UPPSALA UNIVERSITET

UPTEC X 17020
Examensarbete 30 hp
September 2017

Kubernetes as an approach for solving bioinformatic problems

Olof Markstedt

UPPSALA UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Master Programme in Molecular Biotechnology Engineering

Olof Markstedt

The cluster orchestration tool Kubernetes enables easy deployment and reproducibility of life science research by utilizing the advantages of the container technology. The container technology allows for easy tool creation, sharing and runs on any Linux system once it has been built. The applicability of Kubernetes as an approach to run bioinformatic workflows was evaluated and resulted in some examples of how Kubernetes and containers could be used within the field of life science and how they should not be used. The resulting examples serves as proof of concepts and the general idea of how implementation is done. Kubernetes allows for easy resource management and includes automatic scheduling of workloads. It scales rapidly and has some interesting components that are beneficial when conducting life science research.

Handledare: Jonas Persson
Ämnesgranskare: Ola Spjuth
Examinator: Jan Andersson
ISSN: 1401-2138, UPTEC X 17020

Sammanfattning

Bioinformatik är ett tvärvetenskapligt område som kombinerar informationsteknik och bioteknik. Syfte kan vara att utveckla nya metoder och tillvägagångssätt för att analysera och förstå biologisk data i så kallade arbetsflöden. Många arbetsflöden kräver flera olika verktyg som tenderar att vara experimentella och väldigt beräkningsintensiva. Tillämpningar kan vara att analysera tumörceller med syfte att hitta somatiska varianter. Nya framsteg inom sekvenseringsteknologin har lett till en explosion i tillgänglig biologiskdata. För att kunna analysera all data på ett effektivt sätt krävs smarta metoder. Vid beräkningsintensiva analyser används oftast datorkluster för att snabbare kunna utföra analyserna.

Plattformen Kubernetes har utvärderats och testats i detta arbete för att undersöka dess användningsområden inom ”life science”. Fokus har legat på att undersöka hur tekniken kan användas för att effektivisera och underlätta för resultats reproducerbarhet och distributionen av arbetsflöden. Kubernetes är ett kluster orkestreringsverktyg som kan användas till att skapa en miljö där beräkningsintensiva uppgifter kan köras. Tekniken underlättar resurshantering och isolering vilket kan leda till en effektivisering i hur forskning genomförs. Kubernetes är en ny teknik och använder sig av en relativt ny teknik som kallas ”containrar”. Containrar är ett sätt att isolera och kapsla in mjukvara och dess beroende av andra mjukvara för att fungera. Detta gör det enkelt och smidigt att dela med sig av verktyg och arbetsflöden vilket underlättar reproducerbarhet och distribuering. Slutsatsen av arbetet är att Kubernetes och containerteknologin är lämpliga tekniker att använda inom ”life science”. Containerteknologin erbjuder agnostiska lösningar som simplifierar delandet av verktyg och i sin tur reproducerbarheten. Kubernetes är även lämpligt för det gör användandet av containertekniken lättare och resurshanteringen blir väldigt konkret och enkel för användare att nyttja.

Contents

Abbreviations

API	Application programming interface
AWS	Amazon web services
Container	Linux Container
GCE	Google compute engine
JSON	Javascript object notation
OS	Operating system
REST	Representation state transfer
SNP	Single nucleotide polymorphism
VCF	Variant call format
vCPU	Virtual CPU
VM	Virtual machine
WGS	Whole genome sequencing
YAML	Yet another markup language

1 Introduction

The goal for this thesis has been to evaluate the applicability of the container orchestration tool Kubernetes as an approach to instantiate a Kubernetes cluster, facilitate easy deployment and reproducibility of bioinformatic workflows. It allows for the user to easily allocate resources and provides an intuitive front-end for logging. It is meant to scale horizontally, meaning it will create new workers if the workloads required resources are larger than the resources available. Kubernetes is a cluster orchestration tool used to facilitate deployment of containerized software. It runs on any cluster on-premise, public or multi-cloud and on bare-metal. Kubernetes leverages the advantages of containerized software which makes deployment and reproducibility easy. It allows for users to build and package software into containers without regard for the underlying host architecture making it easy to develop and deploy bioinformatic workflows. It comes with a lot of built in services that makes it highly usable for life science workflows. Examples of how these can be utilized in life science research was evaluated in this thesis. A web application was built with the goal to test some of the relevant Kubernetes functions and its applicability on bioinformatic problems.

A life science workflow was containerized and tested in a Kubernetes cluster to determine how well Kubernetes handles heavy workload and its scalability. A Kubernetes component called a “job” and an asynchronous task queue was used for performance testing. Some other examples and some not so suitable examples for Kubernetes are discussed. The bioinformatic workflow used for the evaluation was the Cancer Analysis Workflow (CAW), developed at SciLifeLab [1] (https://github.com/SciLifeLab/CAW). It is a suitable example to use for testing because it requires a lot of dependencies. The pipeline is used for detection of somatic variants from whole genome sequencing (WGS) data. The resulting variant call format (VCF) files are merged and can be used for further downstream processing. The evaluation was done from a perspective of how difficult a Kubernetes cluster is to orchestrate, how containers are built in an efficient manner, how well it handles heavy workload and how well it performs compared to other tools.

2 Background

As sequencing technologies become more sophisticated, the cost of sequencing decreases, yield more throughput, has become portable and in 2017 smartphone compatible (SmidgION), the amount of public data available in the world of biology is rapidly increasing and has been measured to be in the size of petabytes [2]. To be able to analyse the data, sophisticated workflows are required. The genomic workflows used for analysis often consist of several pieces of third-party software and are very experimental which leads to frequent updates and changes thus creating deployment and reproducibility issues [3]. It has previously been proposed [4] that virtual machines (VMs) could solve the problems of deployment and reproducibility, however, only using VMs has some disadvantages. Due to this, new smart software solutions are needed to be able to efficiently analyse the data and make improvements to existing software in such a way that it decreases the dependency on Moore's law. Most bioinformatic workflows are often executed in a high-performance computing environment which requires extensive hardware to provide relatively fast results which tends to be very expensive, requires a lot of service (e.g. central processing units (CPUs), graphical processing units (GPUs), RAM and storage) and even if one has access to the hardware it can still be insufficient (for example if there are many users running heavy workloads). The number of NGS-projects has increased rapidly over the last six years [5], resulting in a higher amount of storage used and active users.

There are several problems that needs to be addressed and many can be solved using cloud based solutions and container technologies such as Docker [6]. The structure and principle of Docker containers makes the technology very prominent and efficient for bioinformatic workflows. One of the reasons being that once a Docker container has been built, it will run on any platform regardless of the operating system (OS) it runs on which simplifies reproducibility and deployment [7], the only restriction being that the environment needs Docker installed. Docker bundles the libraries and dependencies into a container that runs in an isolated environment which prevents conflicts with other installed software in the host environment. Because of this it becomes easy to share workflows across different platforms and it makes reproducibility easy.

Another problem that needs to be addressed when working with containerized software is how to manage them in a reliable and efficient manner. Kubernetes [8] is an open-source system for managing containerized software in a cluster environment. A Kubernetes cluster can run in any environment and has different concepts that adds higher levels of abstractions which makes this type of system very useful when deploying containerized software running bioinformatic workflows. Similar investigations by others regarding reproducibility of workflows has previously been done [9, 10], but to scale the workload in an efficient manner a tool to orchestrate containers is needed. Some workflows are more CPU intensive and some are more memory (RAM) intensive, by leveraging the flexibility of cloud providers it allows for the user to run the workflow on different machine types given that the user has some knowledge about the workflow. My thesis focuses on how and why Kubernetes and Docker running in a cloud environment can be used as an approach to efficiently run bioinformatic workflow to increase productivity and solve many bioinformatic problems.

3 Cloud Computing

Cloud computing is a big field and this thesis will not go into greater depth of the subject. This section will only give a brief overview of some of the terminology and concepts. There are many articles [11-13] that goes into greater depth. Cloud computing is defined by the National Institute of Standards (NIST) [14] as “a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction.” Adopting a cloud computing model can give economic advantages as opposed to buying hardware [14], basically what this means is that a company or organization outsources their hardware.

There are several cloud providers that provide different services. Generally, there are three major types of services are infrastructure as a service (IaaS), platform as a service (PaaS) and software as a service (SaaS). This thesis will mainly focus on IaaS and PaaS. IaaS is the basic service provided by cloud providers, it lets the user create VMs and networking across the web and is very configurable. While IaaS is essentially the building blocks for the virtual environment, PaaS is built on the IaaS components but provides a platform for the user to easily ship and deploy software without having to configure the underlying infrastructure.

4 Why containers?

4.1 Virtual Machines

A virtual machine is an emulation of the underlying hardware on the host OS [15]. Virtual machines are created from an image and these are typically full copies of a specific operating system which tends to be very large and take time to deploy. A hypervisor running on the host machine is required, this can be seen in Figure 1. The role of the hypervisor is to schedule and allocate resources to the guest OS and to interpret the instructions from the guest OS for execution on the hosts hardware [16]. The VMs are isolated from the host OS and other VMs running on the same machine, however, since VMs are entire operating systems and demands a lot of resources from the underlying hardware leading to inefficient use of the resources and causes limitations in the number of VMs that can run on a single host.

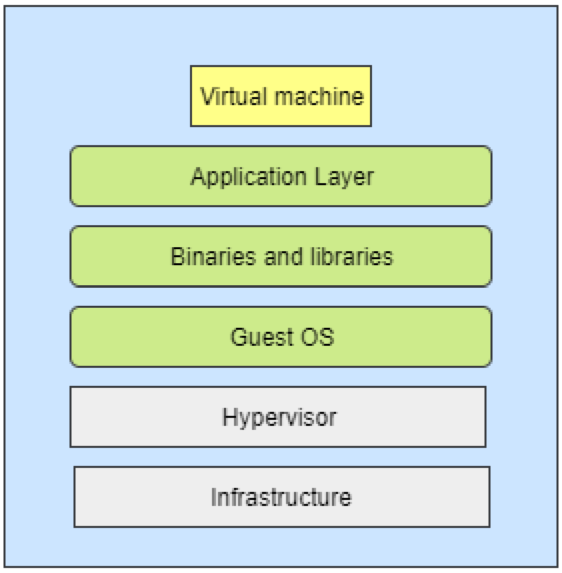


Figure 1: Schematic view of a VM. The use of VMs adds the extra hypervisor layer that oversees scheduling and interpret the guest OS instructions for hardware execution.

When updating the VM, the image itself must be reassembled and redeployed even if there is only a single change in one file. Customizing an image often requires the user to access the VM and run updates/changes/installation, this makes it difficult to track changes made to the image if one wants to assemble the image from scratch. They tend to be resource-intensive [18], inflexible, heavyweight and non-portable. VMs should however not be disregarded since containerized software in the cloud needs a host to run on.

4.2 Linux Containers

A Linux container (container) can be described as a “lightweight virtual machine”[18]. But while a VM emulates the physical resources of the host machine, a container is a process level wrapper that isolates the software within the container through Linux namespaces [19]. This basically means that the virtualization occurs on the operating-system level. It enables the kernel to allocate resources such as CPU, bandwidth, disk I/O and RAM-memory etc. so that the processes within the container have full access to the allocated resources without interference from the rest of the system. As can be seen in Figure 2 containers are decoupled from the underlying infrastructure, host filesystem and isolated from one another. This enables containers to be portable to different cloud environments and operating systems. Each container can be based on a different OS and run different or even the same processes. As previously mentioned this allows for one container to run a specific version of a software, while another container within the same system runs another version of the same software without causing dependency conflicts.

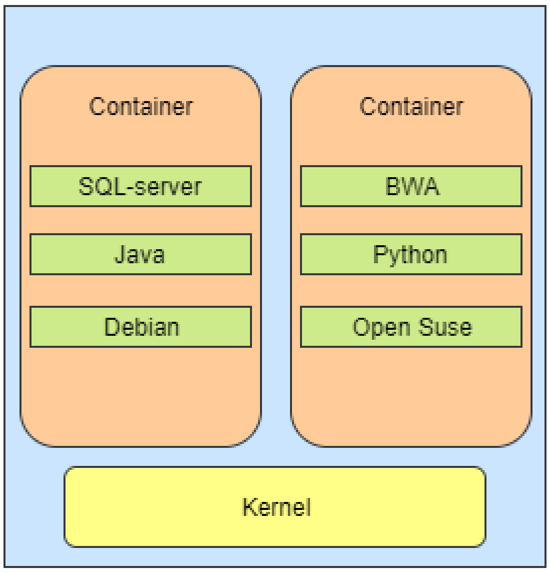


Figure 2: Visualization of containers running on a Linux Kernel. Every container sits on top of the hosts kernel running different software on different operating systems in an isolated manner.

The advantages of using containers over VMs are many. Instead of installing software on a host using the package manager of that OS which can easily cause dependency conflicts and making it hard for the user to know which dependencies are necessary, a container is built with the required dependencies and libraries required by the software. Because the container can be built with only the necessary dependencies for the software to run it will decrease the size of the container making containers much smaller and more lightweight than VMs. In Figure 3 is a typical bioinformatic pipeline. Each step can be containerized instead of running the entire pipeline in for example a bash script. If a tool used in the workflow is missing but has been built, it can simply be pulled down to accommodate for that missing tool which in turn leads to faster workflow creations from a software point of view. There are many tools that has been containerized by bioinformaticians, “BioContainer” [20] is an effort to increase the number of containerized life science tools and make them easily accessible to the public.

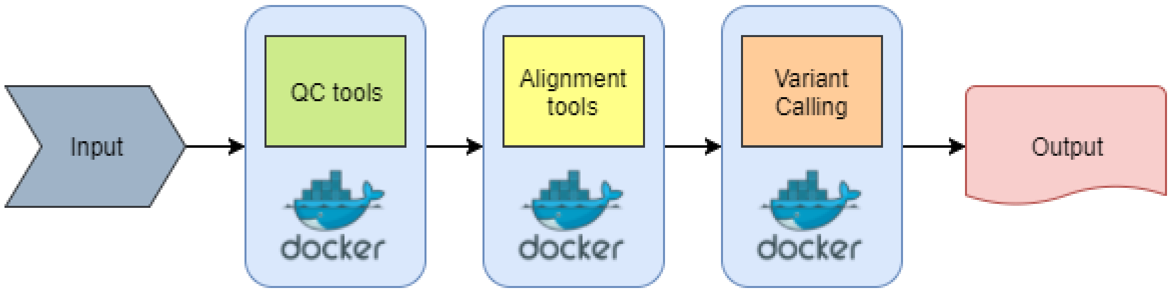


Figure 3: A typical life science pipeline used for variant calling from raw NGS-data. There can be several tools in each container or several containers for each tool. The latter would be the preferred choice to fulfil the purpose of containerized software and making the containers more lightweight.

4.3 Docker

To be able to efficiently facilitate the container technology, a software for container management and runtime is required. There are several others but in this study Docker was used. Docker is convenient to use because it has a large active online community with many prebuilt containers available for the public and is rapidly growing in popularity amongst the scientific community. Many bioinformaticians has started to adopt a more containerized approach. Since containers are system-agnostic executable environments it is very beneficial to use as a life science researcher because it allows for easy sharing of tools and other software without regard for the target OS. For example, if a tool used in research has accidentally been updated and a specific feature from the last version has been removed or changed causing the behaviour of the tool to be inconsistent to that of the previous result, the user would have to rebuild the entire tool which can take some time. With containerized software, this is not necessary.

Docker use an image as a base for creating the container. The user can declare environment variables and add software with its dependencies in something called a Dockerfile. Docker then builds the container based on that image to create a stand-alone executable package with all the dependencies and libraries, this is the resulting image. Docker adds a thin layer on top of the host OS for managing the containers at runtime and during the build process.

During the build process, Docker creates and documents union file system layers for each action which describes exactly how to recreate an action. If another action is added, Docker will only repeat the process for that specific action, making this technology much more efficient compared to VMs. When the image is being built, the user can give the image a unique name and push it to the Docker hub (https://hub.docker.com/) which is a public repository for Docker images after the build process has finished. This makes it very easy to share and use Docker images and once the image has been pulled from Docker hub it will be up and running in a matter of seconds without the need for installation of any software other than Docker, making reproducibility trivial.

5 Kubernetes

Kubernetes [8] was originally created by Google but was later donated to the Cloud Native Computing Foundation. It is a cluster orchestration tool that follows the principle of the master-slave architecture but adds higher level of abstractions which are the building blocks of the Kubernetes cluster. Kubernetes makes it easy to deploy applications in a predictable manner, scale and limit hardware usage. There are many of these blocks and only the essential ones that cover the scope of this thesis will be covered. It is possible to run a Kubernetes cluster locally on a laptop using Minikube [21].

It is important to take into consideration that Kubernetes is still a new system and some of the functionalities used are still in the beta stage and subject to change.

5.1 Kubernetes Basics

Every object created in the Kubernetes cluster are done in a declarative fashion. The user specifies what kind of object that should be created within a configuration file called a “template” or “manifest” in the form of JavaScript object notation (JSON) or Yet another mark-up language (YAML) format. Each item in the manifest is a set of key value pairs.

5.1.1 Pods

The most primitive building block in a Kubernetes cluster is called a “pod” [22], these are always scheduled to a worker node within the cluster and serves as a unit for deployment and horizontal scaling. A pod is an object that adds a higher level of abstraction to containerized software. It consists of one or more containers with shared resources running in an isolated environment within the pod and each pod has a unique IP-address. The pod is ephemeral, meaning they are expected to be short-lived. When a pod dies, it will not restart, instead a new pod is created and everything from the previous pod is destroyed. Due to this a pod is rarely declared on its own, to ensure that at least one pod is running at all time a replication controller is created. If the pod creates data (e.g. workflow results) during runtime that needs to be stored for later access, a volume (i.e. persistent storage) can be attached to the pod for read and/or write access.

5.1.2 Replication Controller

When a replication controller [23] is created, its purpose is to ensure that the declared number of pod replicas are always up and running in the cluster. If a node with pods on dies or fails, the replication controller will then reschedule that pod to another worker node. The number of pods can be scaled at any time once the replication controller is up and running. Scaling occurs on the replication controller, if the number of pod replicas are set to two during initiation it will create two pods. If, however the number of replicas are changed during runtime it will simply create enough or delete pods to match the new number of replicas.

There can (most likely) be several replication controllers running at the same time in a cluster. To be able to target the correct replication controller and its pods, labels are used. A label is a string assigned to a replication controller (or pod) which can be “backend”, “frontend”, “workflow” etc.

The new generation of replication controllers is called a Deployment [24], it works much like a replication controller but adds some extra features such as the ability to conveniently apply rolling updates and also rolling back to previous versions.

5.1.3 Services

The service [25] abstraction defines how to access the pods running in the cluster. Like all other abstraction concepts, a service is declared in a configuration file. The pods are not immediately accessed using their unique IP-addresses (as they are ephemeral) but by using labels and selectors. The selector is declared in the configuration files service and targets a set of pods (labels) created by the replication controller. These services are necessary to allow the pods to communicate with each other in a reliable manner.

A typical Kubernetes cluster structure is seen in Figure 4. A replication controller has been created which has been instructed to create four pod replicas running any arbitrary containerized application, for example a web application. The replication controller will make sure that 4 pod replicas are always up and running and schedules them to the nodes in the cluster. It will perform the scheduling by always scheduling a pod to a node that has the least number of pods running. This service acts as a load balancer which routes internet traffic to the pods. The service uses labels and selectors to know which pods to route the traffic to.

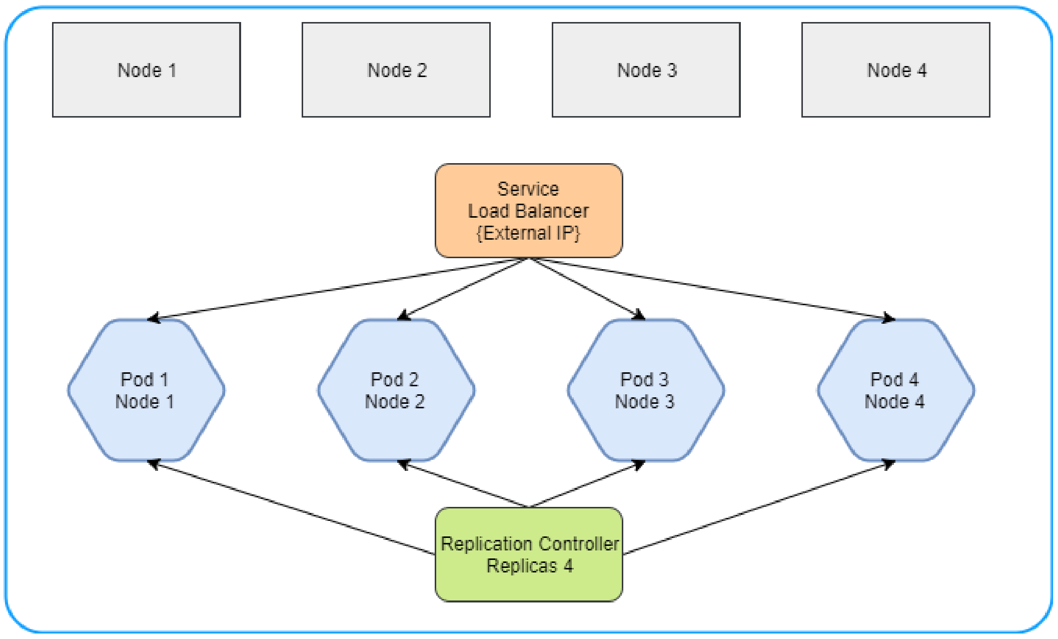


Figure 4: A schematic overview of a typical Kubernetes cluster. This cluster has 4 nodes. There is a replication controller assigned to the cluster handling the scheduling of pods. The Service acts as a load balancer which routes traffic to the pods. Both the service and replication controller has labels and selectors which is represented as the arrows.

If a node in the cluster fails which can be seen in Figure 5, the pod running on that node will be terminated and the replication controller will sense this. This causes the replication controller to create another pod on the most available node, which in this case is node 1. This type of solution will allow for constant uptime unless all nodes in the cluster fails.

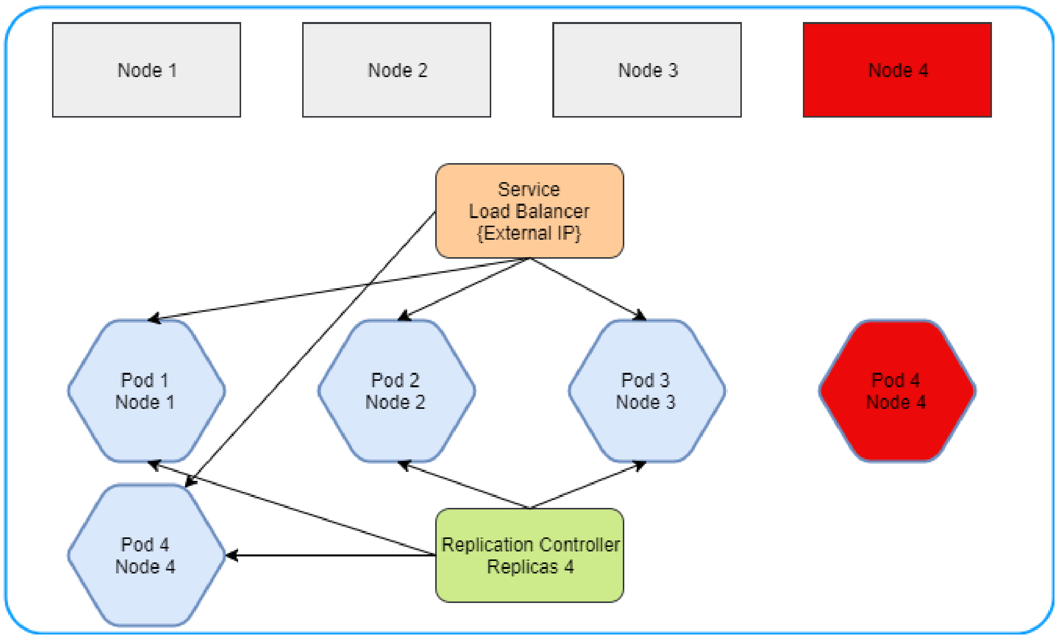


Figure 5: A schematic overview of a typical Kubernetes cluster with one failing node. The failing node is node 4 which is coloured red and its corresponding pod (pod 4) can be seen terminating as well. The replication controllers respond to this is to create a new pod on a healthy node. The service will match its selectors to the labels of the new pod which is the same as the previous pod.

5.1.4 Jobs

A job [26] (Kubernetes job) is one of the useful abstractions for workflows that Kubernetes provides. When a job is created, it will in turn create a pod which runs the container that will execute the workflow. It works much like a batch job, it is created and runs until completion. A job is created like all Kubernetes objects, in a declarative fashion. The user specifies what kind of job to run, gives the job a name, specifies the container to use and the command that should start the job once the pod is running. Resource allocation claims and limitations are made in the job template, the resources are specified in CPU and memory (RAM). If the resources are insufficient, the job will wait until new resources are made available, either by scaling the cluster horizontally or if another job has finished and released its resources. This enables efficient resource allocation and makes it easier to limit the amount of resources an individual user is allocating. To be able to store files potentially created from the job, one can mount a volume to the job pod for write access. It is also possible to mount a volume to the pod with only read access and write to a different volume. This is convenient when the purpose of one volume is to hold the data (read) for a workflow and another volume only has the results from the workflow (write).

Running a monolithic job with all the dependencies inside a single container will most likely cause that container image to be very large (CAW requires about 30Gb of dependencies). It is possible to divide the monolithic container into several containers, each running a single tool. This can however be tedious and becomes prone to error. A good solution is to create a volume (persistent storage) and add all the dependencies to that volume and let the jobs pod read from the volume instead. Since volumes are persistent, the dependencies will remain until the volume is deleted.

5.1.5 Namespaces

A Kubernetes cluster allows for several smaller clusters to exist within it, called virtual clusters and referred to as namespaces [27]. The intention of namespaces is to allow many users, multiple teams and projects to have access to their own cluster. It allows the users to run their jobs in an isolated environment separated from others but still in the same context as the “main” cluster. It is easier for the system administrators to orchestrate namespaces in the “main” cluster, thus only having a single cluster to administrate.

5.1.6 Architecture

As mentioned previously, a Kubernetes cluster follows the master-slave architecture. Kubernetes does not inherently deploy the nodes (workers), these are deployed by the cloud providers most commonly as VMs or in the case where commodity hardware is used, by the cluster administrator. Kubernetes is however aware of the nodes that form the cluster giving it knowledge about the condition of each node.

The Kubernetes API is exposed on the master node via the kube-apiserver and each worker node in the cluster proxy’s requests made to the master. On each node, there is a process watcher called the “kubelet”. When a pod or service is created, a YAML or JSON object containing the pod spec is sent to the kubelet via the kube-apiserver which then makes sure that the containers within the pod are running and healthy.

Kubernetes provides a Command Line Interface (CLI) that communicates with the kube-apiserver running in the cluster. This allows for a user to easily deploy pods and services. There is programming language support for the REST-API such as Python [28] (https://github.com/kubernetes-incubator/client-python) which allows for cluster orchestration in a programmatic way.

5.1.7 Kubernetes Drawbacks

Google [29] and Microsoft Azure [30] provides PaaS solutions (Google Container Engine and Azure Container Service) for Kubernetes which are easy to use, however most cloud providers do not. Setting up a Kubernetes cluster on a cloud provider that does not have native support for Kubernetes can be tedious. There are a lot of abstraction concepts and components that is required and necessary to understand. Kubernetes demands a lot from the user when it comes to administration of the cluster but also from the developer such as microservices and containers.

Concurrency can only occur on one node, e.g. if three nodes exists each having 8 cores which is a total of 24 cores and a job is created that requires 16 cores it will not be able to execute because there is no node that holds that many cores.

6 Materials and Methods

6.1 Kubernetes and Docker

The Kubernetes [8] version used for the most part during the work was version 1.6.4, many concepts of Kubernetes are still in beta but works well with 1.6.4. Several Docker [6] versions were used, using newer more stable versions as they were released. The final Docker version used was 17.05-ce.

6.2 Programming Environment

Programming was done in a Linux environment running Ubuntu 16.04. Most of the code was written in Python, YAML and bash. The written software was mainly used as a proof of concept and for testing and evaluation purposes.

6.3 Cancer Analysis Workflow

The goal of CAW is to detect somatic variants from WGS data. It requires 16 samples in total as input, 8 tumours and 8 normal cells. The workflow starts by aligning the samples using Burrows-Wheeler alignment tool (BWA) [31], it then merges all the samples into one bam-file per group using Samtools [32], duplicates are marked, realignment of known single nucleotide polymorphisms (SNPs) and finally recalibration of reads.

It uses Nextflow [33] which is a domain specific language for building workflows. The total number of third-party software required to run the workflow is 13. Each software is available in pre-packaged containers [34] made by the authors so each step can easily be reproduced by using Nextflow and Docker. However, in this study a more monolithic approach was used in which a persistent storage (volume) was created where all the dependencies were installed.

6.4 KubeNow

KubeNow [35] (https://github.com/kubenow/KubeNow) is a cloud agnostic platform used for deploying a Kubernetes cluster using provisioning tools such as Terraform [36] and Ansible [37] on a cloud providers IaaS. It was developed within the frame of EU project PhenoMeNal (http://phenomenal-h2020.eu). The platform aims to simplify a Kubernetes deployment and it currently works on Amazon Web Services (AWS), OpenStack and Google Cloud Engine (GCE).

6.5 Google Cloud Cluster

A cluster on GCE was used for testing purposes with different machine types. GCE provides a wide range of different machine types but in this study mainly three were used, the standard, high memory and high CPU types with a different number of virtual CPUs (vCPU). Googles cloud engine has native support for deploying a Kubernetes cluster. This service is the Google Container Engine (GKE), which provisions and deploys the Kubernetes cluster in a matter of minutes. Unlike KubeNow however, this is not an agnostic solution.

7 Implementation

This part of the report will go into more details of how and what kind of software was written to test if Kubernetes is a viable option to approach bioinformatic problems. To get a better understanding of how Kubernetes works in practice, this part will include some of the code written. Most of the code was written in Python using the Flask [38] framework. The Python code will not be described in depth, instead the focus will be to describe the parts that are relevant to Kubernetes and Docker, however the repository with all the source code is available at https://github.com/firog/scalableCloudCommunity.

7.1 Web Application

The main goals for the application was to enable easy deployment, testing Kubernetes functions and that it should work out of the box. It was built using Python, Flask and Bash consisting of a backend and a fronted with the purpose to easily start the cancer analysis workflow. As seen in Figure 6 the Kubernetes components are not included.

A user gains access to the application functions via a web browser and the interesting function is the one that creates the CAW job, however in this application described in Figure 6 the Kubernetes components are not utilized. The web application shows how flexible Kubernetes can be to fit almost any purpose.

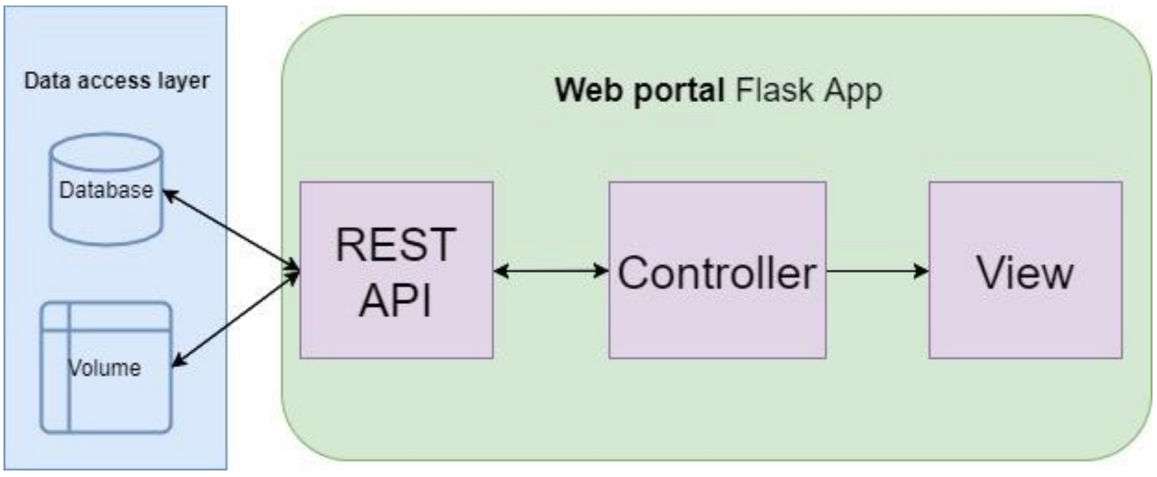
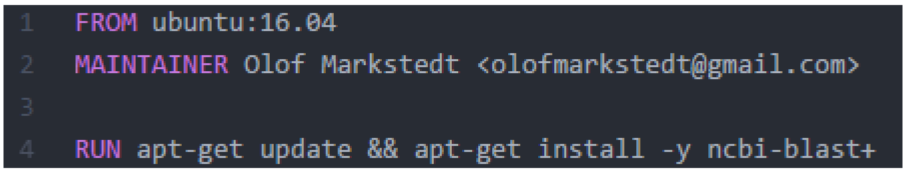


Figure 6: A basic architectural diagram of the web applications components. The basic architecture of the web application. The interaction between the user and application occurs via the web browser using the URL or external-IP pointing to the application. The database in the data access layer holds information of individual users submitting jobs. The

volume holds the dependencies for the cancer analysis workflow used by Kubernetes jobs. The REST API receives and sends HTTP request to and from the data access layer and the controller. The view is accessed using a web browser and the controller serves data to the view. From the view, different jobs can be submitted for example the cancer analysis workflow.

7.2 Containerizing Applications

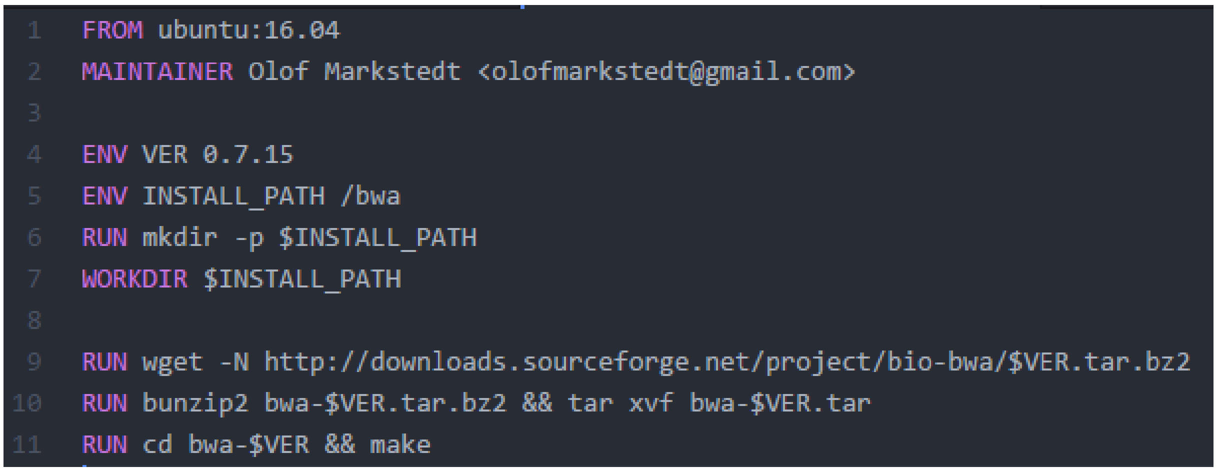
There were several Docker images created in this thesis and only the most interesting aspects of the ones created will be covered. A Dockerfile should typically be simplified as much as possible. For example, Figure 7 describes a very simple Dockerfile that creates an image which installs the BLAST [39] software, this image uses the Ubuntu package manager APT. However, if another base image was used such as Fedora, YUM would replace APT and this applies to all other operating systems with different package managers. This allows for great flexibility when it comes to life science tools, basically any tool can be containerized and the Kubernetes cluster will allow for these to run with no operating system restriction.



```
FROM ubuntu:16.04
MAINTAINER Olof Markstedt <olofmarkstedt@gmail.com>

RUN apt-get update && apt-get install -y ncbi-blast+
```

Figure 7: A basic Dockerfile for creating a Docker image with the BLAST software. The first row tells Docker to pull down a base image that the new image will be based on. The creators name can be specified but is not necessary. The last row tells Docker to run the commands given after the "RUN" keyword. This specific Dockerfile will create an image that has all the basic BLAST components.

Installing and compiling from source is also possible in cases where the software is not available in the Linux core library. In Figure 8 Docker installs and compiles BWA [31] from source. All the steps in Figure 8 are identical to how the software would be installed on a local machine using bash, the only difference being the keywords are not included.



```
FROM ubuntu:16.04
MAINTAINER Olof Markstedt <olofmarkstedt@gmail.com>

ENV VER 0.7.15
ENV INSTALL_PATH /bwa
RUN mkdir -p $INSTALL_PATH
WORKDIR $INSTALL_PATH

RUN wget -N http://downloads.sourceforge.net/project/bio-bwa/$VER.tar.bz2
RUN bunzip2 bwa-$VER.tar.bz2 && tar xvf bwa-$VER.tar
RUN cd bwa-$VER && make
```

Figure 8: A Dockerfile with all necessary steps to install BWA. The “ENV” keyword creates an environment variable called “VER” with its value set to “0.7.15”, an installation path is created and the source code for BWA is downloaded into that directory where it is then compiled.

In cases where third-party software is used, it can be difficult and tedious to know which the essential dependencies are and using Ubuntu as a base image is not always very efficient. In cases where third-party software is not used, a minimalistic base image can be used. For example Alpine [40] is a Linux distribution with a base image of size ~5Mb.

The Docker image for the web application in this thesis uses Python Alpine as the base image seen in Figure 9. It creates the path where the source code will be located and copies all the code from the machine running Docker to the Docker image. Once it has installed all the requirements it will run a bash script which starts the server. The last row after “CMD” is not necessary when using Kubernetes, when creating the pods the initialization script can be started by giving a command in the podspec. This can be seen in Figure 12 where a command is given in the deployment manifest.

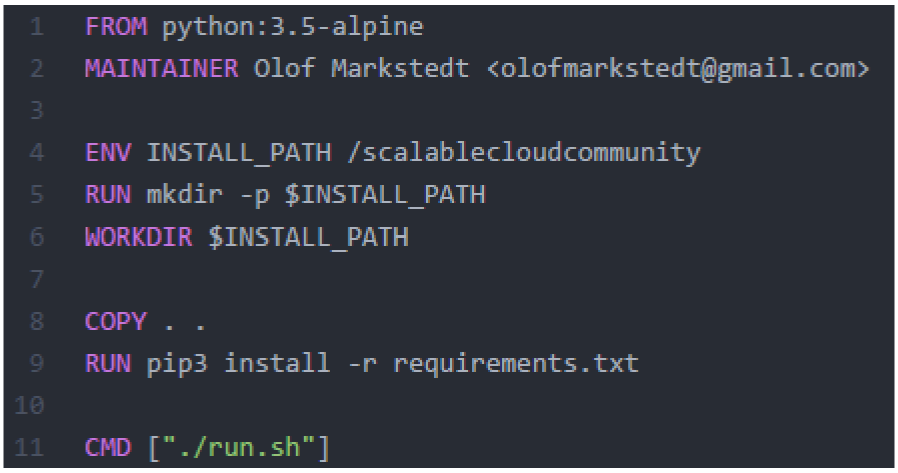

```
FROM python:3.5-alpine
MAINTAINER Olof Markstedt <olofmarkstedt@gmail.com>

ENV INSTALL_PATH /scalablecloudcommunity
RUN mkdir -p $INSTALL_PATH
WORKDIR $INSTALL_PATH

COPY . .
RUN pip3 install -r requirements.txt

CMD ["./run.sh"]
```

Figure 9: The Dockerfile creating the containerized web application. The image uses Python Alpine as the base image (~5Mb in size), Docker will create a folder inside the container and copy all the source code from the local machine to the container, then install all the Python dependencies and finally start the server.

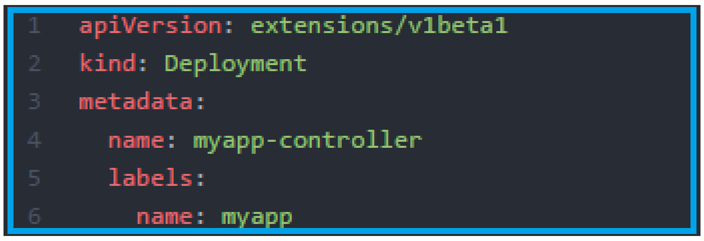
The container running the CAW is an Ubuntu container with the Java development kit. Since all the dependencies resided in a volume which during runtime was attached to the pod running the CAW container, all the dependencies were not required to be containerized. Due to the large dependency size (~30Gb) the workflow requires, the volume was used as a static source for the container to read from during runtime as opposed to creating a container of size 30Gb every time the workflow was used. Bear in mind that the Kubernetes components are all based on custom containers created in the same manner as the ones described in this section.

7.3 Kubernetes Components

The components were created using the kubectl CLI. Replication controllers, services, volumes, jobs and deployments were created. Many of the templates are very similar, but at least one of each used component will be covered. The deployment component works much like the replication controller; thus, it will be covered in the replication controller section.

7.3.1 Replication Controller

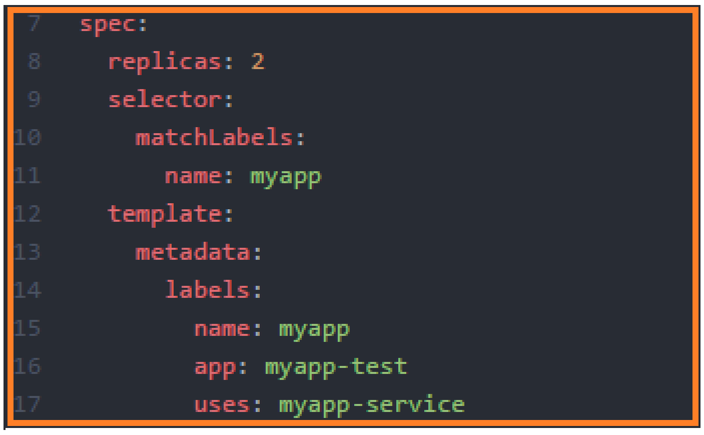
Figure 10 describes the first part of the deployment manifest for the web frontend. The manifest is divided into three parts. The first two rows are mandatory in every manifest. The API version is set to the value in which the specific Kubernetes component exists, followed by the type (kind) of component. In this case the component is "Deployment". Every component is given some set of metadata, i.e. the name of the component and the labels.



```
1  apiVersion: extensions/v1beta1
2  kind: Deployment
3  metadata:
4    name: myapp-controller
5    labels:
6      name: myapp
```

Figure 10: First part of the deployment manifest. Row 1-6, are instructions telling Kubernetes how the component should be treated and where it should be created from.

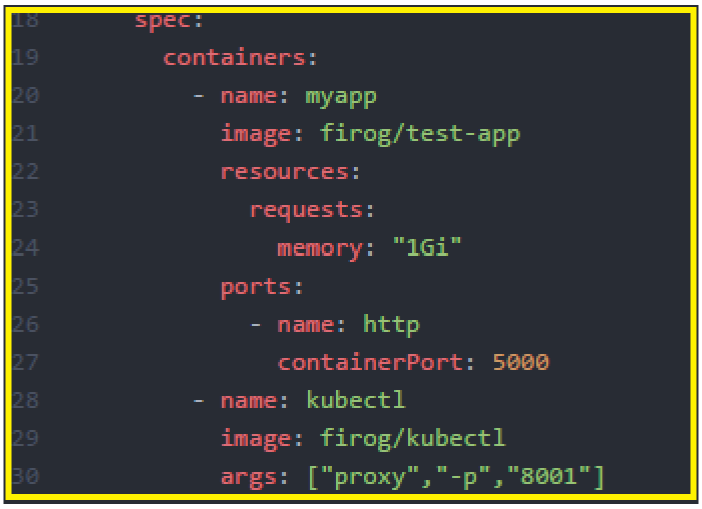
The most important part of this manifest is the "spec" key, in this manifest it starts at row 7 as seen in Figure 11. These are the instructions given to the deployment (or replication controller) on the behaviour of the pods. It describes how many replicas of the pod that should run, tells the deployment what the labels of the target service is and gives the pods some set of key value pair labels (name, app, uses) that is targeted by the service. The components in Kubernetes can have any number of labels with any value.



```
7  spec:
8    replicas: 2
9    selector:
10     matchLabels:
11       name: myapp
12   template:
13     metadata:
14       labels:
15         name: myapp
16         app: myapp-test
17         uses: myapp-service
```

Figure 11: Second part of the deployment manifest. Row 7-17, are instructions to the deployment component. These tell the component how the pods will be recognized by other components (such as services).

The second "spec" key can be seen in Figure 12 on row 18 is the pod template. This key holds the information about which containers that should run in the pod. The first container called "myapp" is the web portal both backend and frontend. It uses the Docker image "firog/test-app" that is available on the dockerhub. The container requests 1Gb of memory to be allocated for that specific container and that it should listen to requests on port 5000.



```
18      spec:
19        containers:
20          - name: myapp
21            image: firog/test-app
22            resources:
23              requests:
24                memory: "1Gi"
25            ports:
26              - name: http
27                containerPort: 5000
28          - name: kubectl
29            image: firog/kubectl
30            args: ["proxy","-p","8001"]
```

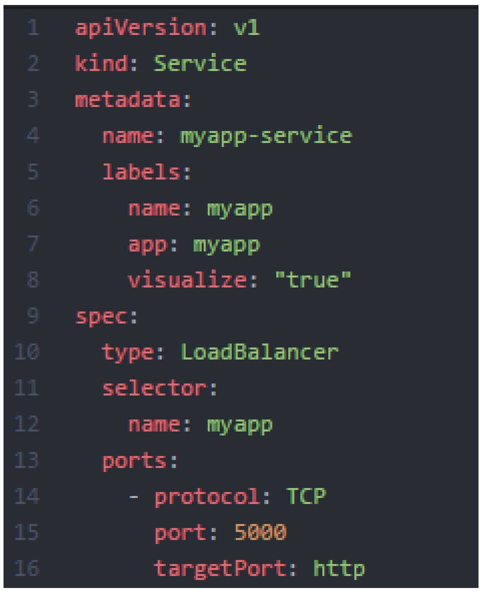
Figure 12: Third and last part of the deployment manifest. Row 18-30, are instructions given to the pods. This instructs the pods which containers to deploy.

The second container uses the image “firog/kubectl” which is a custom image created with the purpose of proxying the kube-api server from the same pod to the node it runs on. This allows for Kubernetes operations to be done within the web portal using the Kubernetes Python client library, e.g. start a Kubernetes job. This allows for users to create jobs without having to create a new job manifest.

7.3.2 Service

The services differ from a replication controller/deployment when it comes to the template and it does not create any pods. Figure 13 is a service manifest and the first 7 rows look a lot like that of Figure 10. The purpose of this service is to create a load balancer which will route traffic from the web to the pods created by the deployment. In the “spec” key, a load balancer type is specified. This type of load balancing only works if the cloud provider has an external load balancing service, such is the case of GCE. The load balancer endpoint will be exposed on an external IP which can be seen in Figure 4.

The selector will target the labels on the pods created by the deployment, thus it will know which pods to route the traffic to. If the labels and selectors does not match it will simply not do anything. A Kubernetes service has TCP and UDP support, TCP being the default protocol. The service is instructed which port to listen to (5000) and what the name of the target port is. However not all cloud providers or clusters provide an external load balancing service. For this purpose, KubeNow was used. KubeNow creates a Kubernetes cluster on AWS, GCE or OpenStack using existing orchestration tools and is rather straightforward. It is not required that a service is given a type. In cases where the components in the cluster need to communicate with other components within the cluster a simple service will suffice without a specific type. For example, consider that a database is running within the cloud in its own separate pod. To be able to communicate with this pod a service that targets this database pod must be created, otherwise the only component that will be able to communicate with the database will be the containers running inside the pod (if the pod has multiple containers running).

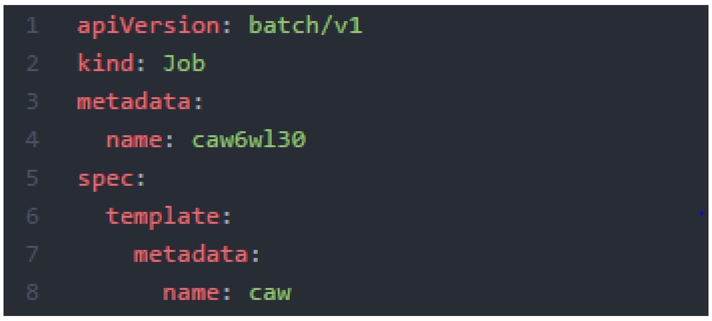


```yaml
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: myapp-service
5    labels:
6      name: myapp
7      app: myapp
8      visualize: "true"
9  spec:
10   type: LoadBalancer
11   selector:
12     name: myapp
13   ports:
14     - protocol: TCP
15       port: 5000
16       targetPort: http
```

Figure 13: A Kubernetes service manifest. The first two rows are instructions for Kubernetes about where to find the component (API version v1) and what kind of component (service). Rows 3-7 is metadata given to the service, i.e. the name of the service and the labels. The spec contains the specification of the service.

7.3.3 Job

The job component is the most interesting one for life science research. The template for the job is divided into three parts for easier explanation. Figure 14 describes the first part of the job manifest, as any other component template the API version and kind is specified with some metadata. In the first “spec” key a template key is given which includes a second “spec” key with more data as seen in Figure 15.

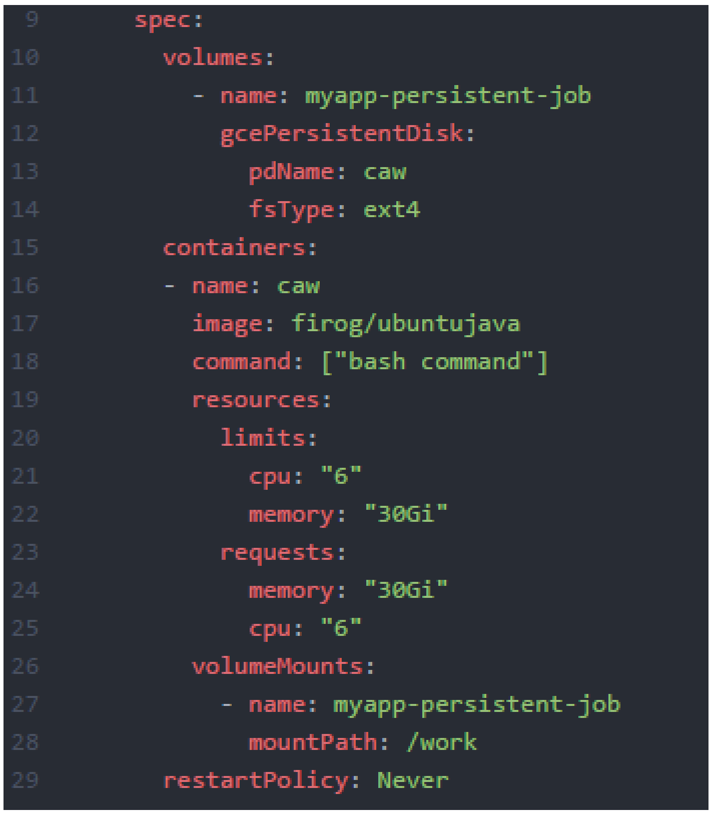


```yaml
1  apiVersion: batch/v1
2  kind: Job
3  metadata:
4    name: caw6wl30
5  spec:
6    template:
7      metadata:
8        name: caw
```

Figure 14: First part of the job manifest. The API version where the job component is located is given as a value. The name of the job, “caw6wl30”. Followed by the spec with the rest of the templated.

The job manifest continues with a volume attachment on row 10 seen in Figure 15. This is the pre-created volume that holds the dependencies for the CAW. Note that the key where the name of the volume is specified is called “volumes”, so again several volumes can be mounted to a pod. The manifest specifies the reference to the volume (“myapp-persistent-job”), the type of persistent storage which in this case is a GCE persistent disk. The name given to the disk after creation and the file system type (ext4).

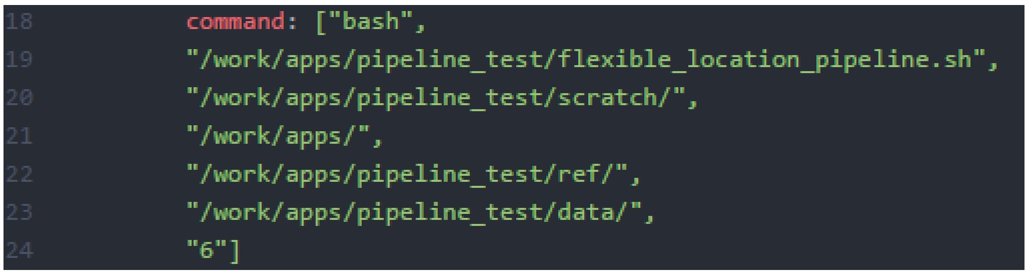
The next part is to specify some container information. The name of the container, what image to use (firog/ubuntujava), the command that the container within the pod should execute once it has been created (described in Figure 16) and some resource requests and limitations. The number of CPUs can be requested as one entire core or milicores, in Figure 15, six cores are requested. Same goes for memory which can be requested in any byte size, e.g. 1024Mi is equivalent to 1024 megabytes. In this case the resource limitation and requests are the same. This is because of the nature of bash. When the bash script executes it will pool all the resources it can utilizing more memory and CPU than requested which induces some problems. It will saturate the resources available in an unpredictable way and the limitations will force the bash script within the container to only allow for the usage of the requested amount of resources.



```
 9      spec:
10        volumes:
11          - name: myapp-persistent-job
12            gcePersistentDisk:
13              pdName: caw
14              fsType: ext4
15        containers:
16        - name: caw
17          image: firog/ubuntujava
18          command: ["bash command"]
19          resources:
20            limits:
21              cpu: "6"
22              memory: "30Gi"
23            requests:
24              memory: "30Gi"
25              cpu: "6"
26          volumeMounts:
27            - name: myapp-persistent-job
28              mountPath: /work
29        restartPolicy: Never
```

Figure 15: Second part of the job manifest. The name of the volume is specified. The type of volume “gcePersistenDisk” followed by the disk name and the file system type. The container specification is given, the bash command to initialize the job which in this figure is truncated (see **Figure 16** for the command). The resource and limit requests are: six cores and 30Gb of RAM. The key on row 26 specifies what the name of the mount path is. The restartPolicy key tells Kubernetes to never restart the job in case of failure.

The long bash command in Figure 16 initializes the workflow. The bash script is located in the volume mounted to the pod of the job in “/work” which was the name given to the mount path seen on row 28 in Figure 15. The first parameter is the location of the bash script followed by the second parameter, the work directory in which the results will be stored. The third parameter is the location of all the dependencies. The fourth are the reference genomes and the fifth is the input data to run the workflow on. The last parameter is the number of threads that each tool supporting threading within the workflow should use. For example, the workflow uses tools such as BWA [31] and Picard Tools [41] which supports threading.



```
command: ["bash",
"/work/apps/pipeline_test/flexible_location_pipeline.sh",
"/work/apps/pipeline_test/scratch/",
"/work/apps/",
"/work/apps/pipeline_test/ref/",
"/work/apps/pipeline_test/data/",
"6"]
```

Figure 16: Third part of the job manifest including the bash command. The files are in the mount path "/work". The command starts by executing the script which takes some parameters. The "scratch" directory is the work directory where the workflow will store the results. The "/work/apps" parameter points to the workflow dependencies. The "/ref" directory contains the reference files of tumour and non-tumour genomes, followed by the data to analyse. The last parameter is the number of threads for the workflow to use.

While this thesis only examines the CAW use case, any workflow can be containerized and executed in a Kubernetes cluster using the job component. As CAW has a large dependency size which poses problems, it is a good example of how such a problem can be solved using volumes together with Docker and Kubernetes. The widely-used BLAST [39] tool can run with all the dependencies within a container and would very much work the same way as the job component for the CAW. The only difference being the containerization process and the command to initialize the algorithm and the data to analyse.

7.4 Cluster Creation

The cluster creation was done in two separate ways. Two machine types were tested to run the workflow using a different number of CPUs. There are several machine types that GCE provides but the ones listed in Table 1 are the most suitable ones for the purpose of this thesis. The descriptions of the different machine types can be found in Table 1. The X in the machine type names is the number of vCPUs. It can be any multiplier of two where the maximum number is 64 and the memory of the machine type is X*Memory/vCPU. This is only a restriction during cluster creation, a job component can have access and utilize 11 cores for instance. For example, n1-highmem-4 has four cores with a total of 26Gb of memory. The differences between the two machine types is the memory size. The n1-highmem-X type is a high memory machine type suited for work that requires more memory. The n1-highcpu-X type is a machine type suited for work that is more CPU intensive and does not required too much memory. The workflow was tested on n1-highcpu-X and n1-highmem-X up to a total of 12 cores with a different amount of memory limitations ranging from 10Gb to 70Gb.

In order to compare the job components scalability, Celery [42], an asynchronous task queue was also tested for the purpose of performance measurements. Celery uses the concept of tasks, where each task is basically a job sent to the Celery worker. A task will execute concurrently on worker nodes but lacks some of the benefits of Kubernetes jobs. It scales horizontally and uses distributed messaging for task execution. The Celery task will execute in the same pod as the web application or on a different pod controlled by a separate replication controller.

Table 1: The different machine types used with description. The machine types are listed in the first column with their respective memory/CPU in the second. The n1-highmem-X type is ideal for tasks requiring more memory relative to CPU and the n1-highcpu-X type is ideal for tasks that are more CPU intensive relative to memory usage. The X in the machine type name determines the number of CPUs for each machine.

Machine Type	Memory/CPU (Gb)
n1-highmem-X	6.50
n1-highcpu-X	0.90

GCE has support for rapid and easy cluster deployment using Google Container Engine. With a simple bash command the cluster will be up and running within minutes and all the Kubernetes cluster building blocks will be ready, it simply works out of the box. The cluster was created using the GCE CLI. Cluster creation was also done using KubeNow [35], the final cluster of KubeNow works very much like the final cluster of GKE in terms of running pods and executing jobs.

The architecture of the implemented cluster can be seen in Figure 17. It is a single node cluster running the web application with a view for submitting CAW jobs. Due to the cost of executing tasks and GCEs CPU and memory limitations only a single node was used.

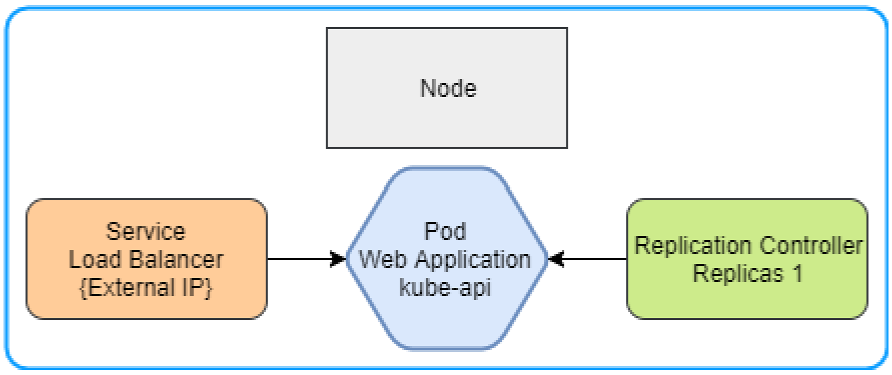


Figure 17: The architecture of the implemented test cluster for running jobs.

In Figure 18 a job is created either using the kubectl CLI or via the web application. This spawns a second pod that will execute the job independently from the other components in the cluster. The volume will be attached as described in Figure 15 and once the pod is created, the workflow will start running. The execution time of the workflow was logged for every test run. The purpose of this was to evaluate whether the it was more CPU intensive or memory intensive.

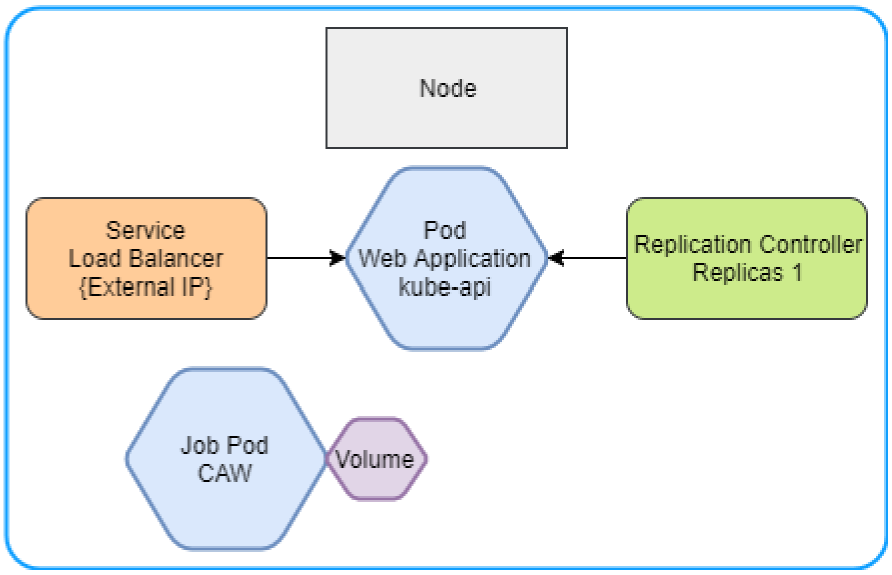


Figure 18: The running job component in the Kubernetes cluster. The pod will spawn within the cluster on any available node with the available resources. Once the container has been pulled and created it will start the command given in the template.

8 Results

The Cancer Analysis Workflow takes approximately 25 minutes to run on a laptop with 4 cores clocked at 2.40GHz and 12Gb of memory. The dataset used for testing was only a subset of the complete dataset for easier benchmarking. The Kubernetes Job proved to be a robust component to use for running life science tasks. As expected the CAW is highly memory intensive as most life science workflows are [5]. In Figure 19 is the performance of the job component with different resource allocation. As expected an increase in number of CPUs and amount of memory given to the job component the execution time decreases. The time it takes for the job to start the first time is approximately 30 seconds which is not considered when doing the measurements, after the first deployment it is much faster (~10 seconds) because the container has already been pulled.

The time it takes for the job to finish reaches little over seven minutes at best but is then saturated. As described in Figure 20 this probably has to do with the small dataset used. A big increase in performance is seen when comparing the machine with two cores vs the one with eight cores and above. It is almost a twofold decrease in time, now given that a large dataset might take days to run, a twofold decrease in execution time is good.

Many Kubernetes jobs can be initialized or sent to the cluster. Each job will try to allocate the resources demanded by the user in the template and if the resources are insufficient, the job will be waiting in a pending mode. This is a reliable way for many users to create and run jobs in the same cluster without the need for connecting remotely to the cluster via SSH or a similar technology, allocating the resources and then start the job. The “queuing” system of a

Kubernetes job works very well and it is easy to print the log of a specific job. Kubernetes stores everything from STDOUT to a log file available to all that has access to that cluster or namespace.

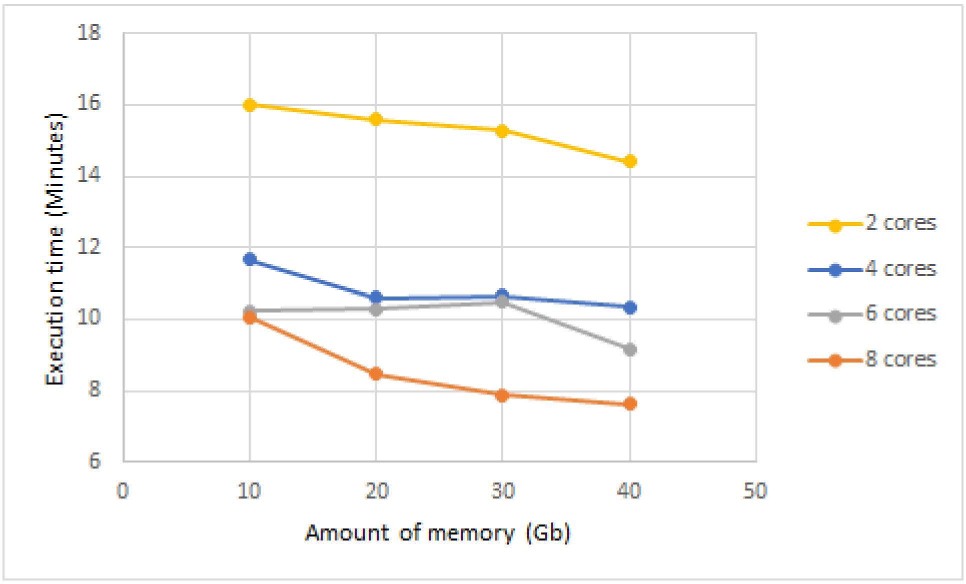


Figure 19: Workflow performance of a Kubernetes job measured in minutes with different machine setups. Each line differentiates in the number of cores. The x-axis is the amount of RAM-memory used for that measurement and the y-axis is the execution time rounded up to minutes. The executions times were measured in the same cluster. The job was given different amount of resource allocation and limitations. The general trend being that the execution time decreases as the resources increases which is expected.

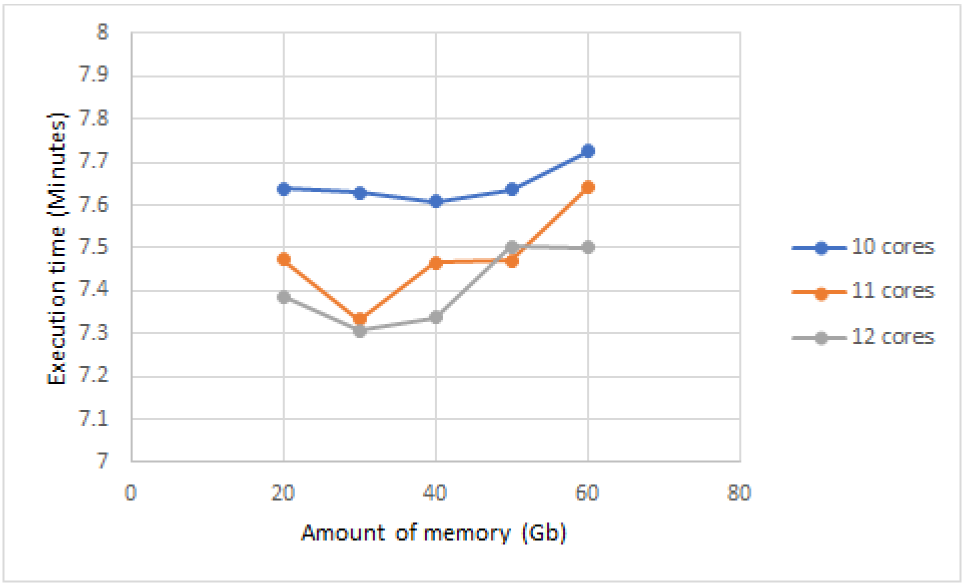


Figure 20: Workflow performance of a Kubernetes job measured in minutes with another set of machine setups. As can be seen the increase in performance levels out fast and rather gets a decrease in performance. This has most likely to do with the small dataset used for measurement.

In Figure 21 are the results of the asynchronous Celery worker next to their respective Kubernetes job component. Some machine types were only tested because Celery does not allow for resource limitations when a task is executing. The Celery worker will much like a bash script, pool all the available resources during execution. Therefore, a cluster with one node having the machine setups described in Figure 21 was created for testing purposes. The Celery worker is not as reliable as a Kubernetes job because the resources it uses are not limited to a single asynchronous process. There may be other threads waiting to be executed on the same core, thus the performance is not as reliable as a Kubernetes job. For example, the first staple in Figure 21 is a machine with 2 cores and 10Gb of ram which had an execution time of 35 minutes. Comparing this to a Kubernetes job of the same machine type which took a little under 16 minutes to execute, is a large decrease in performance on the Celery machine. When the number of cores and amount of memory increases, the difference between the two becomes smaller. This has most likely to do with the dataset used by the workflow being rather small and thus as mentioned earlier saturated.

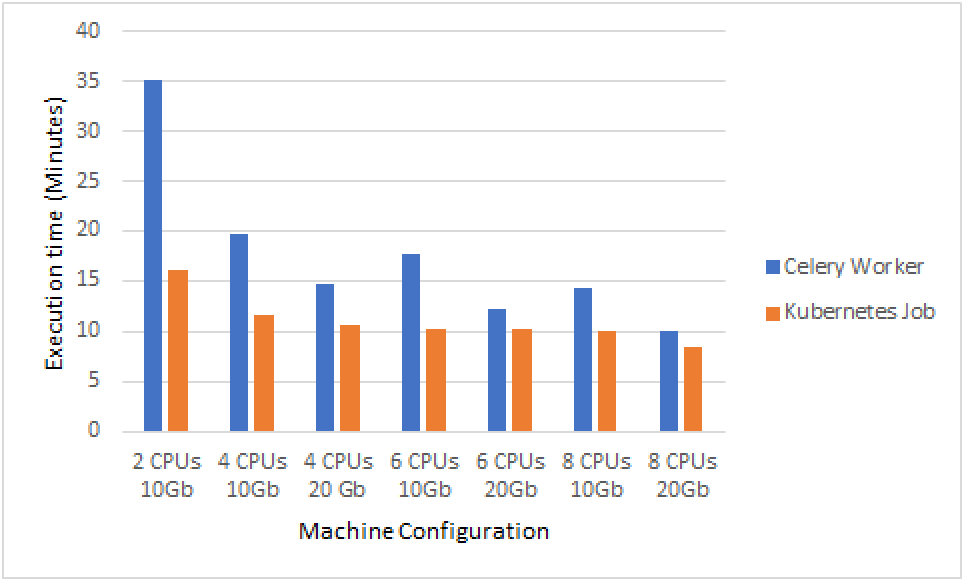


Figure 21: Performance of asynchronous Celery worker vs Kubernetes Jobs. The workflow was executed on a celery worker with a different machine setup. Each staple group represents a unique setup on the x-axis. The execution time is on the y-axis. The scalability of a Kubernetes job is superior to the scalability of Celery.

9 Discussion

9.1 Kubernetes and Life Science Research

Kubernetes is designed to be highly scalable and works well in small clusters with a few containers running and large clusters with several thousand or even million containers running. KubeNow and GKE allowed for easy and rapid cluster deployment, as well as allowing the user to create clusters of different flavours. Since Kubernetes can run on almost anything and is highly customizable, it makes for a viable option when conducting life science research. Every component can basically be customized to fit the purpose of the user and the container technology is as well highly customizable. Kubernetes will not replace the tasks required to run in high performance computing (HPC) environments, instead by utilizing existing HPC environments with all the benefits of it and coupled with the benefits of containers it will be able to increase productivity and make reproducibility trivial when it comes to life science research. There will be no need for installing new software nor update tools related to life science research in the HPC environment, instead, the software can simply be containerized on the user's local computer and then used within the HPC environment. Therefore, the software does not have to be installed on the clusters. The job component allows for easy execution of workflows with different tool setups. For example, if a specific version of a tool is required that is not installed in the environment or to mix different tools. The same tool can run in separate jobs with different versions as well.

The Kubernetes technology is suitable for several types of life science research. The main use case seems to be running heavy workload workflows due to the nature of the job component working like a batch job with more control over the resources. Another use case would be to group orthologous protein sequences for several different species or bacteria strains to determine the phylogenetic placement of a novel strain. There are a number of steps to do this and many tools are available such as OrthoMCL [43]. OrthoMCL uses a relational database for the analysis and the different steps in the workflow are similar to that of a typical life science workflow, many of the steps are very compute intensive (CPU and RAM dependent). One of the steps is an all to all BLAST that generally takes days to run (depending on the dataset). Another is pairing the proteins which also is a resource demanding step. Other use cases would be for typical *de novo* assembly using e.g. Abyss [44] or Velvet [45].

However, Kubernetes is not suitable for all types of life science research. There are some cases where it would be excessive. For example, if the number of users are low, it would be excessive to install and configure a Kubernetes cluster only to run simple jobs that can run locally or when jobs aren't executed very often. This would only lead to wasted CPU cycles, causing the allocated resources to be idle. Another case would be if part of the analysis requires visualization tools it is more suitable to that sort of analysis on a local computer. Let Kubernetes do the heavy lifting prior to the visualization step.

While this thesis only touches on a few examples on how container technology and Kubernetes can be used in life science research, more and more bioinformaticians has started to adopt the container technologies for easier reproducibility and increase productivity. The result of this is an increase in the amount of life science images available to the public. For instance, there is a project for the web-based bioinformatics workflow platform Galaxy [46] with the purpose to run Galaxy on Kubernetes [47]. There is another project [48] with the purpose to integrate native Spark [49] in Kubernetes and use the Kubernetes scheduler to manage the Spark cluster. As Spark and Galaxy are both frameworks widely used in the bioinformatics community, Kubernetes will gain an increase in popularity over time within the field of life science research. There are some problems however with Docker for example which requires root access to run which most users don’t have access to in a HPC cluster.

9.2 Programmatic Kubernetes

The learning curve of Kubernetes is quite steep, not only are the abstractions many but to be able to use Kubernetes in an efficient way one must learn and understand how to use container technology as well. While there are some client libraries for Kubernetes, they often lack thoroughly written documentation. Since Kubernetes is written in the GO language, the GO client library documentation is better written than the documentation for Python. But again, since Kubernetes is such a new project this will most likely get better.

10 Conclusions

Kubernetes and the container technology has proven to good tools for conducting life science research. The examples tested in this thesis have shown that the technologies are well fitted for some of the problems bioinformaticians face today. The examples are indeed very few, but most life science workflows are based on open-source projects that can further be used to create custom container images and applications to use with Kubernetes which has shown to be efficient, reliable and scales well. Most workflows follow the same pattern as CAW, in which each step in the workflow is a third-party software. As mentioned earlier, the number of NGS related projects have been increasing rapidly. More users lead to an increase of inexperienced users; therefore, the resources will most likely not be utilized very efficiently. In HPC environments such as UPPMAX, the number of installed life science software with different versions will increase. By using containers, the software will not have to be installed in the environment but rather be containerized and still be reusable by other users.

Kubernetes provides several services that are beneficial for conducting life science research. It provides for a good architecture to build on top of because of its highly customizable nature. The components can be used in many other different ways than described in this thesis to fit the users purpose. It is backed up by a large community and has started to gain popularity among bioinformaticians. The job component is a good example of how workflows can be executed using a declarative template which has a very concrete syntax.

Even though Kubernetes has proven to be a very useful tool when conducting life science research, it comes with some pitfalls. It can be very troublesome at first to get accustomed to the declarative way of creating the components. Even if a Kubernetes cluster is available to a user, the user still has to know how to create containers and the templates for Kubernetes. While most job templates look very much the same, it can still be tweaked a lot and there are many features in each component that can be added and changed. The goal for the project has mainly been to evaluate and illustrate how Kubernetes and containers can be used in the field of life science research, it has proven to be a suitable tool for this purpose.

References

[1] *CAW: Cancer Analysis Workflow: a complete open source pipeline to detect somatic variants from WGS data*. Science For Life Laboratory, 2017.

[2] P. Tatlow and S. R. Piccolo, ‘A cloud-based workflow to quantify transcript-expression levels in public cancer compendia’, *Sci. Rep.*, vol. 6, Dec. 2016.

[3] P. Di Tommaso, E. Palumbo, M. Chatzou, P. Prieto, M. L. Heuer, and C. Notredame, ‘The impact of Docker containers on the performance of genomic pipelines’, *PeerJ*, vol. 3, p. e1273, 2015.

[4] B. Howe, ‘Virtual Appliances, Cloud Computing, and Reproducible Research’, *Comput. Sci. Eng.*, vol. 14, no. 4, pp. 36–41, Jul. 2012.

[5] M. Dahlö, D. Scofield G, W. Schaal, and O. Spjuth, ‘Tracking the NGS revolution: life science research on shared high-performance computing clusters’, In press.

[6] ‘Docker - Build, Ship, and Run Any App, Anywhere’. [Online]. Available: https://www.docker.com/. [Accessed: 05-Jul-2017].

[7] C. Boettiger, ‘An Introduction to Docker for Reproducible Research’, *SIGOPS Oper Syst Rev*, vol. 49, no. 1, pp. 71–79, Jan. 2015.

[8] ‘Kubernetes’, *Kubernetes*. [Online]. Available: https://kubernetes.io/. [Accessed: 05-Jul-2017].

[9] W. L. Schulz, T. J. S. Durant, A. J. Siddon, and R. Torres, ‘Use of application containers and workflows for genomic data analysis’, *J. Pathol. Inform.*, vol. 7, Dec. 2016.

[10] P. Di Tommaso, M. Chatzou, E. W. Floden, P. P. Barja, E. Palumbo, and C. Notredame, ‘Nextflow enables reproducible computational workflows’, *Nat. Biotechnol.*, vol. 35, no. 4, pp. 316–319, Apr. 2017.

[11] S. Dustdar, ‘Cloud Computing’, *Computer*, vol. 49, no. 2, pp. 12–13, Feb. 2016.

[12] N. Patrignani and I. Kavathatzopoulos, ‘Cloud Computing: The Ultimate Step Towards the Virtual Enterprise?’, *SIGCAS Comput Soc*, vol. 45, no. 3, pp. 68–72, Jan. 2016.

[13] V. Rajaraman, ‘Cloud computing’, *Resonance*, vol. 19, no. 3, pp. 242–258, Mar. 2014.

[14] G. A. Lewis, ‘Cloud Computing’, *Computer*, vol. 50, no. 5, pp. 8–9, May 2017.

[15] J. E. Smith and R. Nair, ‘The architecture of virtual machines’, *Computer*, vol. 38, no. 5, pp. 32–38, May 2005.

[16] J. Nocq, M. Celton, P. Gendron, S. Lemieux, and B. T. Wilhelm, ‘Harnessing virtual machines to simplify next-generation DNA sequencing analysis’, *Bioinformatics*, vol. 29, no. 17, pp. 2075–2083, Sep. 2013.

[17] ‘What is a Container’, *Docker*, 29-Jan-2017. [Online]. Available: https://www.docker.com/what-container. [Accessed: 06-Jul-2017].

[18] A. Silver, ‘Software simplified’, *Nat. News*, vol. 546, no. 7656, p. 173, Jun. 2017.

[19] S. Nadgowda, S. Suneja, and A. Kanso, ‘Comparing Scaling Methods for Linux Containers’, in *2017 IEEE International Conference on Cloud Engineering (IC2E)*, 2017, pp. 266–272.

[20] ‘BioContainers’. [Online]. Available: https://biocontainers.pro/. [Accessed: 17-Aug-2017].

[21] ‘Running Kubernetes Locally via Minikube’, *Kubernetes*. [Online]. Available: https://kubernetes.io/docs/getting-started-guides/minikube/. [Accessed: 18-Jul-2017].
[22] ‘Pod Lifecycle’, *Kubernetes*. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/. [Accessed: 06-Jul-2017].
[23] ‘Replication Controller’, *Kubernetes*. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/. [Accessed: 06-Jul-2017].
[24] ‘Deployments’, *Kubernetes*. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. [Accessed: 18-Jul-2017].
[25] ‘Services’, *Kubernetes*. [Online]. Available: https://kubernetes.io/docs/concepts/services-networking/service/. [Accessed: 17-Jul-2017].
[26] ‘Jobs - Run to Completion’, *Kubernetes*. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/. [Accessed: 14-Jul-2017].
[27] ‘Namespaces’, *Kubernetes*. [Online]. Available: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/. [Accessed: 04-Aug-2017].
[28] *client-python: Official Python client library for kubernetes*. Kubernetes Incubator, 2017.
[29] ‘Google Container Engine (GKE) for Docker Containers’, *Google Cloud Platform*. [Online]. Available: https://cloud.google.com/container-engine/. [Accessed: 17-Jul-2017].
[30] ‘Kubernetes now Generally Available on Azure Container Service | Blogg | Microsoft Azure’. [Online]. Available: https://azure.microsoft.com/sv-se/blog/kubernetes-now-generally-available-on-azure-container-service/. [Accessed: 17-Jul-2017].
[31] H. Li and R. Durbin, ‘Fast and accurate short read alignment with Burrows-Wheeler transform’, *Bioinforma. Oxf. Engl.*, vol. 25, no. 14, pp. 1754–1760, Jul. 2009.
[32] H. Li *et al.*, ‘The Sequence Alignment/Map format and SAMtools’, *Bioinforma. Oxf. Engl.*, vol. 25, no. 16, pp. 2078–2079, Aug. 2009.
[33] ‘Nextflow - A DSL for parallel and scalable computational pipelines’. [Online]. Available: https://www.nextflow.io/. [Accessed: 17-Jul-2017].
[34] *CAW-containers: Containers for CAW*. Science For Life Laboratory, 2017.
[35] *KubeNow: Deploy Kubernetes. Now!* kubenow, 2017.
[36] ‘Terraform by HashiCorp’, *Terraform by HashiCorp*. [Online]. Available: https://www.terraform.io/index.html. [Accessed: 18-Jul-2017].
[37] ‘Ansible is Simple IT Automation’. [Online]. Available: https://www.ansible.com/. [Accessed: 18-Jul-2017].
[38] ‘Welcome | Flask (A Python Microframework)’. [Online]. Available: http://flask.pocoo.org/. [Accessed: 21-Jul-2017].
[39] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, ‘Basic local alignment search tool’, *J. Mol. Biol.*, vol. 215, no. 3, pp. 403–410, Oct. 1990.
[40] ‘index | Alpine Linux’. [Online]. Available: https://alpinelinux.org/. [Accessed: 24-Jul-2017].
[41] ‘Picard Tools - By Broad Institute’. [Online]. Available: https://broadinstitute.github.io/picard/. [Accessed: 31-Jul-2017].

[42] ‘Homepage | Celery: Distributed Task Queue’. [Online]. Available: http://www.celeryproject.org/. [Accessed: 03-Aug-2017].
[43] L. Li, C. J. Stoeckert, and D. S. Roos, ‘OrthoMCL: Identification of Ortholog Groups for Eukaryotic Genomes’, *Genome Res.*, vol. 13, no. 9, pp. 2178–2189, Sep. 2003.
[44] S. D. Jackman *et al.*, ‘ABySS 2.0: resource-efficient assembly of large genomes using a Bloom filter’, *Genome Res.*, vol. 27, no. 5, pp. 768–777, May 2017.
[45] D. R. Zerbino and E. Birney, ‘Velvet: Algorithms for de novo short read assembly using de Bruijn graphs’, *Genome Res.*, vol. 18, no. 5, pp. 821–829, May 2008.
[46] ‘Galaxy Community Hub’. [Online]. Available: https://galaxyproject.org/. [Accessed: 04-Aug-2017].
[47] *galaxy: Data intensive science for everyone*. Galaxy Project, 2017.
[48] *spark: Apache Spark enhanced with native Kubernetes scheduler back-end*. Apache Spark on Kubernetes, 2017.
[49] ‘Apache Spark^{TM} - Lightning-Fast Cluster Computing’. [Online]. Available: https://spark.apache.org/. [Accessed: 05-Aug-2017].