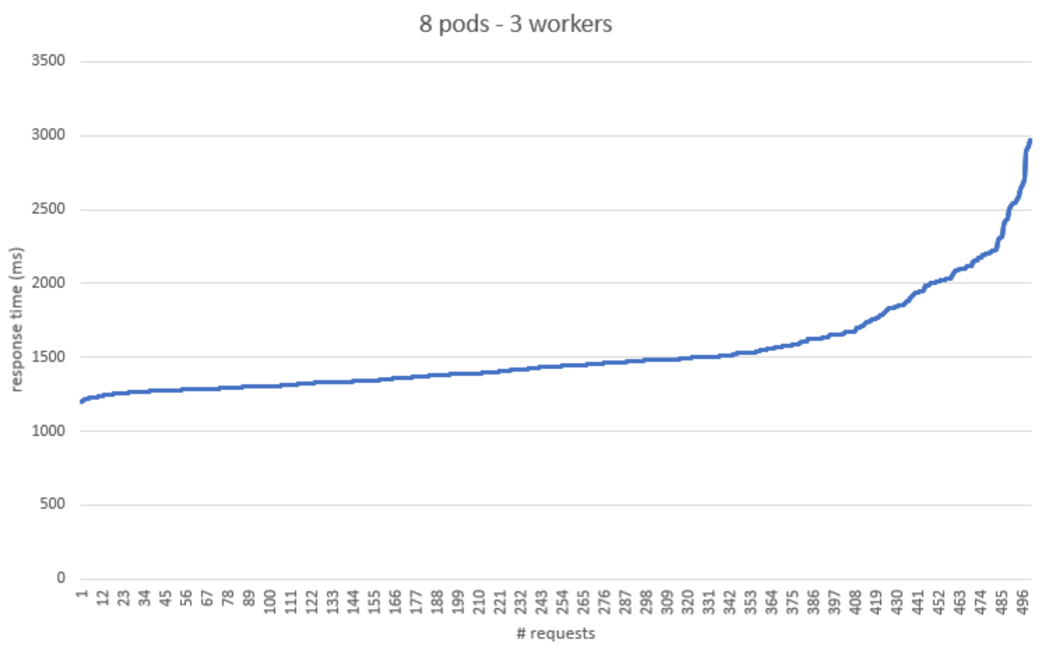


FIGURE 6.7: Response time - eight pods.

It is worth noticing that as we reach our idle response time (1400 milliseconds), although we doubled our resources as well as our pods, response time wasn't affected as much as previously (from one to two or from two to four pods).

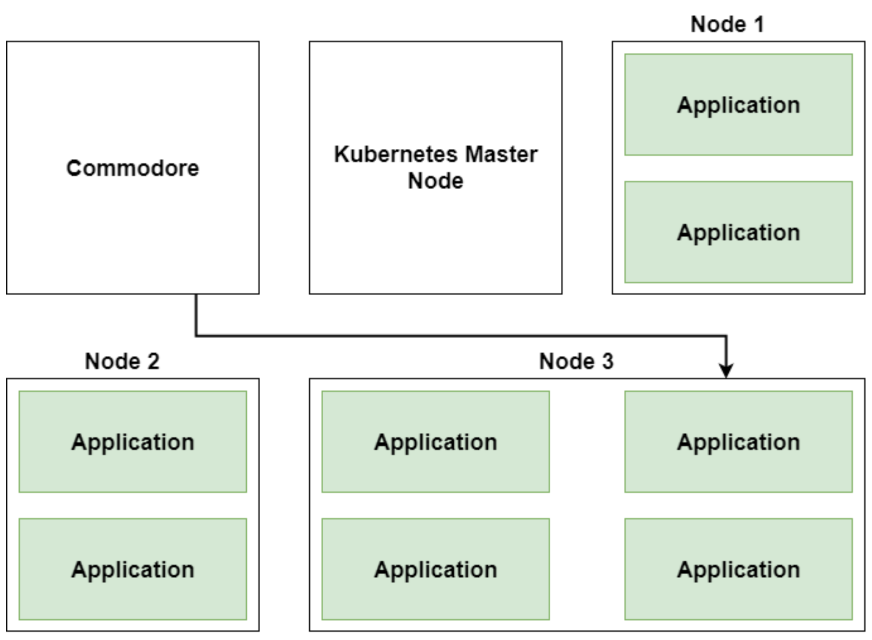


FIGURE 6.8: Infrastructure with eight pods deployed

6.3.5 Fourteen pods

Lastly, we scaled the application up to fourteen pods. Kubernetes couldn't schedule the pods and emitted events. Commodore received them and started two machines, one medium and one small flavor as Figure 6.10 illustrates. Then we run the benchmark and the average response time was 1400 milliseconds.

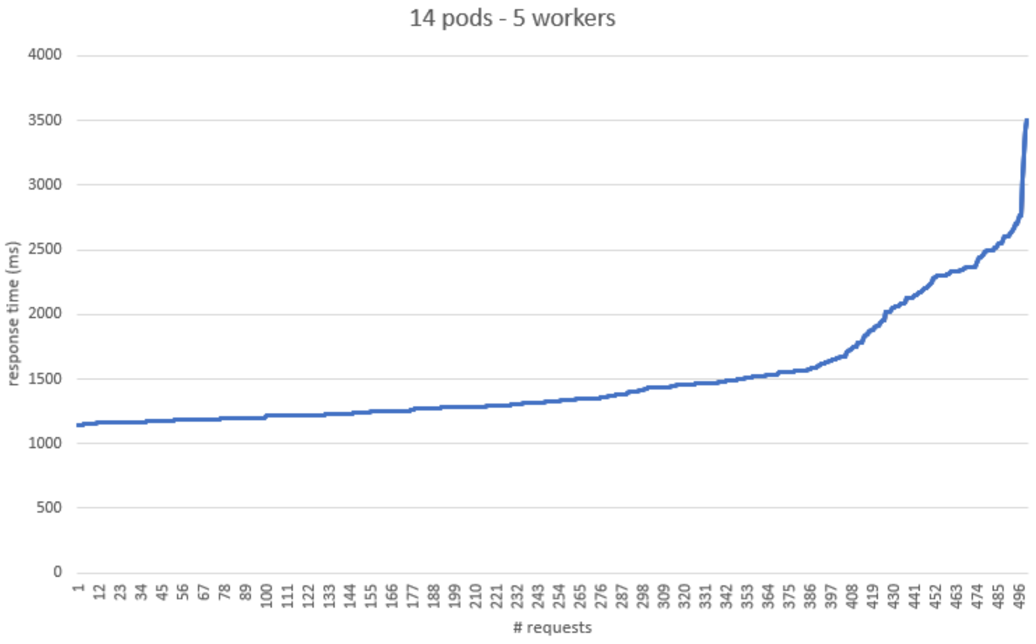


FIGURE 6.9: Response time - fourteen pods.

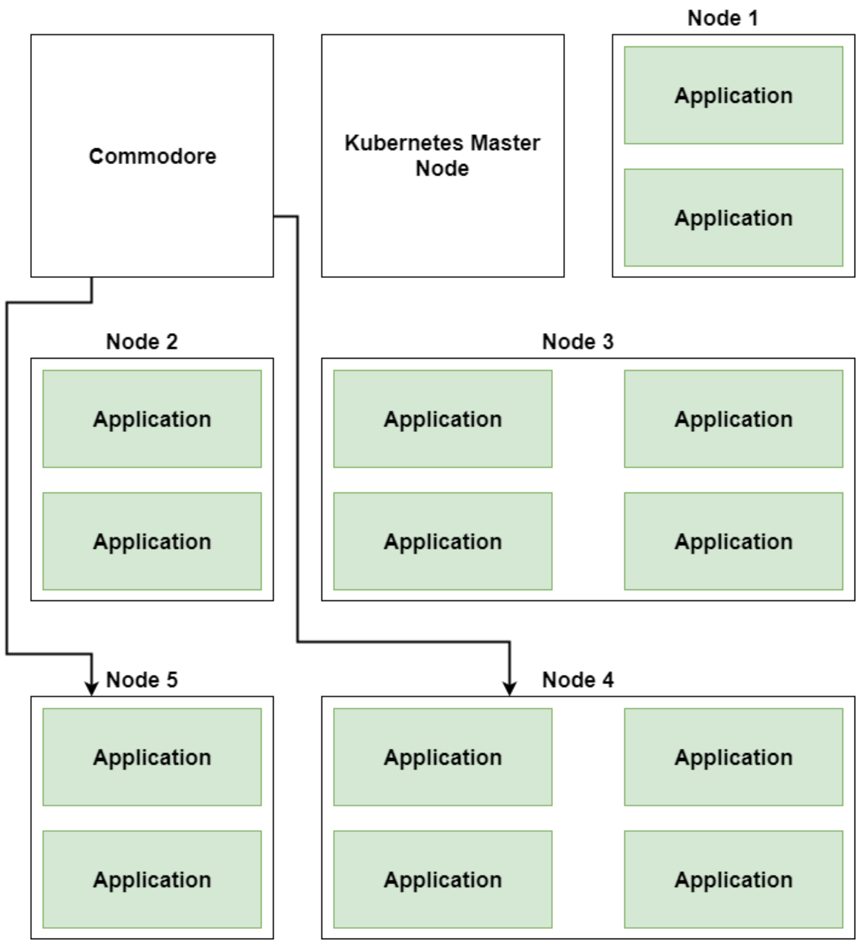


FIGURE 6.10: Infrastructure with fourteen pods deployed

6.3.6 Evaluation Summary

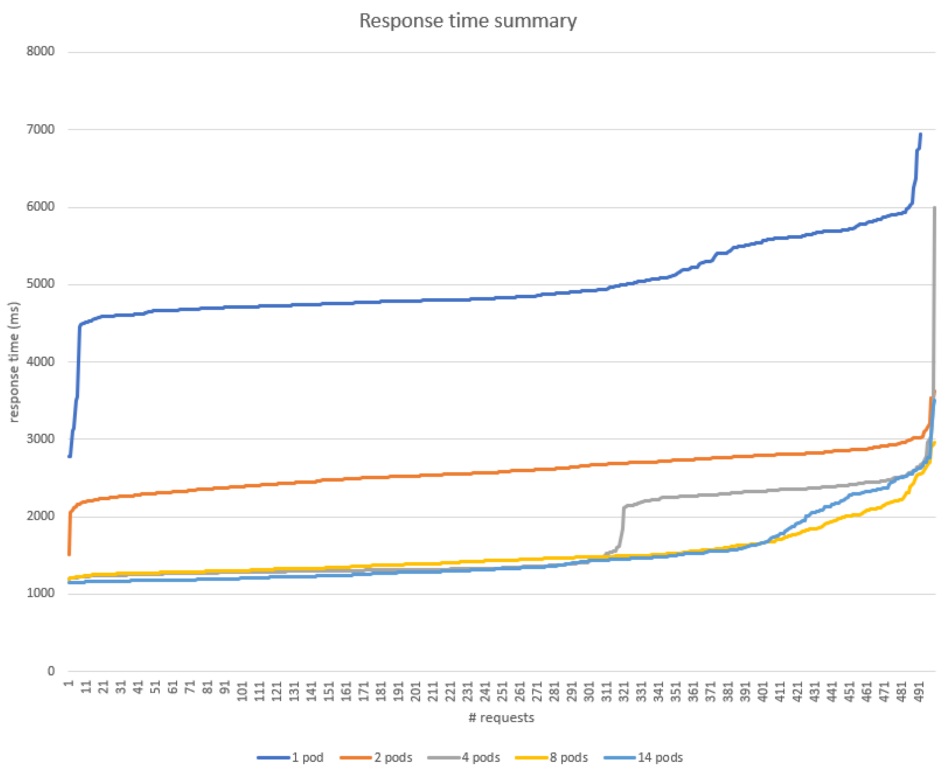


FIGURE 6.11: Response time - Summary.

As Figure 6.11 illustrates, our test application scales almost linearly until the four pods mark. From this point onward, scaling the application didn't yield the same benefits as before since it reached its idle response time. The same graph would have been produced in the case where Commodore wasn't deployed and we manually added new nodes to the cluster. With Commodore in place, we didn't need to, since it scaled the infrastructure accordingly (starting from one small flavor worker machine to five of varying flavors). In this example, without Commodore we wouldn't be able to scale automatically above two pods since Kubernetes wouldn't be able to provide the requested compute resources.

Although this evaluation process illustrated that Commodore operates properly (i.e. scales the infrastructure as the number of pods increases), it didn't promote its actual value. In this example, there was only one application deployed on the cluster and the number of nodes was small, therefore it's easy (e.g. for a system administrator) to determine how many resources are needed in order to deploy the pods. Imagine a cluster with tenths of nodes and hundreds of pods, where new applications are created and existing ones are scaling at the same time. In such cluster, manual infrastructure scaling is not applicable, since resource demands always change. That's the perfect use case for Commodore.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis we introduced, implemented and tested a concept architecture for scaling Kubernetes clusters with an automated manner. Summarizing, the contributions of this work are the following:

- Minimize the cluster resources without compromising the clusters' performance resulting to reduction of operational costs.
- Execute cluster-level operations such as scale up or scale down without affecting the already deployed applications
- Introduce a vendor-agnostic architecture which can be deployed to any cloud vendor with only some configuration changes
- Introduce a modular architecture that features can be added by adding more components and not modifying the existing ones.

7.2 Future Work

The following are important issues for future work:

- Extending collectors' capabilities including a more detailed collection of metrics from the cluster nodes that specifies per container utilization of resources. Such feature, could potentially power an additional service that performs pod-level operations such as altering the requested resources of the deployed pods and also changing the number of replicas based on actual utilization. Such system could introduce actual elasticity at Kubernetes level, on top of autoscaling at infrastructure level.
- Utilization of the collected metrics in order to gain insights on a business level, like peak traffic hours, most utilized services etc.
- Improve the rule engine component in order to operates in an event driven matter, e.g. when specific events occur, and not using time intervals like it does now.

Bibliography

[1] Jonathan Baier. *Getting Started with Kubernetes*. Packt Publishing Ltd., 2015. ISBN: 978-1-78439-403-5.

[2] Dan Radez. *OpenStack Essentials*. Packt Publishing Ltd., 2015. ISBN: 978-1-78398-708-5.

[3] James Turnbull. *The Docker Book*. 2016.