UNIVERSITY OF AGDER

Queuing system for containerized processing

Author
Øystein Andreassen
Bendik Egenes Dyrli

Supervisor
Sigurd Kristian Brinch

Bachelor Thesis for DAT304

Faculty of Engineering and Science
University of Agder
Grimstad, Spring 2018

Group
Gruppe24

Status
Final

Keywords
Containerization, Queue system, Docker, Kubernetes

Obligatorisk egenerklæring/gruppeerklæring

Den enkelte student er selv ansvarlig for å sette seg inn i hva som er lovlige hjelpemidler, retningslinjer for bruk av disse og regler om kildebruk. Erklæringen skal bevisstgjøre studentene på deres ansvar og hvilke konsekvenser fusk kan medføre. Manglende erklæring fritar ikke studentene fra sitt ansvar.

1.	**Jeg/vi erklærer herved at min/vår besvarelse er mitt/vårt eget arbeid, og at jeg/vi ikke har brukt andre kilder eller har mottatt annen hjelp enn det som er nevnt i besvarelsen.**	☒
2.	**Jeg/vi erklærer videre at denne besvarelsen:** - ikke har vært brukt til annen eksamen ved annen avdeling/universitet/høgskole innenlands eller utenlands. - ikke refererer til andres arbeid uten at det er oppgitt. - ikke refererer til eget tidligere arbeid uten at det er oppgitt. - har alle referansene oppgitt i litteraturlisten. - ikke er en kopi, duplikat eller avskrift av andres arbeid eller besvarelse.	☒
3.	**Jeg/vi er kjent med at brudd på ovennevnte er å betrakte som fusk og kan medføre annullering av eksamen og utestengelse fra universiteter og høgskoler i Norge, jf. Universitets- og høgskoleloven §§4-7 og 4-8 og Forskrift om eksamen** §§ 31.	☒
4.	**Jeg/vi er kjent med at alle innleverte oppgaver kan bli plagiatkontrollert.**	☒
5.	**Jeg/vi er kjent med at Universitetet i Agder vil behandle alle saker hvor det forligger mistanke om fusk etter høgskolens retningslinjer for behandling av saker om fusk.**	☒
6.	**Jeg/vi har satt oss inn i regler og retningslinjer i bruk av kilder og referanser på biblioteket sine nettsider.**	☒

Publiseringsavtale

Fullmakt til elektronisk publisering av oppgaven

Forfatter(ne) har opphavsrett til oppgaven. Det betyr blant annet enerett til å gjøre verket tilgjengelig for allmennheten (Åndsverkloven. §2).
Alle oppgaver som fyller kriteriene vil bli registrert og publisert i Brage Aura og på UiA sine nettsider med forfatter(ne)s godkjennelse.

Oppgaver som er unntatt offentlighet eller tausehetsbelagt/konfidensiell vil ikke bli publisert.

Jeg/vi gir herved Universitetet i Agder en vederlagsfri rett til å gjøre oppgaven tilgjengelig for elektronisk publisering:	☒**JA**	☐**NEI**
Er oppgaven båndlagt (konfidensiell)? (Båndleggingsavtale må fylles ut)	☐**JA**	☒**NEI**
- Hvis ja: **Kan oppgaven publiseres når båndleggingsperioden er over?**	☒**JA**	☐**NEI**
Er oppgaven unntatt offentlighet? (inneholder taushetsbelagt informasjon. Jfr. Offl. §13/Fvl. §13)	☐**JA**	☒**NEI**

Abstract

This project aims to create a proof of concept for queuing compute requests and distributing these requests in containers in a cluster of physical and virtual worker nodes. We are going to focus on NVIDIA GPU processing by utilizing NVIDIA DGX-1 servers present at University of Agder. The cluster and the platform should be scalable both horizontally and vertically. Through qualitative research on different products and their documentation, Kubernetes was used to handle clustering and distribution of container workloads. We attempted to use Vue.JS to create an easy to use front end, Laravel/PHP to create a middleware, and Golang to perform communication between the queue and the Kubernetes cluster to create and manage containers. Due to time constraints and lack of development experience the system did not get completed. Familiarizing ourselves and testing the Kubernetes platform turned out to take more time than anticipated. The group therefore focused on detailing the model in such a way that it should be possible to implement the components based on this report.

This is a promising avenue to accomplish the original goal of creating a scalable queuing system for compute requests. We believe that the platform will be feasible and scalable by using the described technologies and a microservice perspective. But further research and testing is necessary.

Contents

List of Tables

List of Figures

Acknowledgement

We appreciate the great guidance provided by our supervisor, Sigurd Kristian Brinch. We would also like to thank Bjarte Stien Karlsen from Skattetaten department Grimstad for showing an interest in our project, Kristian Lyngstøl from Redpill Linpro for inviting us to Oslo to take part in a Kubernetes workshop meant for technical crew at The Gathering, and André Lucas Carvalho for lending us a NVIDIA GTX 1070.

Additionally, we would also like to thank Piraveen Perinparajan and Kjell Gunnar Kjeldal for providing feedback on this thesis.

Bendik Egenes Dyrli
Øystein Andreassen

Grimstad, May 2018

Chapter 1

Introduction

1.1 Background

University of Agder (UiA) has an increased focus on the advancement of artificial intelligence (AI) research, and has a goal of being the leading university in this regard. The research and projects are steadily growing in complexity, requiring increased compute power to efficiently move forward. UiA is also invested in Mechatronics Innovation Lab (MIL) which is part of the national infrastructure for pilot testing and experimental development, which requires increasing compute power to efficiently run simulations and deep learning networks.

UiA and its partners has therefore invested in specialized compute servers, NVIDIA DGX-1 [1]. These live as separate servers and are currently limited to running Ubuntu. The current system is based on users having access to one of these servers and then utilize Docker to spawn a small environment where they run their compute jobs. This creates operational overhead, and does not scale properly as more users and servers are added, since distributing the workload across the devices is based on where users have access, and who runs what when.

It was therefore requested that a project would look into some sort of queuing solution that would distribute the workload without increasing the overhead, while still maintaining visibility into the workloads.

1.2 Problem definition

Find a solution that gives students and faculty members alike an easy to use interface to request compute resources both with and without GPU which serves these requests in a timely fashion when resources are available. In addition, there should be an interface for the administrator to monitor the queue of unserved requests, monitor the usage of the compute nodes, and have a centralized administrative environment for the jobs running. The solution should be able to scale vertically and horizontally with low effort, and the solution should be able to handle compute nodes with and without a GPU. The primary usage is the Tensorflow Docker image which comes pre-installed with Jupyter, it is a user request to have this accessible from their desktop. [2]

1.3 Assumptions and constraints

Constraints

- Need to borrow projects/people to simulate real workload to predict usability
- Very new technology and platform so documentation might be limited

Assumptions

- Hardware resources will be available to test parts of the solution

1.4 Project strategy

The project will be performed in three phases; Research, implementation and documentation. The research phase consists of researching tools and technologies to be leveraged in the project, and obtain as much information as possible to be able to make qualified choices later on.

During the implementation phase the group will create a proof of concept design as well as implementing and testing individual parts, with the aim of testing the application as a whole.

Lastly the documentation consists of writing the final report.

1.5 Report structure

- **1. Introduction**
  Defines the intent and scope of the project.
- **2. High-level overview**
  Shows the project from a high level overview giving insight into the overall concept, tools and technologies leveraged.
- **3. Theory**
  Describes the theory behind major tools and technologies that has been leveraged in the solution, as well as giving background on terms that are used further in the report. Familiarity with this chapter is important to understand the design and solution.
- **4. Solution**
  Contains explanation of the intended design
- **5. Discussion**
  A post-implementation discussion on the project as a whole, looking back at choices in a self critical manner, as well as future plans for improvements and further work.
- **6. Conclusion**
  A sum of the chapters, and ultimately answering if the project was a success or not.

1.6 Existing products

The group could not find any products that satisfied the problem statement. As such, existing products focus on alternatives to Kubernetes.

Docker swarm

Docker swarm is a usage mode built into Docker to handle clustered Docker environments. It works in a decentralized design, handling specializations such as manager and worker nodes at runtime. It supports consistency of the environment through monitoring the cluster state and attempts to maintain the desired state by distributing workloads and redeploys containers should a worker node go down. Unfortunately, docker swarm does not support NVIDIA-docker and as such it fails one of the requirements in the problem statement. [3]

OpenShift

OpenShift is a product by Red Hat. It uses docker and Kubernetes and as such has much of the same feature set as Kubernetes. Since Red Hat is building on top of Kubernetes they

are behind on versions. Although OpenShift offers a lot of extra features, it only runs on RHEL and CentOS. Since the current infrastructure given the NVIDIA DGX servers require Ubuntu, we can not guarantee stability and functionality after testing, as such we had to dismiss OpenShift as well. [4]

Chapter 2

High-level overview

This chapter contains a high-level overview of the project, describing functional requirements, and an overview of tools and technologies that will be leveraged. This will be further expanded in the third chapter.

2.1 Functional requirements

The functional requirements are based on the user stories approach from Agile. The project is considered a success when the below stories are met:

A user must be able to...

- Queue new request for a docker instance
- Cancel pending requests
- Insert external data into the container for processing
- Retrieve results after processing

A admin must be able to...

- Monitor the status of the queue
- Monitor the status of the docker environment
- Identify the container's creator

2.2 Design overview

This section contains a brief overview of how the end result should look like, and a brief description of the tools and technologies leveraged.

2.2.1 Design illustration

Figure 2.1 illustrates how users can connect to the server through a web interface, and also get access to the shared storage through the server. The server will then take request and spread them across the worker nodes, and save the results on a shared storage medium for retrieval.

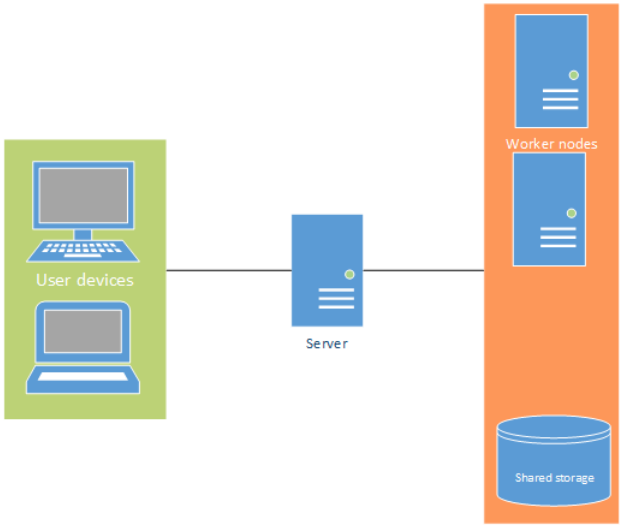

Figure 2.1: High-level overview of how devices will connect to the finished infrastructure

2.2.2 Tools

- VIM[5] and Visual Studio Code[6]
  These are the primarily used editors and IDE for the project

- Git[7]
  Is a version control system, that enables a group of developers to work together on the same project in an organized manner

- Bitbucket[8]
  Distributed version control system for usage with Git

- JIRA[9]
  Issue tracker and time logger

- Vue.js devtools[10]
  Chrome and Firefox extension that enables developers to inspect and edit live Vue components

- Postman[11]
  Allows developers to create custom requests to API and inspect the response

2.2.3 Technologies

- Docker [12]
  Docker is a tool to deploy and manage docker containers
- Docker-compose [13]
  Docker-compose used for handling a multi docker environment.
- MySQL [14]
  MySQL is a relational database that allows persistent storage of queriable data
- Kubernetes [15]
  Kubernetes is a container orchestrating tool
- Grafana [16]
  Grafana is used to graph metric data from multiple data sources
- Prometheus [17]
  Prometheus is a time-series database
- Cadvisor [18]
  Cadvisor gathers live information from docker containers
- Alertmanager [19]
  Alertmanager is used to manage the alerts regarding the data
- Node aexporter [20]
  Nodeexporter collects host metrics

2.2.4 Frameworks and libraries

- py3nvml [21]
  A Python library used to gather information about NVIDIA Graphics Card
- Vue.js [22]
  A framework to build full-fledged web applications in javascript
- Laravel/PHP [23]
  A framework to build PHP based web applications

Chapter 3

Theory

This is the secret document from jerrys work ftp... HEY YOU ARE NOT SUPPOSED TO BE HERE...

3.1 Queuing

Queues facilitate systems to serve work orders in an orderly and predictable fashion. It was first mathematically formulated by Agner Krarup Erlang which created queuing theory as a statistical field of study. Through the predictability of queuing theory we can make decisions on how to scale and coordinate systems based on demand, number of servers and acceptable waiting times. There are three main problems within the field; behavioural problems, statistical problems and decision problems. There are multiple models depending on the kind of queuing issue one is to solve, and what data are available to evaluate. In general, any queue can be modelled as a stochastic variable over time, with dependencies on the input into the queue and departure time of each request.[24] The input can be modelled as a Poisson process defined by

$$X(t) \sim P_n(t) = e^{-\lambda t}\frac{(\lambda t)^n}{n!}$$

which is to say that the probability of n inputs in a time span of t is given by the equation with the mean of $E[X(t)] = \lambda t$, where λ is the rate of occurrence. [24]. Due to the nature it is fair to assume that λ itself is not known exactly, thus λ must also be statistically distributed. Using the Bayesian theorem of inference for the Poisson process we can estimate λ as a gamma distribution with prior hyper parameters $\kappa_0 = \tau_0 = 0$ and posterior hyper parameters $\kappa_1 = \kappa_0 + n$ and $\tau_1 = \tau_0 + t$, then

$$\lambda \sim \gamma_{(\kappa_1,\tau_1)}.$$

Since λ now is following a distribution, $P_n(t)$ will not be precise enough, as such we update to use the gamma-gamma function instead, so

$$X(t) \sim P_n(t) = g\gamma_{(n,\kappa_1,\tau_1)}(t) = \frac{1}{B_{(n,\kappa_1)}} \cdot \frac{\tau_1^{\kappa_1} \cdot t^{(n-1)}}{(\tau_1 + t)^{(\kappa_1+n)}}$$

where B is the Euler Beta function defined by

$$B_{(p,q)} = \frac{\Gamma(p)\Gamma(q)}{\Gamma(p+q)} \quad [25]$$

where Γ is the gamma function. [26] Due to the lack of data this has not been modelled for this project, and from the impression of the current rate; the law of small numbers would imply any estimate would be wildly inaccurate and most likely misleading.

Chapter 4

Project Solution

This chapter consists of the functional- and technical design of the project as a means to resolve the problem statement.

4.1 Functional Design

4.1.1 Front end

Login

The intention for this system was meant to run as an internal site for students and employees, the group choose to remove the register button as one would login with ones existing UiA Login details.4.1

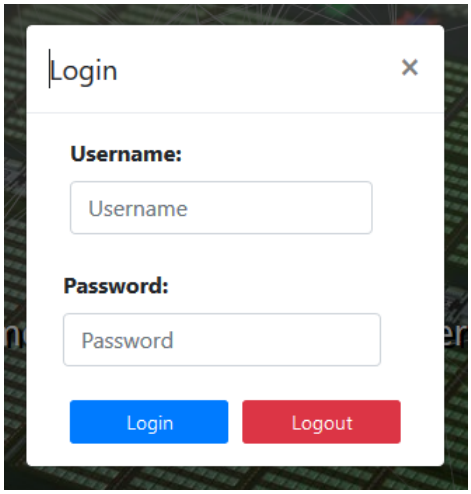

Figure 4.1: Login modal with login form, for getting access to monitor and request containers

Monitoring

As monitoring containers was an important key in the preliminary report we made a simple page where one could instantly get the status of a container and how much resources a container have been allocated.4.2

Figure 4.2: Monitoring your personal containers

4.2 Technical Design

4.2.1 Architecture overview

The architecture, as shown in figure 4.3, is set up such that it will have possibilities to scale with increased demand for both compute- and storage capacity. It is flexible enough to allow for up and down scaling of resources, both from a physical infrastructure view as well as allow for individual parts such as the front end application to be scaled to multiple instances or multiple workers if demand increases.

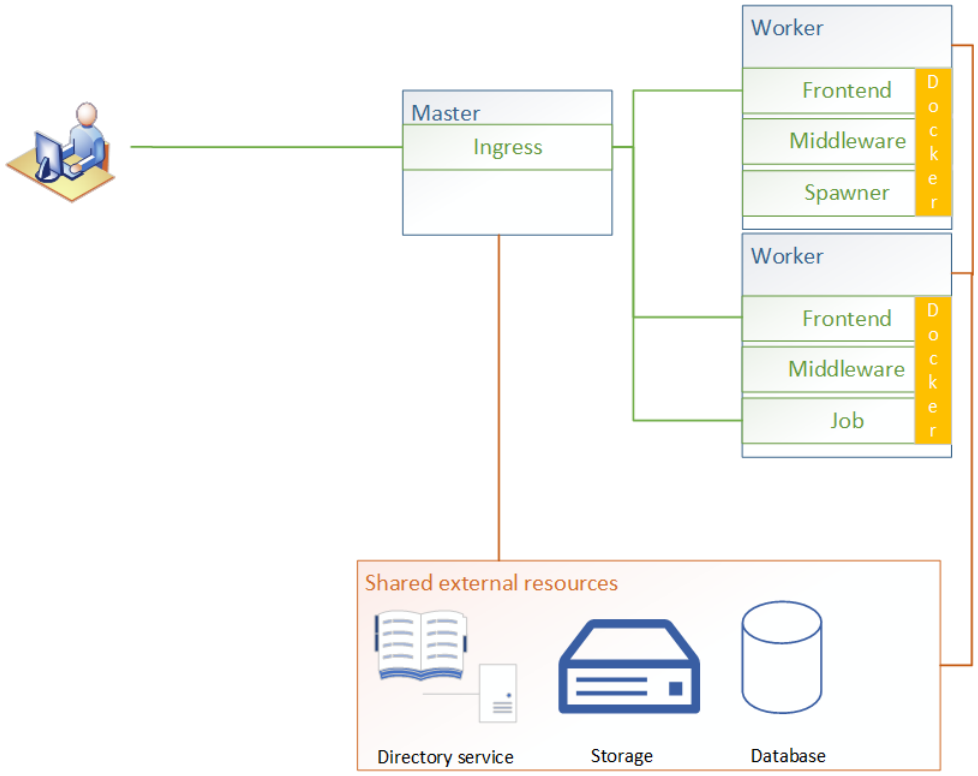

Figure 4.3: High level overview of the architecture of the finished design

Leveraging the systems themselves and using them for what they are designed for. For example MySQL has clustering capabilities, it is therefore better to let the database itself handle its own clustering and scaling instead of building this into Kubernetes using deployments. Running the database outside the cluster also frees as much resources as possible to the compute jobs on the worker nodes.
It is possible to run the master on multiple nodes to ensure high availability, as showing in figure 4.4, using a third party load balancer to distribute requests. [27]

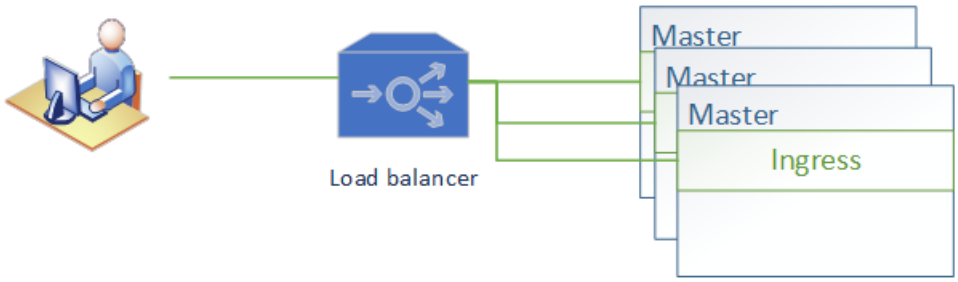

Figure 4.4: Load balanced master cluster

4.2.2 Middleware

Authentication flow

To allow a centralized authentication scheme that can be used with other resources, leveraging existing infrastructure and not having to handle user data in a seperate system, we utilize LDAP. When the user visits the front end he or she will be prompted to provide a username and password to the system, this is then forwarded as a login request to the directory service for validation. If the combination is incorrect the user will not get an access token and as such should not be allowed further onto the system. If the combination is a success, the system will check if the user is part of a valid directory group that is set up for the service, for example "DGX-Users". If this is true, the system will generate an access and refresh token that will be sent back to the user which will be used for further authentication. Figure 4.5 illustrates how the authentication workflow is processed.

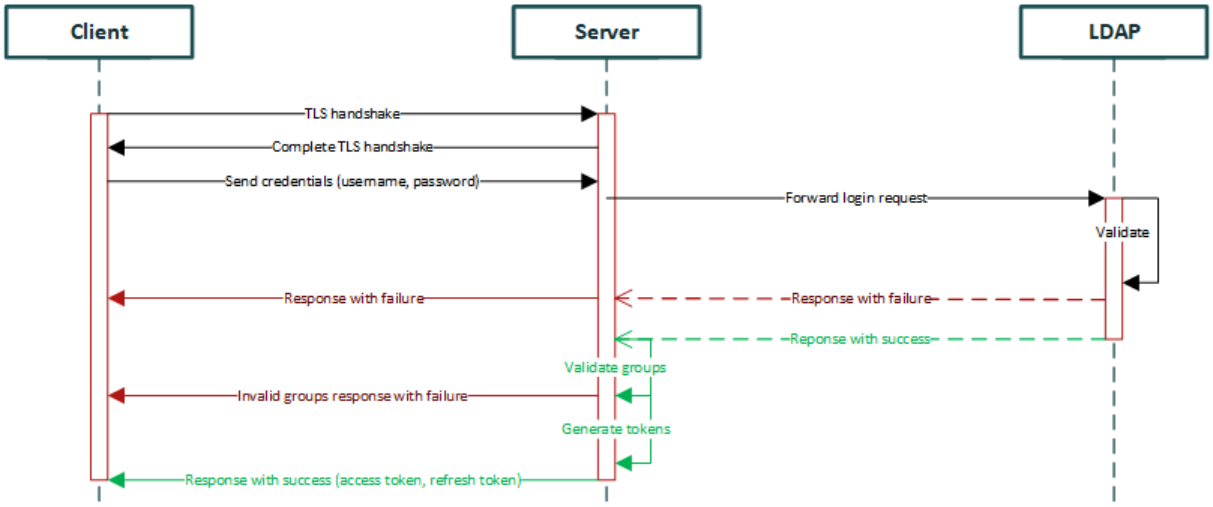

Figure 4.5: Authentication flow

Job model

The job model describes how a request for resources should be stored in the database, and what fields are required for future processing. These fields with description can be seen in table 4.1. It contains information that will be available to the user interface through the API so the user can monitor their requests and get information on how to connect. It also contains information useful for the back end to work with the Kubernetes cluster.

Field	**Type**	**Value**
id	int	An incremental value assigned by the database
CName	varchar	The canonical name of the job specified by the requester
Author	varchar	The username of the requester, this is set by the system on creation
CPU	int	The maximum limit of the number of CPU cores, this can not be less than 1
RAM	int	The maximum limit of the amount of RAM
GPU	int	The number of GPUs requested
Status	int	Specifies the status of the job, see table 4.2 for more detail
Date created	timestamp	The date and time the request was created
Date modified	timestamp	The date and time the request was last modified
SSH	int	Port number for connection to the container
Jupyter	int	Port number for connection to the Jupyter on the container
Deployment/service prefix	varchar	A string composed of the username-UID which will be the name of the Kubernetes deployment and service names

Table 4.1: The definition of the job model to be stored in the database

Figure 4.6: Shows an example of a JSON object of a job returned from the middleware

Code	**Description**
0	Stopped
1	Running
2	Pending restart
3	Pending stop
4	In queue

Table 4.2: The status codes of the queue model

When information for a job or all jobs for the user is requested, or a new job is put on the queue successfully, a JSON formatted object will be returned containing details for the front end to display, a sample request is shown in figure 4.6.

Middleware API routes

The back end contains two controllers

- AccountController
  Handles routes for login and logout
- QueueController
  Handles routes related to issuing and retrieving job requests

The controller endpoints can be seen in table 4.3

Route	**Description**
/account/login	Issues a valid token on success full login
/account/logout	Deauthorizes the token
/queue	Retrieves all the items in the queue for the logged in user
/queue/id	Retrieves a specific item for the queue for the logged in user
/queue/create	Post a new job request
/queue/stop/id	Request that the job be stopped
/queue/restart/id	Request a restart of the job

Table 4.3: Available API endpoints

New job request

The processing of a new job request to be put on the queue starts with checking the validity of the token, this is to ensure that even if you know the route one should not be able to post requests. Once authorization is confirmed, the input data should be validated against a rule set as defined in table 4.4

Field	**Rule set**
CName	Should be a string, of maximum length 255
CPU	Should be an integer, a minimum and maximum size should be proportionate to the average server in the cluster
RAM	Should be an integer, a minimum and maximum size should be proportionate to the average server in the cluster
GPU	Should be an integer, of value 0 to a maximum that should be proportionate to the average server in the cluster

Table 4.4: The basic rule sets for the input fields when requesting a new job from a high level

Given the unknown nature of the jobs to be queued the limits should be flexible enough to allow multiple jobs to be running at the same server, while at the same time not be to strict with regards to user requirements. This needs approval and discussion within the organization, and should be accepted from the top of the chain.
The request should then have the author, status and time stamps appended before being stored in the database for further processing. Finally the endpoint should return a JSON object with the registered information to the front end. The flow of this procedure is shown in figure 4.7

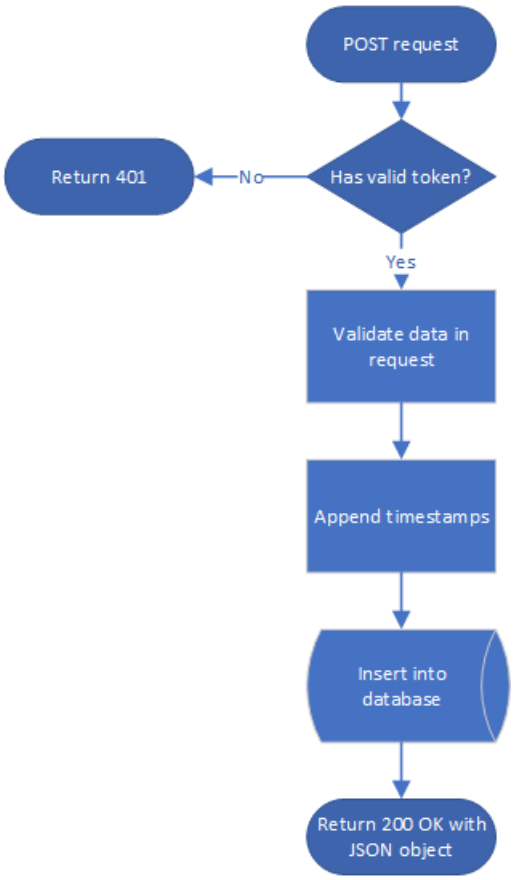

Figure 4.7: New request processing at a high level

Getting jobs for the user

The processing of getting jobs starts with checking the validity of the token. This is to ensure that only a valid user can retrieve their own jobs and not the jobs of someone else. If the token is invalid we immediately return a 401 Unauthorized. This also has the side effect that trying to probe for valid job id's will net a 401 no matter if the id is valid or not.
The process then retrieves all jobs for the user and formats it into a JSON object, which is returned to the front end. This process can be seen in figure 4.8

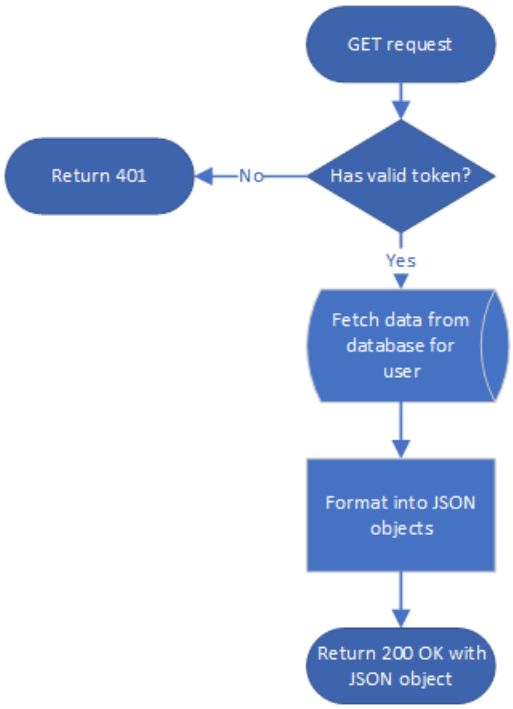

Figure 4.8: Fetching all jobs in queue for the user from a high level

The process of getting a single job is equivalent in its process, as seen by figure 4.9

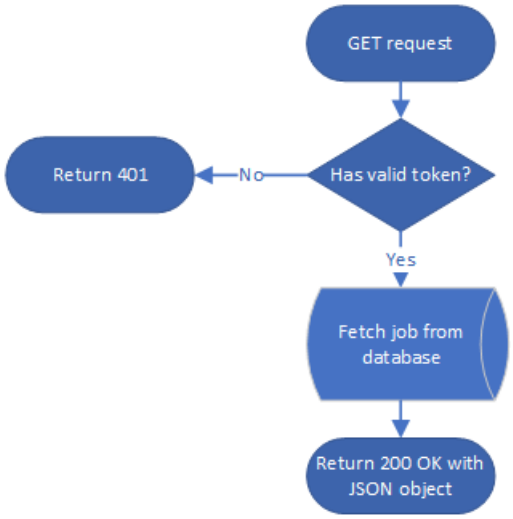

Figure 4.9: Fetch a specific job for the user from a high level

Restart or stop job by id

The process of restarting or stopping a specific job starts by validating the token, this is to ensure that only a valid user can update his or her job. If the token is invalid it should return 401 Unauthorized immediately. Then verifying the existence of the job in the database, and subsequently updating with the correct status code as given in table 4.2. Data changed should also be updated to reflect the time of the action. Finally the edited job is returned as a JSON object. The process can be seen in figure 4.10.

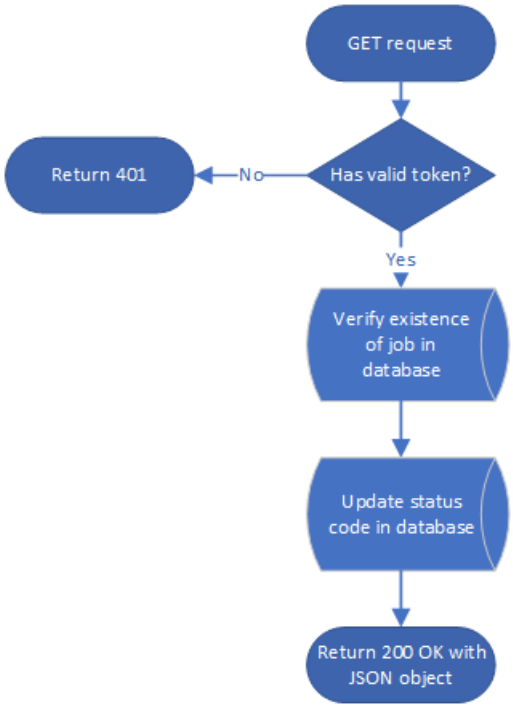

Figure 4.10: Update the status code as requested by user on restart or stop request from a high level

4.2.3 Queue Manager

The queue manager's job is to respond to requests in the queue. This entails creation, restarting and removing of deployments in Kubernetes. It is running as a single entity to ensure that we do not have concurrency issues. The workload should be light enough that a single instance is sufficient. Running separately from the middleware this also ensures that should one get unauthorized access to the middleware layer, this does not give you any access to the Kubernetes cluster. A high level overview can be seen in figure 4.11. The queue manager will run on a timed trigger, for example every 5 minutes to process the queue.

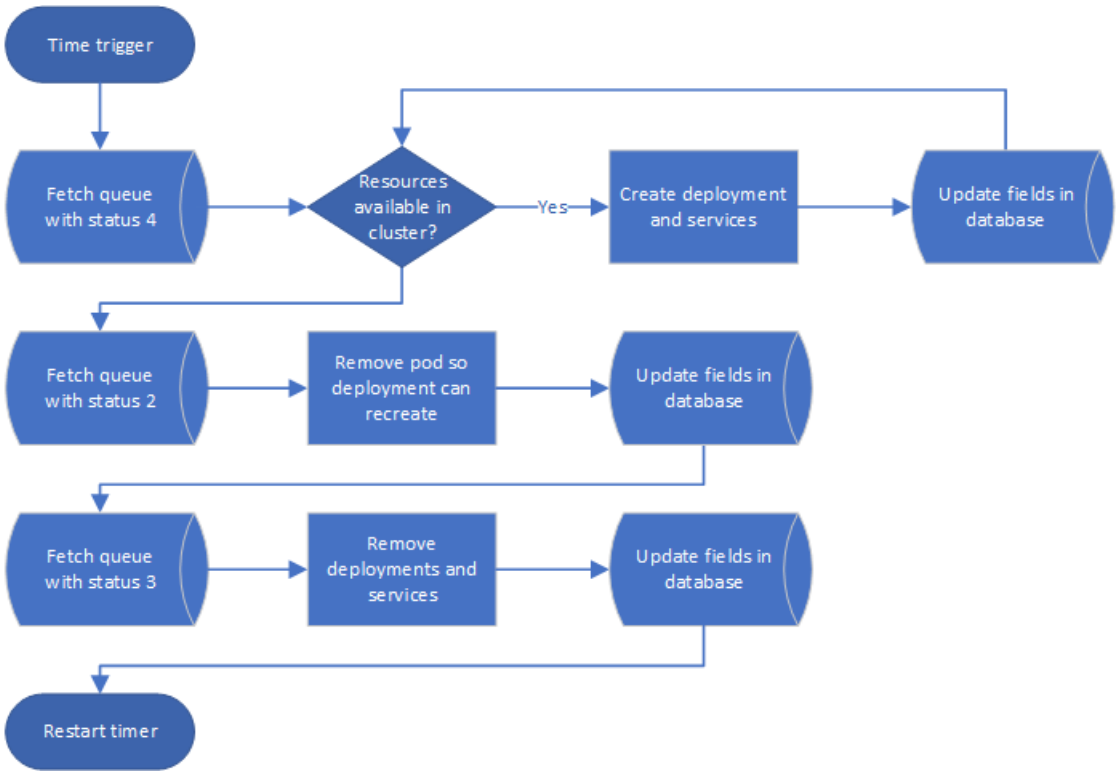

Figure 4.11: Timed creation, restart, stop of deployments for the work requests from a high level

New requests

The queue manager will query the database for all jobs that are new sorted from oldest to newest based on created time stamp. Given the impression of a relaxed workload, this will work as a FIFO queue. Based on the specifications of the first job it will check if any worker node in the cluster has enough resources available, if not enough resources are available then it is done.
If resources are available it will create a deployment based on the specifications and push this to Kubernetes. This will ensure that the pod will be recreated should the current working node go down or the pod be killed by some other means. It will also create a service endpoint so SSH and Jupyter can be reached from outside the cluster. Finally it will update the fields in the database with the prefix of the deployment and service in accordance with table 4.1.

The drawback with using the FIFO queue model, is that a single job, that requires all the GPU capacity of a worker node, can block all other requests. As such an implementation of weighted fair queuing can alleviate this to some extent by weighing based on resource specification by setting a limit on what is a "fair" amount of resource usage. One can then prioritize getting smaller jobs through by treating for instance 0 to 4 GPUs as fair. This has business implications and does need more analysis based on usage patterns as well as discussion and approval from CEO level.

Restart requests

This will query the database and retrieve all jobs with the restart status code, using the prefix it then identifies the correct deployment and can then get access to the specific pod that the job is running under. Deleting the pod, Kubernetes will attempt to do a graceful shutdown, and then recreate the pod. Finally it will then update the status in accordance with table 4.2 and changed time stamp field in the database.

Stop requests

This will query the database and retrieve all jobs with the stop status code, using the prefix it then identifies and removes the correct deployment and services from Kubernetes. Finally it updates the status code in accordance with table 4.2 and changes the changed time stamp field in the database.

4.2.4 Kubernetes

Master

The master node is responsible for managing the cluster, and distributing the jobs across the worker nodes. Using docker through official channels, it is set up with flannel for in-cluster routing, kube-dns for namespace resolution inside the cluster, as well as applying the feature set to schedule GPU resources.

Worker without GPU

A worker without GPU requirements only requires docker from the official channels, as well as the kubelet and kubeadm services.
Using `kubeadm join --token <token> --discovery-token-ca-cert-hash <hash>` to join the cluster, the master will automatically assign required infrastructure to fulfill requirements. The tokens by default only have a 24 hour lifespan, as such it is required to create a new token on the master node, the simplest way is
`kubeadm token create --print-join-command`, one can append `--ttl <duration>` in the format of XhYmZs, if the value is 0 it will never expire. [28]

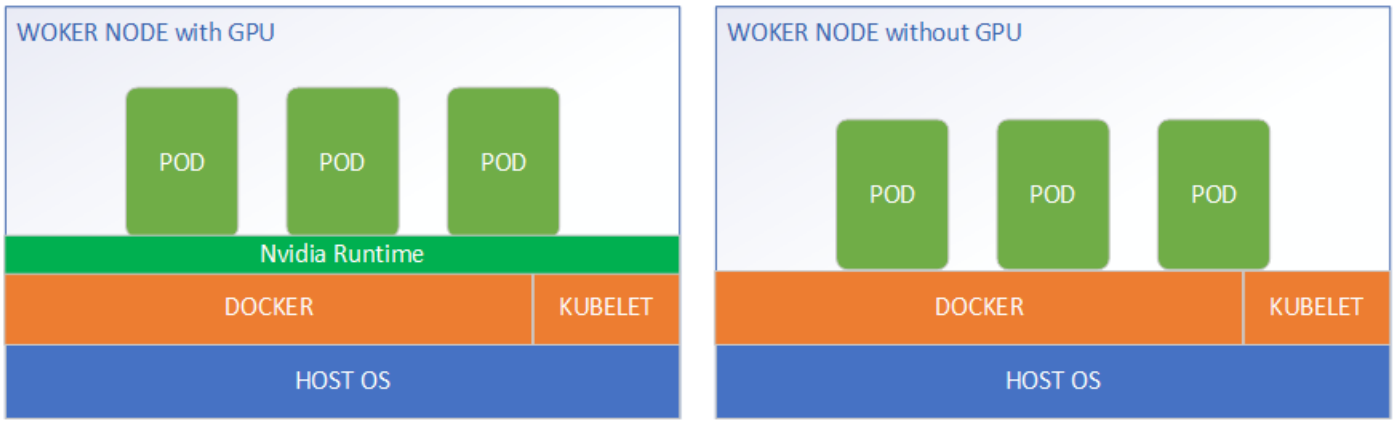

Figure 4.12: Worker nodes with software components installed, both with and without a GPU resource to be used

Worker with GPU

A worker with GPU requires the installation of a compatible driver, testing has shown version ≥ 390 works, as well as CUDA drivers. To leverage NVIDIA-docker the Docker community edition has to be used and NVIDIA has to be set as the default runtime for Docker. Upon

joining the cluster as described in "Worker no GPU" the master will assign the required infrastructure to fulfill requirements.

Ingress

The ingress is responsible for forwarding requests to the front end of the application from outside the cluster using a Kubernetes ingress control pointing to a Kubernetes service. This ensures that the deployment of the front end application can be replicated over multiple hosts as demand increases, and the service is responsible for identifying and forwarding to running pods.

Jupyter and SSH access

Through the usage of Kubernetes nodeport service requests can be forwarded based on the port number to a pod.

4.2.5 Shared external resources

Shared storage

To enable independence of which host the job will be run at, a shared external storage should be used. This is due to the size of training data for AI purposes can be quite large, thus having to replicate this to every node is not feasible. This further allows the administrator an easy to manage storage infrastructure that can be scaled separately from Kubernetes.
As such NFS can be leveraged through a shared storage network that is separate from the cluster, using a path name scheme such as `/storage/share/<username>` will allow definition of NFS mount points to only mount the requester' structure as volume on the created pod.

Database

The database is responsible for storing the queue information. This can be run inside the cluster, but it is recommended to run outside. This is to leverage the database platform itself to have a distributed system, and so it can scale independently of Kubernetes. As well as freeing more resources to process job requests.

Authentication

Since the a deployment will happen in an existing environment, there is no need for an independent system of user names and passwords. As such leveraging the abilities through

LDAP to access directory services outside of the cluster will ensure availability and easier user management through the usage of group assignments for authorization into the system. Through an API endpoint authentication is performed, and an access token is generated for the session which is then used for authorization to the queue endpoints as described in the section 4.2.2

4.2.6 Monitoring

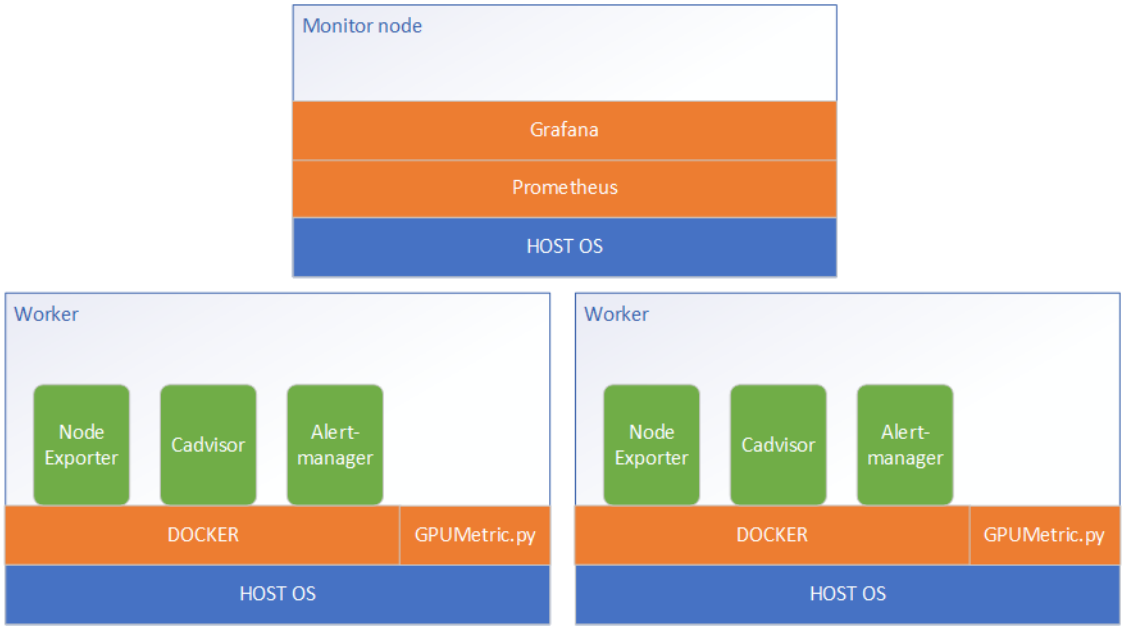

Figure 4.13: An overview of the monitoring stack

The group managed to get most of the monitoring working for both dockers and the GPU. Each of the services below is ran in Docker containers, and is managed with Docker compose. Due to that all of the services need to communicate with each other it is needed that all of them would start at the same time as well as it makes everything a lot easier to manage when its supposed to be deployed on multiple nodes. However the GpuMetric script would not be running in a Docker container as it would occupy all the GPUs on the system as a GPU can only be allocated once. This would mean that it had to be ran with a service manager such as systemd [29] or systemctl [30].

GpuMetric

GpuMetric is a python script written by the group to gather information about the NVIDIA specific GPU's and push it to a text file, which then can be imported by the Prometheus

[17] node-exporter [20]. GpuMetric is using a python library py3nvml [21] to gather specific information such as temperature, memory- and power usage about the graphics cards on the host.

Node Exporter

Node Exporter is part of Prometheus and it is in command of ensuring to collect information about the host.

Alertmanager

Alertmanager [19] is part of Prometheus, and its purpose is to handle the alarms that is being sent from Prometheus.

Cadvisor

Container Advisor [18] is a tool made by Google to collect live metrics from running containers.

Prometheus

Prometheus is a time-series database and a powerful monitoring tool. It has multiple integration points as well as an efficient way of storing data.

Grafana

Grafana is a highly customizable dashboard for analyzing and monitoring data both real time and historical data. It also support multiple data sources like here we are using Prometheus to point out one of the most popular ones.

Chapter 5

Discussion

5.1 Front end

Front end programming language

The group did not manage to produce a working proof of concept for the front end, due to the lack of a working back end API. It was initially thought of to make a front end where user would be able to request containers based on their hardware needs, monitor their state meaning running, stopped or pending in queue. Users were also intended to be able to request to have their containers stopped and restarted. It was also intended that an administrator should be able to watch the entire queue.

The front end was written in Vue.JS due to it being a simpler framework than most of the other that exist today. Though the framework was simpler than most frameworks it was still though to make a front end since the group had no prior experience with front end development. It was also considered to use React as the framework for the front end, but after some research it was concluded by the group that it was a rather steep learning curve in comparison to Vue.JS .

5.2 Back end

Docker management

Kubernetes is still in a rapid development process. It is starting to mature but the system itself is complicated as it tries to be modular enough to fit into as many environments as

possible. As such there is no "standard way" to deploy Kubernetes, it has to be customized for each environment. This also makes finding relevant examples outside the official documentation hard, as most discusses single server instances usually through what is called Minikube [31]. GPU support is also very fresh through the use of NVIDIA-docker which was only recently released. As mentioned in existing products the group could not find any projects that could be used in place and as such Kubernetes was the only product we could have used, if not we would have to develop our own Docker management platform but that would be way out of scope and have multiple drawbacks.

Back end programming languages

The group did not manage to produce a working proof of concept for the middleware API and the queue manager. Initially both where intended to be written in Golang as an extension in the Kubernetes dashboard. But due to the complexity of the Kubernetes source, as well as lack of experience with Golang, this was deemed too ambitious and decided against. It was therefore experimented to write the middleware API in PHP using Laravel, but the group did not manage to get this working properly with the authentication. The queue manager was still to be created in Golang as there are libraries to communicate with the cluster available. But due to time constraints and lack of development experience in these languages, none of these were successfully implemented.

Database

During the attempted development, MySQL was used as the members of the group already had existing MySQL instances running and was somewhat familiar with it. There are no preferences on which SQL database to be used, considerations to take into account are what existing engines are running that can be leveraged, what the frameworks being used are supporting, and the database administrators are familiar with.

Queuing

It was intended to implement a First In First Out queue, due to the fact that to implement a proper Weighted Fair Queue we would need to assign what is defined as fair in the current environment. This would imply that we would need consensus from management and the users, and we would need data with respect to the amount of requests as well as resource demands to make an informed decision. As such a WFQ is left as potential future work if a FIFO implementation is not sufficient as the demand increases. Given the modularity this can be swapped in place once an implementation is in place.

Authentication

Using other systems to authenticate such as the built in user management in Laravel, or other identity management systems are possible, but this will have impacts on modularity if the application is to be rewritten to better support a developers specialization with regards to programming languages. We therefore still believe that leveraging the existing directory services will net the best result, as this gives flexibility for future development in terms of authorization using a token based approach once the user is authenticated through LDAP.

5.3 Future work

Future work details additional features that should be researched and implemented to make a more flexible, robust and secure system which the group did not get to work on due to limited time.

Feature	**Priority**
Central docker image repository	High
Jupyter password	High
SSH password	High
Run custom dockerfile	Medium
Administration panel	Medium
Detecting when a job is done	Low
Logs and stdout output available to requester	Low

Table 5.1: Future work

Central docker image repository

By default each node in Kubernetes has its own local collection of Docker images, if a request comes in which requires an image the node does not have already it will look at the public Docker hub. This means that if custom images are to be used, it has to be created on all potential nodes a pod could be created on. As such a solution that lets for instance the Master node to be a central repository would be appropriate, meaning that if the worker does not have the image already, it should first ask the master node before trying to use the public Docker Hub.

Jupyter password

As of the current implementation a simple port scan will reveal running instances of Jupyter, these are by default not protected and as such will accept any connection.
A solution for Jupyter seems to be the ability to pass the password as an environment variable upon creation of the container as discussed in an issue on GitHub. [32] One could then generate a random string and pass this to the container, and expose it through the front end interface.

SSH Password

As with Jupyter, currently the Tensorflow image does not have any root password, as such anyone can reveal and access running pods over SSH by a simple port scan. There are two avenues of approach

- **Custom docker container**
  On every request that gets a deployment set up, we can auto generate a dockerfile that will run the appropriate commands to create a new image with a custom password. This does have dependency on the central docker image repository to be feasible on scale.

- **Execute remote commands into pod**
  Once the pod is created we can execute remote commands into it, the drawback is that every time the pod is recreated the password will not persist. As such there is a need for a timed service that checks and keeps passwords up to date on running compute deployments.

Run custom dockerfile

To increase flexibility it should be possible to upload a custom dockerfile upon requesting resources. One would then have to confirm that the dockerfile is valid, and distribute it to the cluster in one of the following ways

- **Centralized docker image repository**
  If the task of a centralized docker image repository is solved, then we can simply create the Docker image on the particular centralized node, and the worker that is assigned will fetch the image.

- **Wait for worker choice**
  Once the worker node is selected, we can detect the node and assign that node to build the docker image, which once ready will be picked up by the Kubernetes deployment process.

- **Create on all worker nodes**
  Lastly we can build the image on all worker nodes.

The two latter ones are an administrative nightmare as the cluster scales, as such this is a medium priority and should be set to high once the centralized repository has been solved.

Administration panel

As of now there only exists an interface for the users, an administrative panel for the queuing system should be put in place to easier manage jobs and get an overview. This panel should

also include usage statistics as well as predictive calculations based on the queue model defined in chapter 3.

Detecting when a job is done

In current design once the job is done, the system has no way of detecting this. As such the user is responsible to log onto the front end and stop the job manually to remove it from the queue, and thus releasing resources back to the cluster. An idea that has been discussed is to create a long unique identifier for each job when their deployments are created. This identifier should be passed into the pods as a variable that scripts can pick up and then subsequently use to call an API endpoint when the job is done.

Chapter 6

Conclusion

When we started, the intention was to create a full proof of concept implementation of the system that should be easy to use, improve, scale, and maintain over time. But due to time constraints, as Kubernetes alone took most of our time to understand and lack of development experience between the group members, we did not manage to fulfill this. However we still believe this is a promising avenue of approach. We hope that even if we did not manage to implement a working proof of concept, we have shown through the theory and models that there is a good foundation to build upon and implement if someone with more development experience are to attempt the implementation. As we have shown through the future works section, there still is a fair amount of research, testing and work to be done before the product is fully ready for a production environment.

This project has been a challenge for the entire group. In hindsight, it might have been a bit too ambitious. However, from a personal perspective we feel it was a success, as we managed to lay the groundwork for further work. We also got experience with multiple new technologies as well as pushed our limits when it comes to software development, and as such we have grown our basis for decision making in future projects.

References

[1] NVIDIA, *NVIDIA DGX-1: Essential Instrument of AI Research*, [accessed May 2018]. [Online]. Available: https://www.nvidia.com/en-us/data-center/dgx-1.

[2] *B. dyrli, ø. andreassen, nvidia-dgx-1 preliminary report, uia, grimstad, norway, 20. feb 2017.*

[3] Docker Inc., *Swarm mode overview*, [Accessed May 2018]. [Online]. Available: https://docs.docker.com/engine/swarm/#feature-highlights.

[4] Red Hat, *OpenShift: Container Application Platform by Red Hat, Built on Docker and Kubernetes*, [Accessed May 2018]. [Online]. Available: https://www.openshift.com.

[5] *welcome home : vim online*, [Accessed Apr. 2018]. [Online]. Available: https://www.vim.org.

[6] Microsoft, *Visual Studio Code - Code Editing. Redefined*, [Accessed Apr. 2018]. [Online]. Available: https://code.visualstudio.com.

[7] Git, *Git*, [Accessed Apr. 2018]. [Online]. Available: https://git-scm.com.

[8] Atlassian, *Bitbucket | The Git solution for professional teams*, [Accessed Apr. 2018]. [Online]. Available: https://bitbucket.org.

[9] ——, *Jira | Issue & Project Tracking Software | Atlassian*, [Accessed Apr. 2018]. [Online]. Available: https://www.atlassian.com/software/jira.

[10] *vuejs/vue-devtools*, [Accessed April 2018]. [Online]. Available: https://github.com/vuejs/vue-devtools.

[11] Postdot Technologies, Inc., *Postman*, [Accessed Apr. 2018]. [Online]. Available: https://www.getpostman.com.

[12] Docker Inc., *Docker*, [Accessed Apr. 2018]. [Online]. Available: https://www.docker.com.

[13] ——, *Docker Compose*, [Accessed Apr. 2018]. [Online]. Available: https://docs.docker.com/compose/overview/.

[14] Oracle Corporation, *MySQL*, [Accessed Apr. 2018]. [Online]. Available: https://www.mysql.com.

[15] Kubernetes, *Kubernetes*, [Accessed Apr. 2018]. [Online]. Available: https://kubernetes.io.

[16] Grafana Labs, *Grafana - The open platform for analytics and monitoring*, [Accessed 29. Apr. 2018]. [Online]. Available: https://grafana.com.

[17] Prometheus, *Prometheus - Monitoring system & time series database*, [Accessed Apr. 2018]. [Online]. Available: https://prometheus.io.

[18] *google/cadvisor*, [Accessed Apr. 2018]. [Online]. Available: https://github.com/google/cadvisor.

[19] ——, *Alertmanager | Prometheus*, [Accessed Apr. 2018]. [Online]. Available: https://prometheus.io/docs/alerting/alertmanager.

[20] *prometheus/node_exporter*, [Accessed Apr. 2018]. [Online]. Available: https://github.com/prometheus/node_exporter.

[21] *fbcotter/py3nvml*, [Accessed May 2018]. [Online]. Available: https://github.com/fbcotter/py3nvml.

[22] Evan You, *Vue.js*, [Accessed Apr. 2018]. [Online]. Available: https://vuejs.org.

[23] T. Otwell, *Laravel - The PHP Framework For Web Artisans*, [Accessed Apr. 2018]. [Online]. Available: https://laravel.com.

[24] U. N. Bhat, *An introduction to queueing theory, second edition*. Birkhäuser Basel, 2015, [Accessed April 2018].

[25] E. W. Weisstein, *Beta function. from mathwolrd–a wolfram web resource.* [Accessed May 2018]. [Online]. Available: http://mathworld.wolfram.com/BetaFunction.html.

[26] S. O. G. Nyberg, *Statistikk - en bayesiansk tilnærming. Første utgave*. Universitetsforlaget, 2016, []Accessed May 2018].

[27] Kubernetes, [Accessed May 2018]. [Online]. Available: https://kubernetes.io/docs/admin/high-availability/building.

[28] ——, [Accessed May 2018]. [Online]. Available: https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm-token.

[29] Freedesktop, *systemd*, [Accessed May 2018]. [Online]. Available: https://www.freedesktop.org/software/systemd/man/systemd.html.

[30] ——, *systemctl*, [Accessed May 2018]. [Online]. Available: https://www.freedesktop.org/software/systemd/man/systemctl.html.

[31] *kubernetes/minikube*, [Accessed May 2018]. [Online]. Available: https://github.com/kubernetes/minikube.

[32] *What is login password in the jupyter notebook? · Issue #6351 · tensorflow/tensorflow*, [Accessed May 2018]. [Online]. Available: https://github.com/tensorflow/tensorflow/issues/6351.

Appendix

A1 - Preliminary Report

preliminary_report.pdf

A2 - Timesheet

timesheet.pdf