Graduation Report

Implementing container scheduling technology to fit the needs of a fast growing distributed infrastructure

Vasil Lozanov

Sqills

Table of Contents

1. Introduction & Abstract

This document is a self-sufficient report on the graduation project about container orchestration solutions. The report contains the problem definition and analysis, context and stakeholder analysis, research and design and test of the designed solution.

The project is based on current available solutions and takes into account relevant papers from the industry.

Generally speaking workload scheduling is not a new concept. Google, for example, has published papers on their internal scheduling solutions (predecessors of Kubernetes) all the way in 2013 and 2015. (Schwarzkopf, 2013) (Verma, 2015)

This project is heavily influenced by that original research, since it's one of the first to coin the term “container scheduling” and includes a lot of detailed information about how such systems work.

2. List of Used Terminology

Term	Meaning
ALB	Application Load Balancer. AWS' managed layer 7 load balancing service.
AMI	A template for running EC2 instances.
AWS	Amazon Web Services
AWS Secrets Manager	Managed service by AWS for safe storage of sensitive data.
AWS Systems Manager	AWS service which allows users to group resources, execute bulk tasks, automate certain actions, get shell access on EC2 instances running the AWS Systems Manager daemon and store parameters and secrets.
AWS Systems Manager Parameter Store	Key-value store, part of the AWS Systems Manager. Supports storage of sensitive data as well.
Ansible	Tool for automating deployments and configuration management.
CentOS	A Linux operating system.
Container	An isolated application environment.
Docker	The leading container engine.
EC2	Elastic Compute Cloud. AWS' managed VMs service.
EKS	AWS Elastic Container Service for Kubernetes. Managed Kubernetes master cluster by provided by AWS.
HashiCorp	An American company that develops, among others, Consul, Vault, Nomad and Terraform.
HCL	HashiCorp Configuration Language. Used mainly for writing configuration files for Nomad, Terraform, Consul and other applicable HashiCorp products.
JSON	JavaScript Object Notation. Widely popular for configuration files and REST APIs.
Microservice	A small application with a specific purpose.
NLB	Network Load Balancer. AWS' managed layer 4 load balancing service.
Packer	A tool for making AMIs.
Rkt	An alternative container engine.
S3 Passenger	A SaaS platform for (among others) inventory management, booking orchestration and ticket sales developed by Sqills.
Sqills	Company in Enschede, developer and provider of the S3 Passenger platform.
Terraform	Tool for implementing infrastructure-as-code.

Table 2.1 - List of used terminology.

3. Problem Definiton

Sqills is looking to expand their business which means they should be able to facilitate a growing number of clients. In order to make that possible the company is looking into automating the deployment and hosting of S3 Passenger (more information in chapter 4.1) environments so that they can roll out environments for new customers faster than it currently is (current deploy from scratch taking a couple of hours).

The company sees the following as the main problems they want to tackle with the introduction of a container scheduler:

- Automate the deployment of new environments, because the current way requires a lot of manual steps.
- Raise the abstraction level of infrastructure management, because the current infrastructure requires deep knowledge of the intricacies of AWS which slows down the process of orientation of new employees.
- Facilitate easy auto-scaling of environments, since the current infrastructure does not scale automatically and requires a lot of manual steps to do so.
- Make hosting of the S3 Passenger platform more efficient since schedulers can utilise available resources more efficiently than manually deployed containers.

The company is also interested in the following as aspects that may be improved by a container scheduler:

- Central configuration management
- Central service management and service discovery
- Easier service meshing
- Easier scalability
- Possible implementation of extra security solutions (such as Vault)

Any solution that will be developed must be designed with the following key elements in mind:

- Security
- Scalability
- Availability
- Maintainability

3.1 Main and Sub-Questions

These are the questions distilled from the causal model and brainstorming mentioned in this chapter.

Main Question: What is the most efficient solution for implementing container scheduling technology to fit the needs of a fast growing distributed infrastructure.

Sub-questions:

- What requirements does the company have for a possible solution?
- What do the current container management practices look like?
- What container scheduling technologies are there?
  - Pros?
  - Cons?
- How do container scheduling technologies integrate with the currently used tools within the company?
- What are best practices for container scheduling?
- How are container scheduling technologies managed?
  - Monitoring
  - Auditing
- How to provide proof of a working solution?

4. Context Analysis

In this chapter information about the context of problem tackled in this project can be found. Important parts of this analysis is the current and desired situations.

4.1 S3 Passenger

“S3 Passenger” is a booking, inventory management and scheduling platform for passenger transport companies developed by Sqills. The platform itself consists of multiple micro services and 3rd party integrations. Currently all components (except the databases) are packaged in Docker containers. S3 Passenger is **not** a multi-tenant platform, thus every customer gets their own separate environments (usually multiple environments per customer for test, acceptance and production). This is by design since the platform has to comply with various legislations and standards.

4.2 Current Situation

The company has already migrated multiple customer environments (including 1 production environment) from Previder to AWS using a lift-and-shift approach (with some modifications). Some automation has been introduced - Terraform, for example, for implementing the infrastructure-as-code approach. Furthermore Sqills uses Ansible for configuration management and deployment of their various micro services.

The micro services themselves are directly deployed as Docker containers on EC2 instances in AWS. The instances run custom CentOS 7 AMIs made using Packer.

Currently no scheduling technology is being used. The company is looking into, among others, the following possibilities:

- Kubernetes
- Nomad
- Fargate

The generalised steps that need to be taken in order to deploy a fully functioning S3 Passenger environment are:

1. Deploy infrastructure (instances, load balancers, etc.) using Terraform
2. Deploy micro services using Ansible

Should the environment need to be scaled the desired instance count has to be specified in the Terraform configuration of the environment and the steps above need to be re-run.

4.3 Desired Situation

The designed solution should facilitate fast environment deployment while maintaining high levels of observability and maintainability.

Ultimately the final design should be of the complete infrastructure required for running a scheduler, the scheduler implementation and the S3 Passenger suite hosted on the scheduler.

4.4 Container Scheduling

Before beginning to compare the different container scheduling technologies we first have to define what container scheduling means.

A scheduler is a system which ensures the proper utilisation of available resources on which containers can be run. Schedulers are responsible for distributing and running containers on the available resources and ensuring their availability. Schedulers enable maximum utilisation of existing resources by densely packing workloads and smartly provisioning extra resources only when needed.

Such systems usually consist of two main components - a control cluster and a worker cluster. The first one runs all administrative jobs and logic that make the scheduling possible, whereas the second one runs the actual container deployed by users. Both clusters are deployed in high-availability configurations. Furthermore, in most cases failure of a management cluster does not result in a non-functioning deployed application, since that would only hinder the system's capability of adding, moving and removing containers, existing containers will just continue running.

The following diagram, courtesy of Mapr, illustrates the architecture of such clusters. (McDonald, 2018)

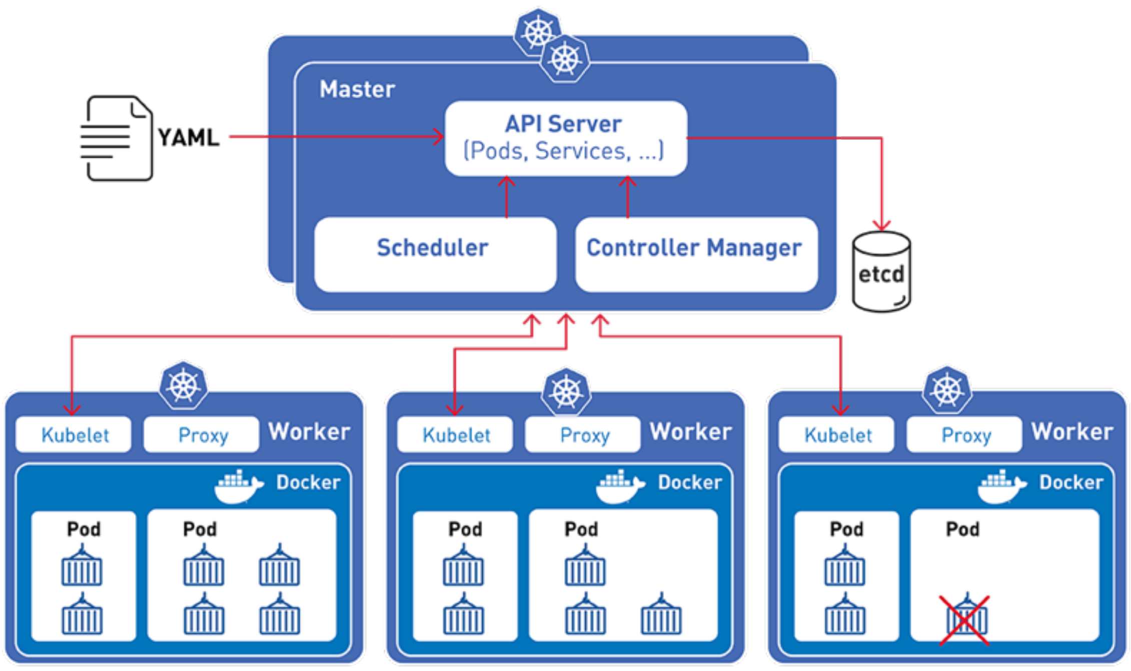


Diagram 4.4.1 - Kubernetes cluster architecture.

Generally speaking these are the following steps a scheduler takes when a new application is deployed or an old one is updated:

1. Job gets registered or updated
2. Look at the state of the system
3. Assign the job to a worker node

The following diagram illustrates how this process works in HashiCorp's Nomad. (HashiCorp, 2019b)

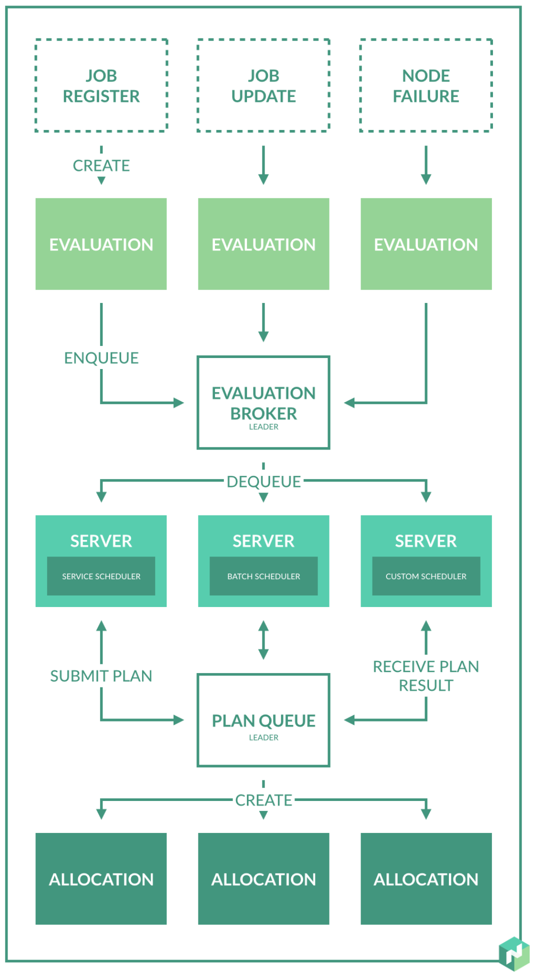


Diagram 4.4.2 - Scheduling process in HashiCorp Nomad.

Some schedulers (Kubernetes and Nomad) also support prioritising jobs. This way it can be ensured that critical applications are always running. (HashiCorp, 2019a) (Kubernetes, 2019g)

4.5 Container Orchestration

It is still debated what the precise difference is between container scheduling and orchestration. For the purposes of this project we will use the following definition:

Orchestrators work on a higher abstraction level than schedulers. They managed the scheduling of running container and their configuration based on a desired state provided by a user. Orchestrators, thus, include features like service discovery, secrets management, configuration management and others in addition to the standard container scheduling functionality.

5. Stakeholder Analysis

In this chapter the stakeholders of this project will be defined.

5.1 General List

The following is a general list of all possible stakeholders in the project. Please note that this list will be refined later in this chapter.

Stakeholder	Relevance
Student	Person executing the project.
Saxion	Grading organisation.
Sqills Product Management	Administrative team within Sqills.
Sqills Infra	Team within Sqills responsible for infrastructure development and operations.
Sqills Development Teams	Teams within Sqills responsible for the development of S3 Passenger product components.
Sqills Product Support	Team within Sqills responsible for providing support to customers using S3 Passenger.
Sqills Clients	Organisations that use S3 Passenger.

Table 5.1.1 - List of stakeholders.

5.2 Shortlist

In order to properly define which stakeholders are relevant for this project we should consider the following possible impacts that the scheduling technology might have:

- The way clients use the application
  - Introduction of visible functional changes to the end customer
- Management procedures
- Financial impact
- System architecture
- Development processes
- Infrastructural impact

Based on that information we can apply the Mendelow model. (Johnson, Scholes, & Whittington, 2008) (Mendelow, 1991) On the model we can see that the stakeholders with most interest and influence are the Sqills Infra and Product Management departments. The Sqills Development teams, although having a lot of influence, have lower interest in the matter since this is mainly a infrastructural project.

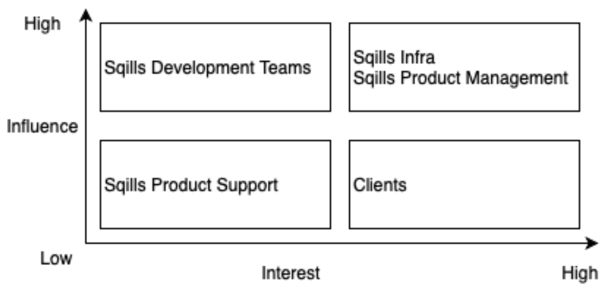


Diagram 5.2.1 - Mendelow Model.

Considering the mentioned above we can narrow down the list of relevant stakeholders to the following:

Stakeholder	Relevance
Sqills Product Management	Hight interest and high influence. Concerned with financial and administrative aspects of the implementation.
Sqills Infra	Hight interest and high influence. Concerned with the infrastructural aspects of the implementation.
Sqills Development Teams	Low interest and high influence.Concerned with potential impact on the application development.

Table 5.2.1 - Shortlist of stakeholders.

6. Requirements Analysis

In this chapter the list of requirements can be found along with the method used to compile them.

6.1 Approach

In order to properly define the requirements all relevant teams within the organisation will be interviewed. The relevance of each team can be found in the stakeholder analysis. The gathered information will then be processed and formulated into system requirements. Furthermore, relevant industry standards will be taken into account in order to ensure the validity of the requirements.

The requirements will be prioritised using the MoSCoW system:

- M - Must have
- S - Should have
- C - Could have
- W - Won't have

6.2 Questions for the teams

The following questions will be asked to the different teams:

Question	Relevant Teams
How important is it for the solution to be available as a managed service?	Infra
Should direct execution of commands on running container be possible?	Infra & Development Teams
Should the solution be immutable? (If something fails, just start a new container/server/etc. instead of attempting repairs.)	Infra
Should it be a multi-tenant solution?	Infra
Should secrets and configuration management be done by the solution or by an external tool?	Infra
How important is it for the solution to support external authentication and authorisation systems? (LDAP, etc.)	Infra
How important is it for the solution to support 3rd party service meshing tools? (Consul Connect, Istio)	Infra
Should the system be able to do service discovery or should it be able to integrate with other service discovery solutions? (Consul)	Infra
Should the system be able to do secrets management or should it be able to integrate with other secrets management solutions? (AWS SSM Parameter Store, Vault, etc.)	Infra
How important is it for the solution to support blue/green deployments?	Infra & Development Teams
Should the solution be able to support A/B testing?	Infra & Development Teams
Should the solution be open source?	Infra & Product Management
What are the acceptable costs of running the solution?	Infra & Product Management

Table 6.2.1 - List of questions for the teams.

6.3 Requirements

In this chapter you can find a list of the requirements that have been defined. These are defined based on the interviews conducted with the different teams and relevant industry best practices and standards. Furthermore, the requirements are prioritised using the MoSCoW system.

Requirement #	Description	Priority
1	Solution should support Docker containers	M
2	Solution should support network connectivity to external services (both within an AWS VPC and public internet)	M
3	Solution should be hostable on AWS	M
4	Solution should have native support for AWS managed services (such as ALB)	M
5	Solution should be Integrateable with current monitoring and logging systems	M
6	Deployed apps should be able to run even if a host or the whole management cluster is down	M
7	Solution should not bring the hosting costs higher than the current ones	M
8	Solution should be simple and it should not bring unnecessary deployment, configuration and management complexity	S
9	Solution should be open source	S
10	Solution should be deployable using infrastructure-as-code approaches (Terraform)	S
11	Solution should be easily scalable	S
12	Solution should be integrateable with LDAP for authenticating and authorising system administrators	S
13	Managed services should be utilised as much as possible	C
14	Solution should support configuration management	C
15	Solution should support secrets management	C
16	Solution should support service discovery	C
17	Solution should support 3rd party service mesh solutions (such as Consul Connect and Istio)	C
18	Solution should support A/B testing	C
19	Solution should support isolated environments	C
20	Solution should support dynamic port allocation	W

Table 6.3.1 - List of requirements.

7. Research

In this chapter you can find the information about the possible technologies that can be used in this project.

7.1 Approach

The research has been done using the following two methods:
- Literature research
- Testing

Firstly, the literature research part focuses on gathering information from sources like articles, papers and official documentation. Secondly, using by building and testing small proof of concepts the information gathered in the previous steps can be verified and caveats can be more easily discovered. By combining the data from both steps we can build a verified overview for each of the researched technologies.

For each technology the following aspects are researched:

Aspect	Description
Running applications	How can applications be run using the researched technology?
Exposing applications	How can applications be made accessible on internal and external networks using the researched technology?
Configuring applications	How can applications be configured using the researched technology? (ENV variables, configuration files, etc.)
Securing applications	How can applications be run securely using the researched technology? (Port management, secrets management, etc.)
Deploying and managing on AWS	How can the researched technology be deployed and managed on AWS?
Pros and cons	What are the pros and cons of using the researched technology.
Requirements evaluation	Which requirements does the researched technology fulfil?

Table 7.1.1 - List of research aspects.

7.2 Kubernetes

Kubernetes is probably the most famous container scheduling technology on the market. It has numerous advanced features for deployments, network management and service discovery. Furthermore, it has native out-of-the-box support for AWS services such as the ALB. In addition to all of that AWS provides a managed Kubernetes service which makes running it even easier.

7.2.1 Running Applications

Before we go on to find out how applications can be deployed we need to understand what the lower-level components of Kubernetes are. These components are responsible mainly for the day-to-day running of containers.

The various architectural patterns for deploying applications (Burns, Oppenheimer, & Google, 2016a) in containers are a topic that's discussed in more detail in chapter 8.7. Here we will focus specifically on the way Kubernetes runs applications.

The most basic component of Kubernetes is the **pod**. A pod can be seen as a space of shared resources, usually meant for one container. There are also cases when multiple containers run in one pod, for example sidecar proxies for service meshing or any other closely coupled software necessary for the functioning of the main container. (Kubernetes, 2019h)

Controllers, the next level of abstraction in Kubernetes, specify the way a specific job or application should run. Some controllers, such as the Replica Set (a controller that maintains the number of running pods) are not directly managed by the user, but by other higher-level controllers.

Depending on the type of workload, Kubernetes offers, among others, the following deployment types:

Type	Description
Deployment	An application with specific configuration.
Daemon Set	An application that is deployed on all instances. Usually used for management services.
Job	A job that needs once and is deleted afterwards.

Table 7.2.1.1 - List of Kubernetes deployment types.

Using a Deployment is the most standard way of deploying application on Kubernetes. The Deployment contains all configuration needed for an application to work:
- Volumes
- Environment Variables
- Secrets
- Replicas

The Deployment practically specifies what the desired state of the application should be. Kubernetes then creates Replica Sets and Pods and manages them.

In the Deployment the desired replicas can be specified. Then Kubernetes deploys the desired amount of replicas. This can be configured for replicas to always be deployed on different instances and/or availability zones.

As seen above, the Deployment is a high-level declarative controller which is managed by the user. The Deployment generates a Replica Set for the current configuration revision. The Replica Set is the one responsible for deploying the pods and making sure they are running properly.

Graphically, the abstraction level looks like this:

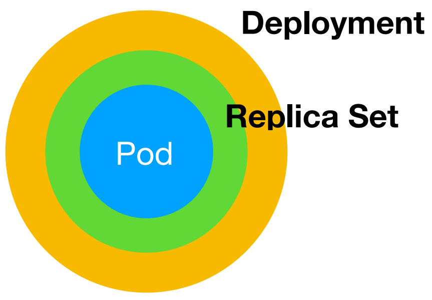


Diagram 7.2.1.1 - Kubernetes workload types abstraction levels.

7.2.2 Exposing Applications

After deploying an application it will run, but it will not be accessible from other applications and the outside. In order to expose and application a Service has to be registered.

A Service is a higher level of abstraction which ties into the way networking works in Kubernetes. Services are not as disposable as Pods and Replica Sets. A service creates an endpoint through which pods can be exposed. It is usually defined only once and the pods get registered to it.

A service can also automatically provision an ALB which exposes it to the internal networking of a VPC or the internet.

It is important to notice that services in Kubernetes operate on level 4. More advanced routing on level 7 can be done using the Ingress resource.

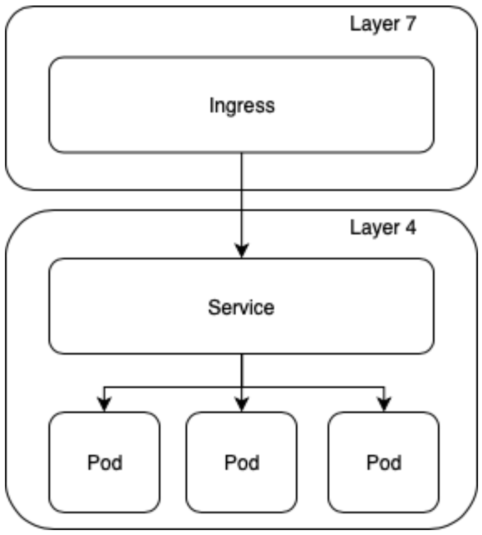


Diagram 7.2.2.1 - Kubernetes ingress and service flows.

7.2.3 Configuring Applications

Applications configurations can be stored in a Config Map. If some configuration is the same across multiple applications in an environment that configuration can be stored in one Config Map and then referenced in the Deployment.

7.2.4 Securing Applications

Kubernetes has a Secret resource in which sensitive data can be safely stored. This resource can then be referenced in a Deployment. The data from it can be either exposed as environment variables our mounted as a file in the container.

Secrets can also contain credentials for private Docker registries which Kubernetes can use to pull images from. In AWS IAM roles can be used for that purpose instead.

7.2.5 Deploying and Managing on AWS

There are currently two main ways of running Kubernetes on AWS - self-managed and fully-managed.

Picking the self-managed option means that the company would have to manually deploy and manage all Kubernetes components - from the master cluster and its persistent data storage to the worker nodes. Although not impossible, this option is very time-consuming and risky due to Kubernetes' complicated nature.

This is why, for the sake of simplicity, high-availability and stability, most companies go for managed Kubernetes solutions. Picking the fully-managed solution means that the company will utilise AWS' EKS (Elastic Container Service for Kubernetes). This service provisions a fully-managed management cluster in a chosen by the user VPC. From there on users can deploy their own worker nodes and connect them to the management cluster. AWS also provides bootstrapping templates and EKS-optimised AMIs which users can use as well.

7.2.6 Pros and Cons

Pros	**Cons**
Extensive functionality	Not a simple platform -> big learning curve
Extensive list of supported integrations	Not as easy to implement as other solutions
Backed by most big technology companies -> Guarantees project stability	
Extensive community support	
Available as managed service on AWS	
Natively integrated in Docker for Desktop for local testing	
Integrates with the company's existing tooling (Terraform, Ansible, etc.)	

Table 7.2.6.1 - List of Kubernetes pros and cons.

7.2.7 Requirements Evaluation

Requirement #	Evaluation	Requirement met?
1	Kubernetes has out-of-the-box support for Docker containers.	V
2	The Kubernetes networking components support routing both internal and external traffic.	V
3	Kubernetes can be hosted in AWS both as managed service or self-managed running on EC2 instances.	V
4	Kubernetes has out-of-the-box support for AWS managed services.	V
5	Kubernetes can be configured to export metrics and logs to the company's current monitoring and logging systems.	V
6	Apps running on the Kubernetes worker nodes will continuing running unchanged despite the status of the management cluster.	V
7	Being a complex system, requiring a decent amount of resources to run, it is possible that it will bring the hosting costs higher. However, since it is the scheduler's job to utilise available resources as efficient as possible it is more than likely that the overall costs for running Kubernetes will be lower than the current environment setup.	~
8	Kubernetes is not a simple system to understand. The learning curve is quite steep and some of the abstraction levels may not be properly understood until one has more experience with them. After the learning period however, deploying, configuring and managing Kubernetes becomes quite easy since the platform is designed to automate a lot of those tasks.	~
9	Kubernetes is an open source project.	V
10	Both managed and self-managed Kubernetes can be deployed on AWS using Terraform.	V
11	Kubernetes' cluster-autoscaler component makes it easy to scale the worker nodes.	V
12	Kubernetes can be connected to LDAP for system users management.	V
13	AWS offers Kubernetes as a managed service.	V
14	Kubernetes has out-of-the-box support for configuration management.	V
15	Kubernetes has out-of-the-box support for secrets management.	V
16	Kubernetes has out-of-the-box support for service discovery.	V
17	Istio supports Kubernetes for service meshing.	V
18	A/B testing can be done if a proper service mesh is implemented.	~
19	Kubernetes has support for namespaces, which enables the isolated deployment of multiple environments on one cluster.	V
20	Kubernetes has out-of-the-box support for dynamic port allocation.	V

Table 7.2.7.1 - Kubernetes requirements evaluation.

7.3 Nomad

Nomad is a HashiCorp scheduler build upon the principles of Google's Borg. (HashiCorp, 2019c) Nomad, unlike Kubernetes, does not include extra functionality such as service discovery and secrets management. It does, however, integrate with other HashiCorp software for these types of services.

7.3.1 Running Applications

Nomad supports various application types (referred to as Task Drivers):

- Docker containers
- Raw command execution on hosts
- Rkt containers

For the purposes of this project we will be focusing only on the Docker Task Driver.

Nomad supports three different scheduler types (comparable to the Kubernetes deployment types).

Type	Description
Service	An application that is run on the cluster.
Batch	A job that needs to run a set amount of times on the cluster.
System	An application that needs to run on all possible clients defined.

Table 7.3.1.1 - List of Nomad scheduler types.

In Nomad there are three abstraction levels of a job configuration - the job itself, a group and a task. The table below explains the differences between the three.

Type	Description
Job	Declarative description of one or more tasks that should be run.
Group	A set of tasks that should run on the same Nomad client.
Task	A running container or other supported deployment type.

Table 7.3.1.2 - List of Nomad job abstractions.

If Consul has been configured a task can also be registered there for service-discovery purposes.

The job definition is usually written in an HCL or JSON file which then can be used to deploy the job using the Nomad CLI or UI.

7.3.2 Configuring Applications

Applications can be configured by using environment variables and/or mounting volumes with configuration files that can be read by the running application. These can be defined directly in the job configuration file.

7.3.3 Securing Applications

If Vault has been configured tasks can be configured to get sensitive data, such as passwords and application tokens, from it.

7.3.4 Exposing Applications

Unlike Kubernetes and Fargate, Nomad does not have out-of-the box support for AWS ALBs and NLBs. This means that those have to be configured manually per job. This can be somewhat automated with Terraform.

Nomad does support load balancing with Nginx, HAProxy or Fabio in combination with Consul. The mentioned above software can be configured to proxy to the jobs running on Nomad by resolving the DNS name provided by Consul. Those can then be placed behind an ALB.

7.3.5 Deploying and Managing on AWS

Currently AWS does not offer any managed Nomad solutions. This means that the company would be fully responsible for deploying and managing a Nomad cluster. Usually, for service-discovery and configuration purposes, Nomad is deployed together with Consul. This automatically means that the company would have to manage both of these solutions manually.

7.3.6 Pros and Cons

Pros	Cons
Simple to implement and use	Limited community support compared to other open source projects
Integrates out-of-the-box with other HashiCorp products	Limited support for external integrations (AWS ALBs, etc.)
Integrates with some of the company's existing tooling (Terraform)	Not offered as managed service by AWS.

Table 7.3.6.1 - List of Nomad pros and cons.

7.3.7 Requirements Evaluation

Requirement #	Evaluation	Requirement met?
1	Nomad has out-of-the-box support for Docker containers.	V
2	Nomad supports routing both internal and external traffic.	V
3	Nomad can be hosted in AWS only self-managed running on EC2 instances.	V
4	Nomad does not provide support for other AWS managed services.	X
5	Nomad can be configured to export metrics and logs to the company's current monitoring and logging systems.	V
6	Apps running on the Nomad worker nodes will continuing running unchanged despite the status of the management cluster.	V

Requirement #	Evaluation	Requirement met?
7	It is highly probable that running Nomad will raise the monthly charges both in hosting (since other services such as Consul and Vault would need to be deployed alongside it) and in man-hours (since it is not provided by AWS as a managed service, which will result in more time spent managing Nomad).	~
8	Nomad by itself is a simple solution that is relatively easy to deploy. Building a production cluster, however, can get complicated since additional components (Consul, Vault, etc.) are required. This adds more complexity to the deployment and management of the platform.	~
9	Nomad is an open source project.	V
10	Nomad clusters can be deployed on AWS using Terraform.	V
11	Nomad worker nodes can be put in AWS ASGs where scaling can be trigged using CloudWatch events.	V
12	User management via LDAP can be done only via Vault.	~
13	AWS does not offer Nomad as a managed service.	X
14	Configuration management can be done via Consul.	~
15	Secrets management can be done via Vault.	~
16	Service discovery can be done via Consul.	~
17	Consul Connect supports Nomad for service meshing. Istio also has limited support for Nomad (in combination with Consul).	V
18	Nomad has out-of-the-box support for A/B testing.	V
19	Nomad has out-of-the-box support for canary deployments.	V
20	Nomad has out-of-the-box support for dynamic port allocation.	V

Table 7.3.7.1 - Nomad requirements evaluation.

7.4 ECS & Fargate

ECS is a managed solution by AWS which allows users to run containers either on EC2 instances or in a serverless way using Fargate. A standard ECS cluster comprises of EC2 instances running in the user's account. These instances are place in an autoscaling group which allows ECS to scale worker nodes in and out depending on the load.

Fargate is a solution from AWS based on their current ECS platform. Using Fargate we can deploy containers and run them without managing any servers whatsoever. This makes the service one of the most simple ways of running containers in the cloud.

The way the Fargate works is essentially the same as ECS with the major difference being that with Fargate you do not have to manage ECS EC2 instances. The two services are so intertwined that AWS specifies Fargate as a different launch type for ECS tasks.

The graphic below (courtesy of AWS) shows the differences between the two services.

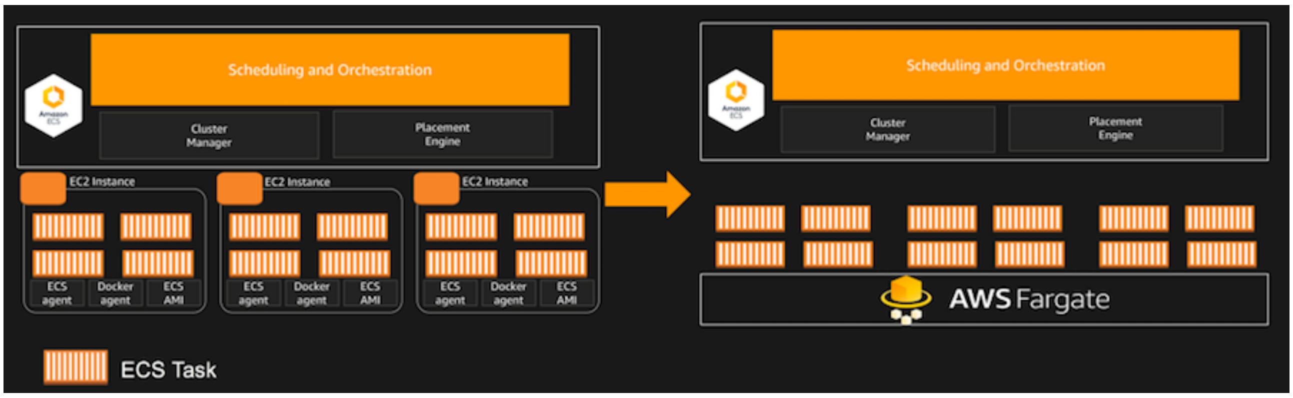


Diagram 7.4.1 - Architectural differences between AWS ECS and AWS ECS with Fargate.

As seen above, Fargate is essentially the "behind the curtains" distributer of containers on managed by AWS instances.

7.4.1 Running Applications

In order to run applications on AWS Fargate there are essentially 3 steps that need to be taken:

1. Create ECS cluster
2. Create Task Definition with Fargate launch type
3. Create Service

An **ECS Cluster** is an isolated environment of resources on which tasks can be run. If using the ECS deployment type the cluster comprises of EC2 instances which run the ECS workers and on which tasks can be deployed. If using the Fargate launch type the ECS Cluster can be seen more as a namespace. There are no specific hosts on which containers are run, that all happens behind the curtains.

A **task definition** is practically a deployment definition. In the task users can specify what they want to run, different configuration options, credentials, etc. A deployed task definition is just called a **task**.

The next level of abstraction is called a **service**. The service can compared to a deployment in Kubernetes. A service is responsible for running tasks on a cluster. In the service specification users can configure the amount of replicas needed to be run, the deployment strategy (rolling deployment or blue/green deployments with AWS CodeDeploy), networking and service discovery. Tasks can be configured to run in the public subnets of a VPC, thus getting a public IP address. It is, however, best practice to run tasks in a private subnet and expose them via a load balancer. This makes sure the load is properly distributed and makes DNS management easier because there will only be one endpoint for contact. (Note: load balancers get multiple IP addresses based on the amount of subnets they're deployed in.)

7.4.2 Exposing Applications

Applications can be exposed in multiple ways. As already mentioned in chapter 7.4.1 tasks can have an external IP or be put behind a load balancer.

Tasks that are put in the public subnet of a VPC can get public IPs and be reachable from the outside world. In that case communication goes through the VPCs Internet Gateway.

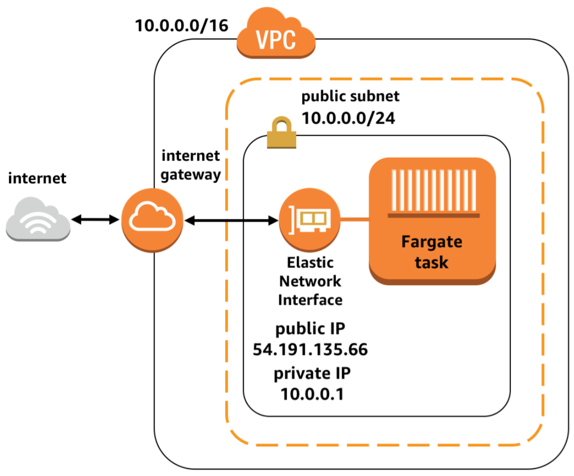


Diagram 7.4.2.1 - Fargate tasks in a public subnet.

Usually tasks are run in the private subnets of a VPC and placed behind a load balancer. This ensures that containers can't directly be accessed from the outside, it also makes rolling deployments possible when more that 1 replicas are deployed. The load balancers can then just route the traffic to the running replicas while the rest are being deployed. In this case ingress traffic goes through the load balancer and is routed to the Fargate task, the egress goes through the NAT Gateway.

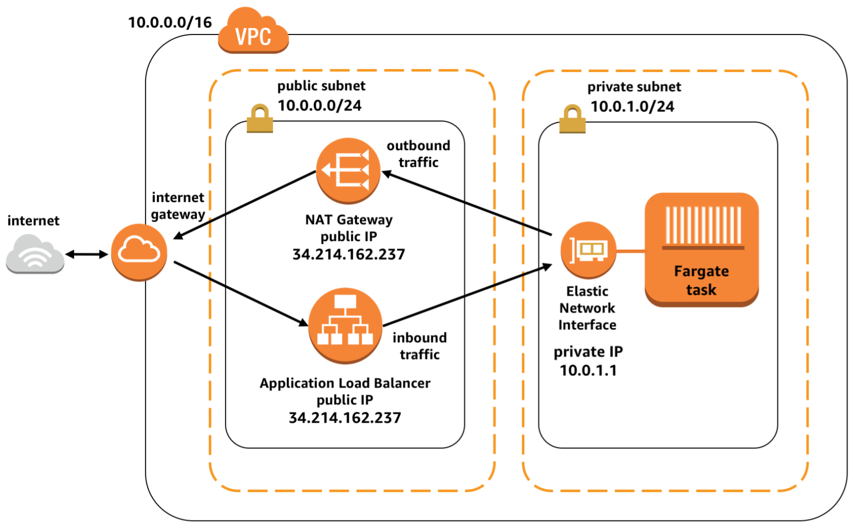


Diagram 7.4.2.2 - Fargate tasks in a private subnet with a public ALB.

7.4.3 Configuring Applications

As mentioned before in ECS the **task definition** is where the configuration for a running service is. In it users can configure, among others, environment variables, volumes, resource limits and logging options. A task definition is defined in JSON and can be created using the AWS CLI or the AWS' web UI.

7.4.4 Securing Applications

Sensitive data stored in the AWS Systems Manager Parameter Store or in the AWS Secrets Manager can be configured to be passed as ENV variables to the running task.

7.4.5 Pros and Cons

Pros	Cons
Easy to implement	Proprietary -> Vendor lock in AWS
Integrates with other AWS services	Does not integrate with 3rd party Service Discovery and Service Meshing solutions
No server management necessary in the case of Fargate	
Integrates with the company's existing tooling (Terraform, Ansible, etc.)	
Supported by AWS -> Guarantees project stability	
Is an AWS managed service	

Table 7.4.5.1 - List of ECS & Fargate pros and cons.

7.4.6 Requirements Evaluation

Requirement #	Evaluation	Requirement met?
1	ECS & Fargate have out-of-the-box support for Docker containers.	V
2	ECS & Fargate support routing both internal and external traffic.	V
3	ECS & Fargate are only available as AWS managed services. ECS note - the worker nodes are still the responsibility of the end user.	V
4	ECS & Fargate have out-of-the-box support for AWS managed services.	V
5	ECS & Fargate have out-of-the-box support for CloudWatch monitoring and logging. These can be connected to the company's current monitoring and logging solutions.	~
6	Apps running on the ECS worker nodes will continuing running unchanged despite the status of the management cluster. Apps running on Fargate will continuing running since there are no management and worker nodes. The service is fully managed by AWS.	V

Requirement #	Evaluation	Requirement met?
7	There is a high chance that ECS & Fargate will decrease the costs of running a customer environment, not only because they are offered as managed services by AWS (which results in less man hours for manual management), but also because the general usage costs are low. Furthermore, for deployments running on Fargate users do not have to pay for EC2 instances, since they are not even used.	V
8	ECS and especially Fargate are simple solutions to deploy and manage.	V
9	ECS & Fargate are AWS proprietary solutions.	X
10	ECS & Fargate can be deployed on AWS using Terraform.	V
11	ECS worker nodes can be put in AWS ASGs where scaling can be trigged using CloudWatch events. In Fargate there are no management or worker nodes to be managed.	V
12	ECS & Fargate can not be connected to LDAP for system users management.	X
13	ECS & Fargate are only available as AWS managed services.	V
14	ECS & Fargate have out-of-the-box support for configuration management.	V
15	ECS & Fargate support secrets management via AWS Secret Manager and SSM.	~
16	ECS & Fargate have out-of-the-box support for service discovery.	V
17	No 3rd party tools support ECS & Fargate for service meshing.	X
18	A/B testing can be done with Route 53 weighted records.	~
19	Environments can be isolated from each other by running them on separate ECS or Fargate clusters.	V
20	ECS & Fargate have out-of-the-box support for dynamic port allocation.	V

Table 7.4.6.1 - ECS & Fargate requirements evaluation.

7.5 Docker Swarm Mode

Docker also have their own scheduler implementation. They have, in fact, tried multiple times to come with a technology that can compete with the likes of Kubernetes. Docker Swarm was the original implementation, later developed into Docker Swarm Mode. There is also Swarm Mode and Kubernetes management functionality in Docker EE.

This chapter focuses exclusively on Docker Swarm Mode on Docker Community Edition since using open source products is one of the must requirements of this project.

Swarm Mode is out-of-the-box functionality of the Docker daemon. This means that there is no extra software that needs to be installed. After installing the daemon a swarm can be initialised, nodes joined and applications can be deployed. It is also possible to initialise a Swarm on already running nodes. This means that Swarm can be directly integrated with the company's existing infrastructure.

7.5.1 Running Applications

Running applications in Swarm Mode is pretty straightforward. There are only 2 important deployment concepts - service and task.

Type	Description
Stack	A set of applications that can be deployed together. Based on docker-compose.
Service	The definition of a running application (how many replicas should there be, environment variables, etc.).
Task	The running container on a node in the swarm.

Table 7.5.1.1 - List of Docker Swarm Mode deployment types.

Services can be deployed via the command line, just like deploying a standalone container. Stacks, on the other hand, can be defined in a docker-compose file, but instead of deploying it on a single machine with docker-compose it will be deployed as distributed services in a swarm.

7.5.2 Exposing Applications

Swarm Mode does not have any support for AWS ALBs or other centralised ways of exposing applications. Ports can be published, just like with standard Docker containers. This means that if a service has tasks running on multiple hosts each container will expose the requested port on the host. From there the user can decide how they want to route and load balance the traffic.

7.5.3 Deploying and Managing on AWS

Essentially the deployment process will be almost exactly the same as with deploying standalone Docker EC2 instances, however in this case they need to be joined in a swarm by:

1. Making sure the required ports are open in the Security Groups
2. Initialising a swarm
3. Running the join command on the rest of the nodes

7.5.4 Pros and Cons

Pros	Cons
Easy to implement (even on existing infrastructure)	Limited community support
Integrates with some of the company's existing tooling (Terraform, Ansible, etc.)	Limited support for external integrations (AWS ALBs, etc.)
	Not offered as a managed service by AWS

Table 7.5.4.1 - List of Docker Swarm Mode pros and cons.

7.5.5 Requirements Evaluation

Requirement #	Evaluation	Requirement met?
1	Docker Swarm Mode has out-of-the-box support for Docker containers.	V
2	Docker Swarm Mode supports routing both internal and external traffic.	V
3	Docker Swarm Mode can be hosted on AWS self-managed running on EC2 instances.	V
4	Docker Swarm Mode does not provide support for other AWS managed services.	X
5	Docker Swarm Mode can be configured to export metrics and logs to the company's current monitoring and logging systems.	V
6	Apps running on the Docker Swarm Mode worker nodes will continuing running unchanged despite the status of the management cluster.	V
7	There is a high probability that the monthly costs for running a customer environment will not change since Docker Swarm Mode is relatively lightweight, easy to implement and easy to manage.	V
8	Docker Swarm Mode is a simple solution to deploy and manage.	V
9	Docker Swarm Mode is an open source project.	V
10	Docker Swarm Mode can be deployed on AWS using Terraform.	V
11	Docker Swarm Mode worker nodes can be put in AWS ASGs where scaling can be trigged using CloudWatch events.	V
12	Docker Swarm Mode can not be connected to LDAP for system users management.	X
13	AWS does not offer Docker Swarm Mode as a managed service.	X
14	Docker Swarm Mode has out-of-the-box support for configuration management.	V
15	Docker Swarm Mode has out-of-the-box support for secrets management.	V
16	Docker Swarm Mode has out-of-the-box support for service discovery.	V

Requirement #	Evaluation	Requirement met?
17	No 3rd party solution has official support for service meshing. However, it can still be achieved by implementing Consul and Consul Connect.	X
18	A/B testing can be done by deploying custom load balancers or implementing a proper service mesh.	~
19	Environments can be isolated from each other by running them on separate Docker Swarm Mode clusters.	X
20	Docker Swarm Mode has out-of-the-box support for dynamic port allocation.	V

Table 7.5.5.1 - Docker Swarm Mode requirements evaluation.

7.6 Comparison

In this chapter you can see a comparison of the technologies discussed in the research part of this document. The comparison is based on the requirements defined in the beginning of this document (chapter 6.3.). Each technology gets a score in numeric value between 1 and 5 which translates into the following:

Value	Description
🟥 1	The requirement can **not** be met out-of-the-box and there are **no** other alternatives for implementation.
🟧 2	The requirement can **not** be met out-of-the-box, however it can be implemented using non-best-practice workarounds.
🟨 3	The requirement **can** be met out-of-the-box, however it is difficult to implement.
🟦 4	The requirement can **not** be met out-of-the-box, however a workaround can be implemented easily and within best-practices and standards.
🟩 5	The requirement **can** be met out-of-the-box and is easy to implement.

Table 7.6.1 - Comparison table legend.

On the following page you can see the table of requirements and component scores.

Requirement #	Kubernetes	Nomad	ECS & Fargate	Docker Swarm Mode
1	5	5	5	5
2	5	5	5	5
3	5	5	5	5
4	5	4	5	4
5	5	5	4	5
6	5	5	5	5
7	3	3	5	5
8	3	3	5	5
9	5	5	1	5
10	5	5	5	5
11	5	4	5	4
12	4	4	1	1
13	5	1	5	1
14	5	4	5	4
15	5	4	4	5
16	5	4	5	5
17	5	3	1	2
18	3	5	4	4
19	5	1	5	3
20	5	5	5	2

Table 7.6.2 - Schedulers and orchestrators comparison.

7.7 Conclusion

Given the results of the research the recommended solution to continue forward with is Kubernetes. It does not only fully cover most “must” requirements, the platform goes further to cover a big part of the “should” and “could” requirements as well. The “must” requirements that are not completely covered can still be met with some extra configuration.

Furthermore, Kubernetes is by far the most mature platform on the market, with the biggest user base and community. The future of the project looks promising and stable, which we can derive from the amount of companies and communities behind the project.

Kubernetes is also provided as a managed service by AWS which will significantly decrease the man-hours needed for setting up and managing clusters.

It can be said with confidence that Kubernetes is capable of addressing the company’s problems (described in chapter 3). The platform enables the teams to deploy and manage environments faster and more efficient than ever before. It raises the abstraction level of infrastructure management within the organisation and it facilitates automation processes.

Because of these reasons the rest of this report will be focused on the design of a Kubernetes environment for running the S3 Passenger platform.

8. Design

8.1 Approach

Based on the information gathered from the research a choice on which technology will be used will be made.

Afterwards an iterative approach will be taken when it comes to developing the design. Building a prototype and incrementally updating it and testing it ensures a flexible and agile way to develop a validated design.

The built prototype has multiple purposes. It serves as a platform for developing the design and eventually providing proof of the feasibility of it. It will also serve as a test environment where the design can be validated.

When the final version of the design is complete the prototype created during the design process will be used for the design evaluation.

The design is divided between the following aspects:

- Architecture
  - Global
  - Application landscape
- Networking
  - Inside the VPC
    - Between instances
    - SGs
    - NAT
    - Load Balancing
  - Inside the cluster
    - Between applications
    - Services
    - Service mesh
    - Proxies
  - External
    - Egress
      - To other AWS services
      - To the internet
    - Ingress
      - From the internet
- Storage
  - Persistent
    - Databases
    - Persistent volumes
    - EFS
  - Caching
    - ElastiCache/Redis
- Monitoring and Logging
- CI/CD

The design chapter will begin with a Kubernetes deep dive where all relevant to the design components are explained.

8.2 Kubernetes Deep Dive

This chapter aims to give an overview of all relevant Kubernetes components, their purpose, how they are configured and how do they fit in the whole system.

8.2.1 Components Overview

In the table below you can find an overview of the Kubernetes **master** components. (Kubernetes, 2019d)

Component	Purpose
Kube API Server	Serves the Kubernetes API.
ETCD	Key-value store.
Kube Scheduler	Responsible for scheduling pods on nodes.
Kube Controller Manager	Global manager for all Kubernetes controllers.
Cloud Controller Manager	Responsible for cloud-specific controller processes.

Table 8.2.1.1 - List of Kubernetes master components.

In the table below you can find an overview of the Kubernetes **node** components. (Kubernetes, 2019d)

Component	Purpose
Kubelet	An agent that runs on every server of the Kubernetes cluster.
Kube Proxy	Responsible for maintaining the network rules and forwarding.
Docker	The container runtime.

Table 8.2.1.2 - List of Kubernetes node components.

The diagram below, courtesy of the official Kubernetes documentation, illustrates the relationship between these components. (Kubernetes, 2019j) As shown below, the Kube API Server is the central point where everything connects to. The API Server persists it data in ETCD. The Cloud Controller Manager connects to the APIs of the the user's chosen cloud platform.

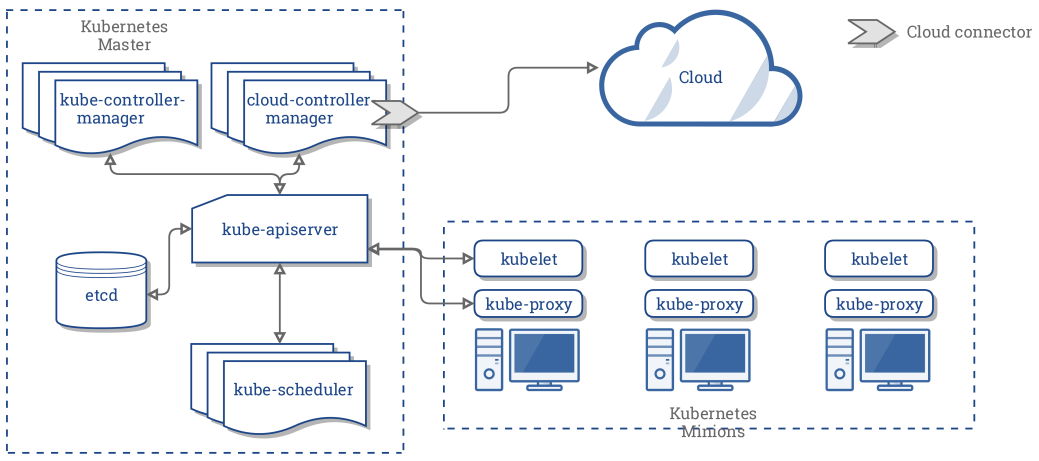


Diagram 8.2.1.1 - Relationship between Kubernetes master components.

8.2.2 Controllers Overview

In the table below you can find an overview of the Kubernetes controllers.

Controller	Purpose
Node Controller	Monitors the state of the nodes in the Kubernetes cluster.
Replication Controller	Responsible for ensuring that the correct amount of pods are running as specified in the relevant Replication Controller configuration.
Endpoints Controller	Responsible for adding pods to services.
Service Account & Token Controllers	Creates default accounts and API tokens for all namespaces.

Table 8.2.2.1 - List of Kubernetes controllers.

8.2.3 Kubernetes API

The Kubernetes API is an extremely vast subject which due to time, space and project scope constraints will not be discussed extensively in this document.

The most important takeaway about this is that the Kubernetes API is the central point of all actions in Kubernetes. Every single resource, specification and action is essentially part of the API. Resources are made and updated via the API, metrics are acquired via the API, authentication goes through the api, etc.

There are 3 main API groups that separate API endpoints based on their maturity - alpha, beta and stable. Alpha and beta groups can be explicitly enabled or disabled on the API server.

8.2.4 Authentication & Authorisation

Authentication

There are two types of users in Kubernetes - service accounts and normal users. The first type are managed by Kubernetes itself and are used internally in the cluster for authenticating services that need access to the Kubernetes API. The second type are human users that are managed by an external identity provider. (Kubernetes, 2019b)

There are 3 different authentication possibilities in Kubernetes:

- X.509 client certificates
- Tokens
  - Static token files
  - Bootstrap tokens
  - Service account tokens
  - OpenID Connect tokens
- Static password files

The following table provides an explanation about how each of those methods work:

Method	Explanation
X.509 client certificates	The Kubernetes API server can be given a CA to trust. Then users can log in with certificates signed by the trusted CA.
Static token files	The Kubernetes API Server can be given a static list of bearer tokens that it will trust. Clients can then pass these tokens in the headers of their requests.

Method	Explanation
Bootstrap tokens	These tokens, generated by Kubernetes itself, are stored as Kubernetes secrets and are meant to be used while bootstrapping a new cluster.
Service account tokens	Automatically generated tokens for service accounts.
OpenID Connect tokens	Kubernetes can be configured to use a 3rd party identity provider (that supports OpenID Connect). When users authenticate with their chosen provider a JWT token is generated which is used for authentication with the Kubernetes cluster.
Static password files	The Kubernetes API Server can be given a static list of users and passwords. Clients can then pass the username and password in the headers of their requests.

Table 8.2.4.1 - List of Kubernetes authentication methods.

Authorisation

When a user or a service account makes a request the first needs to be authenticated. If that action succeeds the user's request needs to be authorised. Kubernetes currently uses Role-based Access Control (RBAC) for authorising users and service accounts.

As already mentioned, Kubernetes resources are modified via the API. There are multiple modifications that can be performed on a resource. Generally speaking the following actions can be performed on Kubernetes resources:

Action	API Action
Retrieve	Get
List	List
Continuously retrieve	Watch
Create	Create
Update	Update
Modify	Patch
Delete	Delete

Table 8.2.4.2 - List of actions that can be done on Kubernetes resources and their respective API action equivalents.

If we want to retrieve information about a specific deployment, for example, we will perform a "get" request on the deployment we want to get information about.

In Kubernetes rules can be made that specifically allow certain actions on certain resources. These rules are a resource by themselves - Roles (namespace-bound) or Cluster Roles (cluster-bound). These roles can then be assigned on specific users or service accounts by a Role Binding.

8.3 Architecture

In this chapter you can find the architectural overview of the chosen solution.

8.3.1 Global

In the following diagram a global overview of the architecture of an EKS cluster can be found.

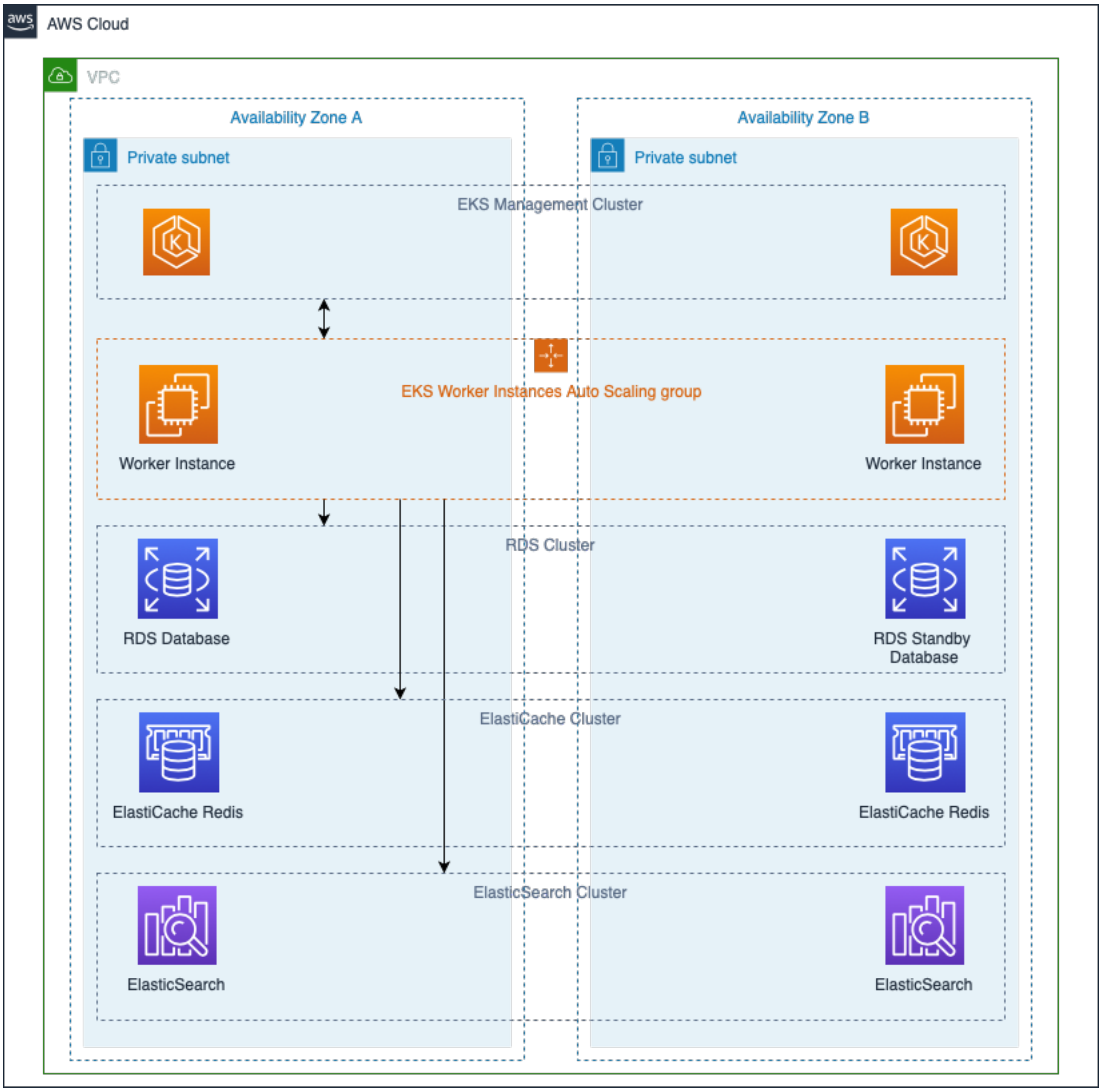


Diagram 8.3.1.1 - Global architectural overview of the Kubernetes cluster.

8.3.2 Application

For simplicity and NDA-related purposes the following sample app architecture will be used.

As seen on the diagram below requests from the internet are being handled by the ingress service. This service routes the request to the relevant micro service. There are two micro services in this example landscape. The first one (App A) is connected to a cache (Redis) and the second one (App B) is connected to a database (PostgreSQL).

This sample landscape is relatively close to the S3 Passenger landscape since it also consists of ingress services, multiple micro services, caches and databases.

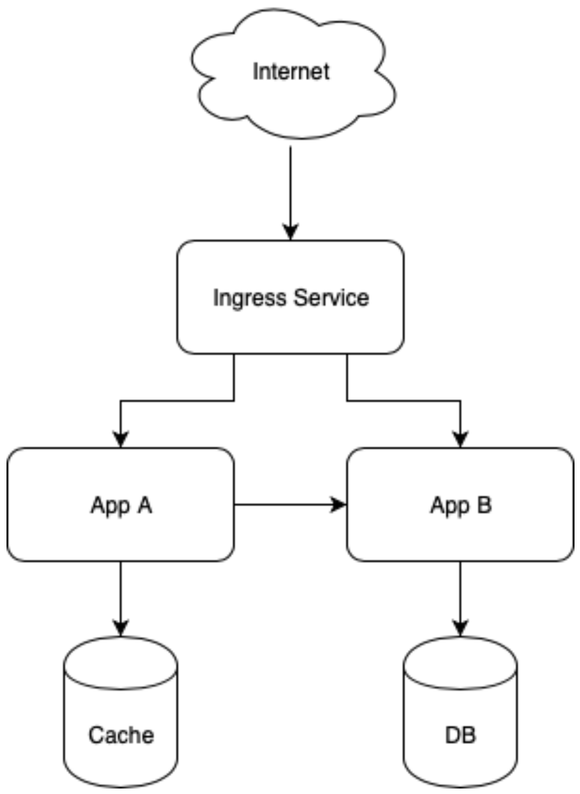


Diagram 8.3.2.1 - Sample app architecture.

8.4 Networking

In this chapter the networking design can be found. This includes application and cluster level networking.

8.4.1 VPC

In AWS the region where customer infrastructure runs is divided between multiple availability zones (usually 3). These are physically separate data centres within the same geographical region. A VPC, the network block available for a customer within a region spans all availability zones.

Customers can then configure subnets within their VPC. It is important to note that a subnet can only be in one singe region, not multiple like the VPC. It is best practice to define public and private subnets in a VPC. The difference between public and private subnets in AWS is that anything running in the public subnets will automatically get its own public IP address and traffic to the internet will be routed through the Internet Gateway. Anything running in the private subnets will only get a private IP and if internet access is required outgoing traffic will be routed through the NAT Gateway.

It is considered best practice to put all applications and other components in the private subnets and only put external load balancers in the public subnets. (Amazon Web Services, 2017)

As you can see in chapter 8.2.1 all of our infrastructure will be deployed inside the private subnets. Kubernetes can then create public load balances in the public subnets if necessary.

The diagram below provides a graphical overview of the architecture of an AWS VPC.

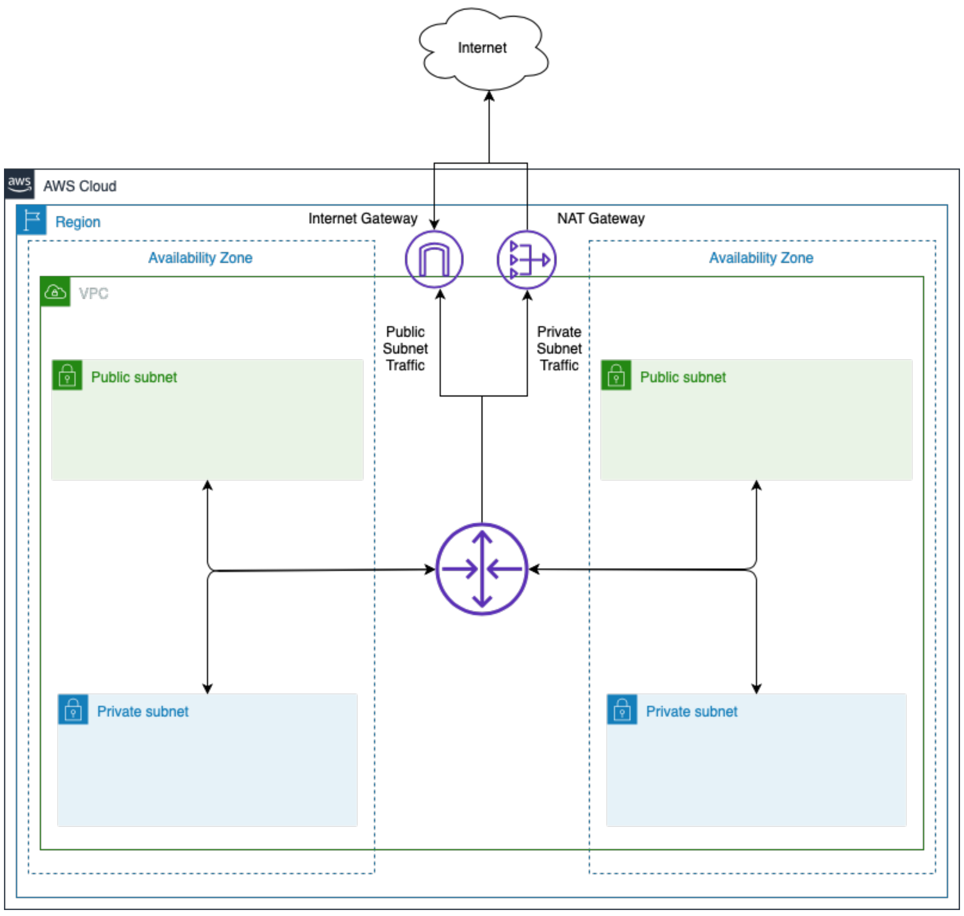


Diagram 8.4.1.1 - AWS VPC network overview.

8.4.2 Applications

As mentioned in the Kubernetes research chapter (7.2) applications communicate between each other via Kubernetes services. Each service has a DNS name which resolves to an internal cluster IP. DNS names are formed using the following pattern: `service.namespace.svc.cluster.local`. (Kubernetes, 2019e) Applications can use these DNS names instead of hard-coded IPs. Even though the DNS name resolves to one singular IP address (that of the service) Kubernetes load-balances the traffic to all possible pods registered in the service. This load balancing happens by using the local node's IP tables. The kube-proxy component is the one responsible for receiving service configuration from the Kubernetes API servers and updating the IP tables of the nodes.

The graphic below, courtesy of the official Kubernetes documentation, illustrates the way clients access services via the node's IP tables and how they are updated by the kube-proxy. (Kubernetes, 2019i)

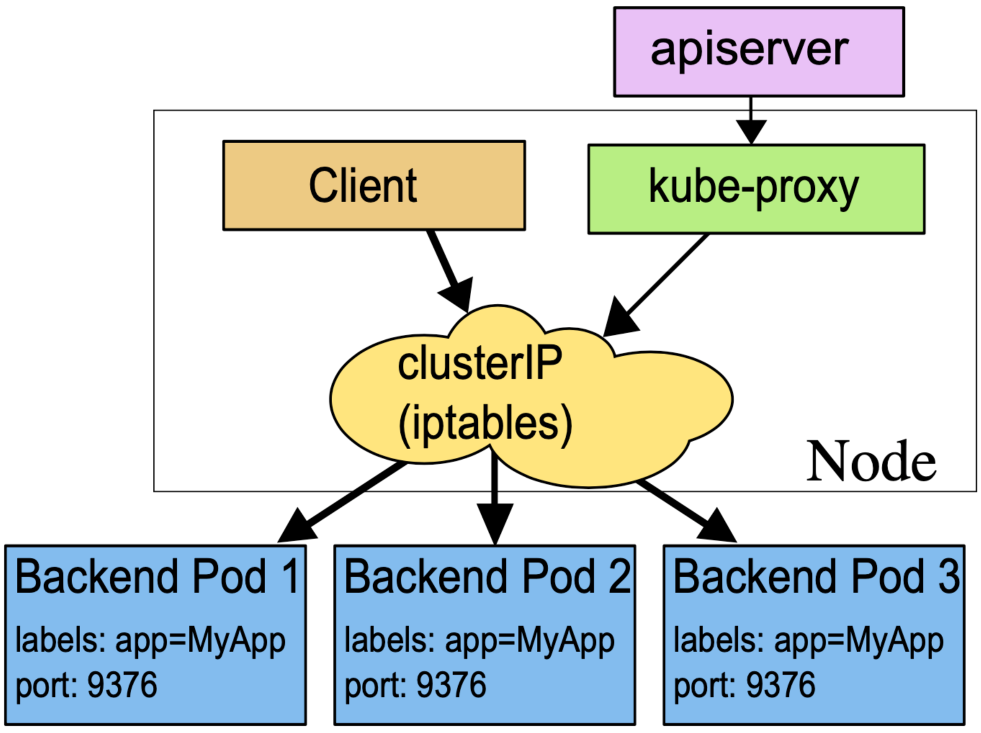


Diagram 8.4.2.1 - Kubernetes Services iptables routing.

The following graphic illustrates a call from one service to another.

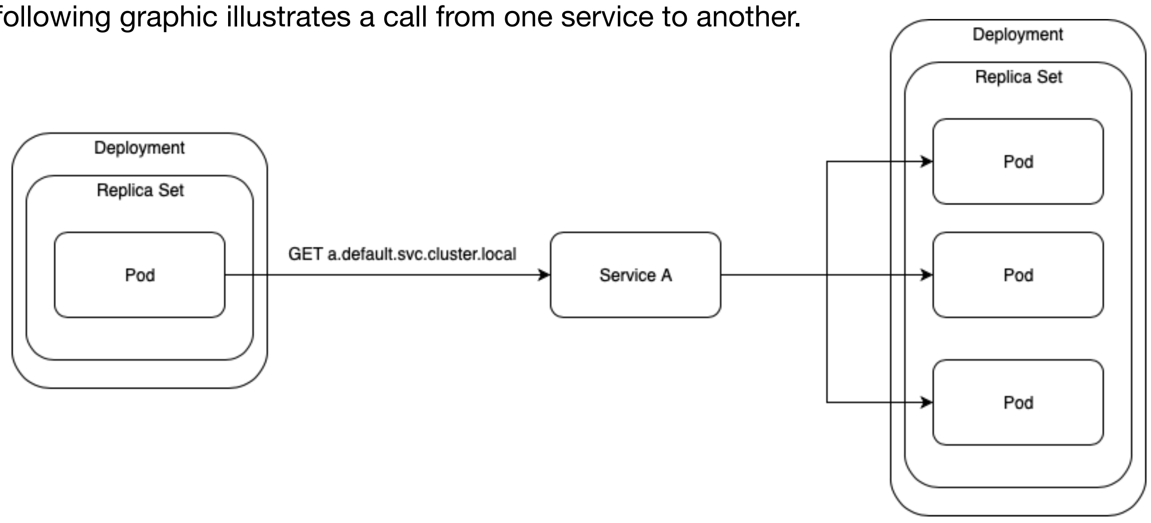


Diagram 8.4.2.2 - Communication flow between Kubernetes services.

It is important to note that Kubernetes services are a Layer 4 network resource. They can **only** do TCP and UDP load balancing. For more advanced routing and utilisation of the AWS ALBs Kubernetes provides the Ingress resource.

Kubernetes Ingresses are a Layer 7 network resource that understand HTTP traffic. This enables, among others, name-based routing. Furthermore, the Ingress can be used to deploy and configure AWS ALBs using the AWS ALB Ingress Controller. (Kubernetes Community - AWS Special Interest Group, 2019)

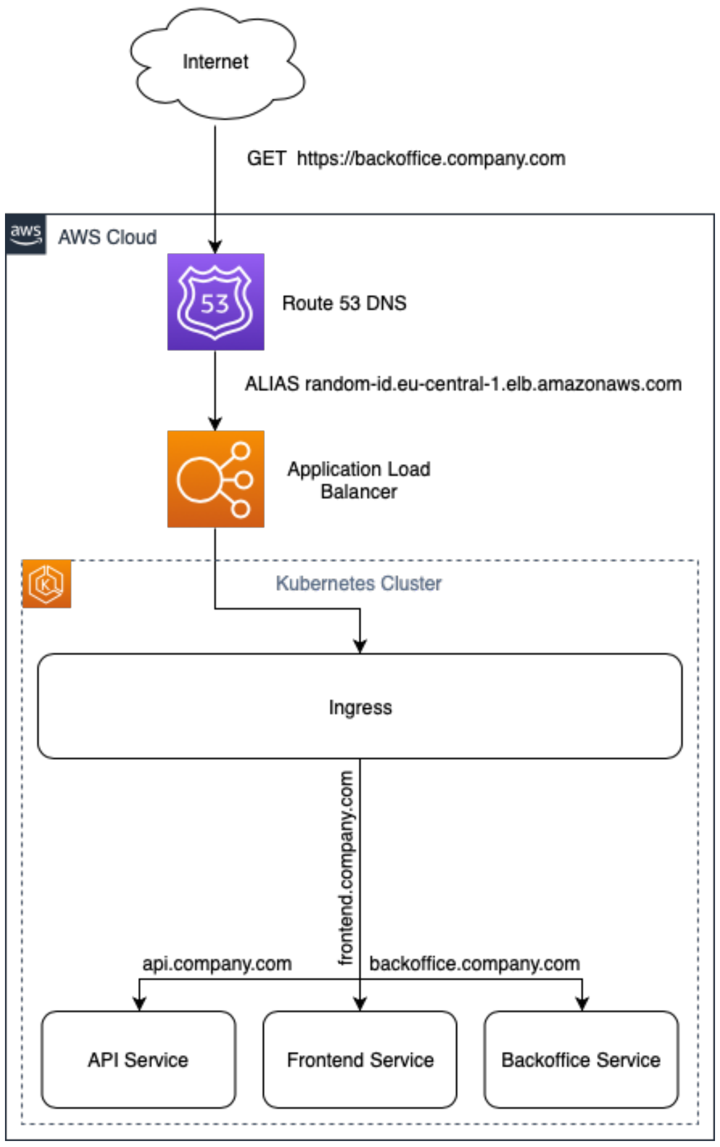


Diagram 8.4.2.3 - Network flow for routing to specific service.

8.5 Storage

There are three storage components required for the functioning of S3 Passenger:

- PostgreSQL database
- Redis session storage
- NFS share for file storage

There aren't any specific configurations required for these components, apart from the usual HA and security settings (replicas in multiple AZs, required ports open only for instances that run applications which require access to these components, etc.).

Connections to the database and cache services go directly from the S3 Passenger component that needs to use them to the designated by AWS endpoint.

NFS shares, however, need to be mounted on the EC2 instances running the S3 Passenger components that need to utilise them. In order to do this dynamically the EFS provisioner component of Kubernetes can be deployed in the cluster. This component creates the EFS storage class and is responsible for mounting NFS shares to the relevant EC2 instances and exposing them to Kubernetes as persistent volumes. (Kubernetes Community, 2019)

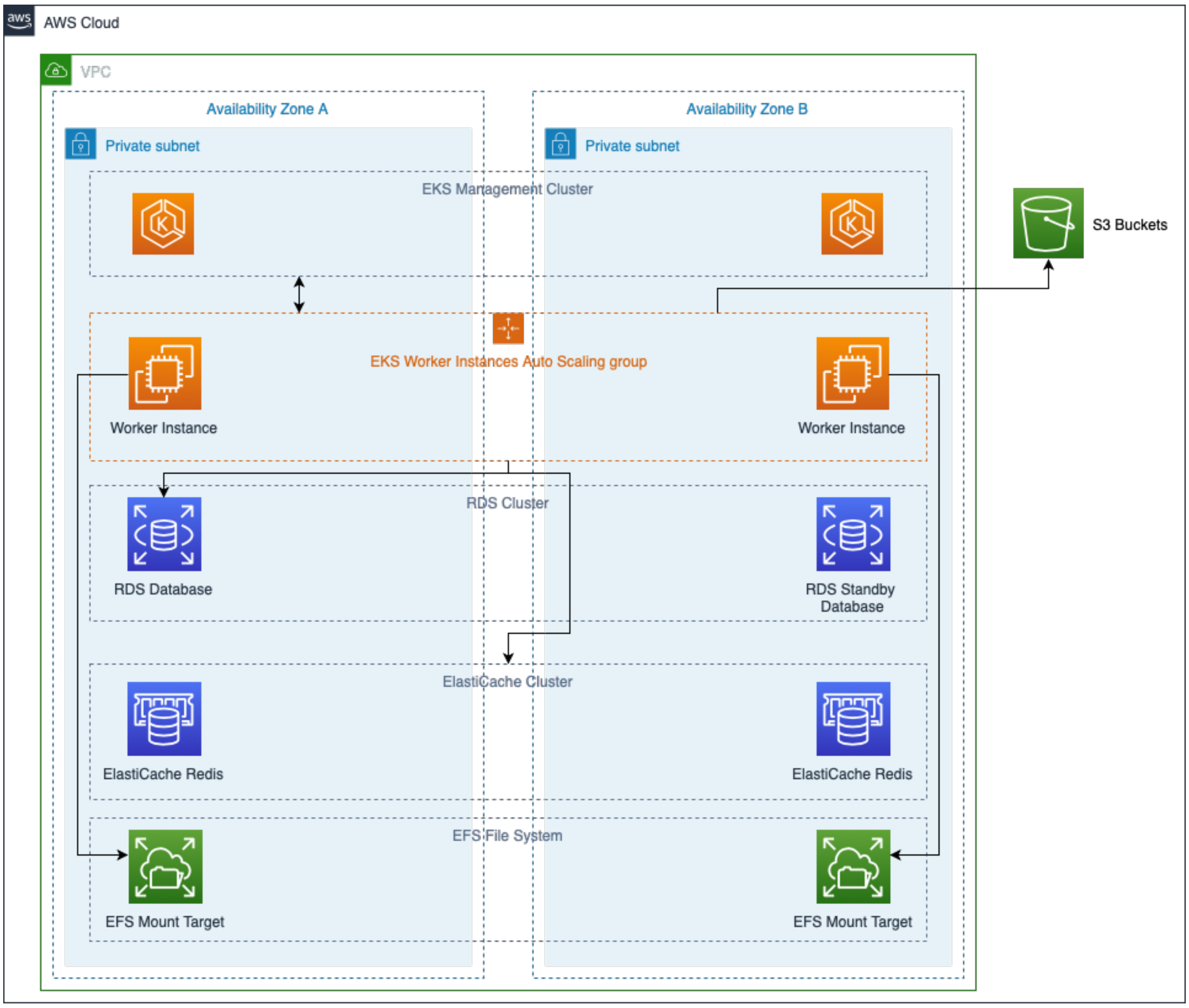


Diagram 8.5.1 - Storage overview

8.6 Monitoring & Logging

This chapter explains the monitoring and logging design of the proposed solution.

8.6.1 Monitoring

Kubernetes distinguishes between two types of monitoring pipelines - the core metrics pipeline and the monitoring pipeline. (Kubernetes Community, 2018)

As the name suggests, the core metrics pipeline is utilised by the main systems to execute tasks like scheduling and other core Kubernetes functionalities that require such metrics. The core metrics pipeline is realised by the Kubelet, the metrics server, and the metrics API. (Kubernetes Community, 2018) (Kubernetes, 2019f)

The monitoring pipeline is meant for the consumption of metrics by end users. (Kubernetes Community, 2018) Such pipeline may consist of tools like Prometheus or other 3rd party integrations for cluster metrics gathering. (Kubernetes, 2019m)

For a detailed description of the inner workings of the Kubernetes monitoring architecture please see appendix A1.

Since the core metrics pipeline is available by default on every cluster this design will focus on the monitoring pipeline, which is user-configured.

The standard monitoring pipeline in Kubernetes consists of four components - Prometheus, Alert Manager, Push Gateway and Grafana. Prometheus is the aggregator of all possible monitoring data - from the Kubernetes metrics server and Prometheus exporter daemon sets running on each node. Alert Manager, as the name suggests, is the tool that realises alerting in this pipeline. Alert Manager can be configured to send alerts if specific conditions have been reached based on the data gathered in Prometheus. The third component, Grafana, is the visualisation tool for the Prometheus data. (Prometheus, 2018)

Sqills already utilises Prometheus and Grafana for the monitoring of their current infrastructure and application landscape, thus such a system will be easily implementable in Kubernetes in conjunction with the existing ones.

The following graphic, courtesy of Prometheus, illustrates such a monitoring pipeline. (Prometheus, 2018)

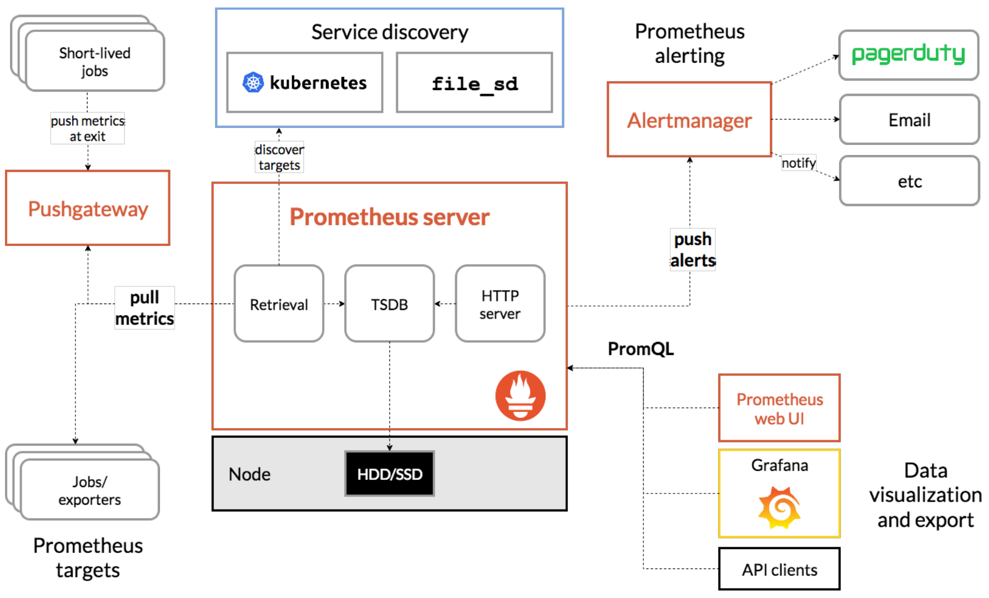


Diagram 8.6.1.1 - Prometheus monitoring pipeline.

8.6.2 Logging

There are two important log groups in Kubernetes - logs of Kubernetes components and logs of non-Kubernetes component (or the software that the user is running in the cluster).

Having insights on both of these groups is critical for proper cluster management.

Kubernetes saves all logs in JSON files on the respective nodes where the components are running. Using software like Fluentd we can scrape these logs and send them to a central database - Elasticsearch, for example. (Kubernetes, 2019l)

The standard logging solutions for Kubernetes clusters are the Elasticsearch, Logstash and Kibana (ELK) or Elasticsearch, Fluentd and Kibana (EFK) stacks. The functionality of both stacks is exactly the same, which makes choosing one of the two solutions purely based on functionality a hard task. There is, however, one very important difference when it comes to platform integration. Fluentd natively integrates with Kubernetes which allows the collection of Kubernetes-specific metadata. Furthermore, the company currently uses an EFK stack which makes the configuration of application-specific settings easier. (Giant Swarm, 2018)

Sqills also currently uses an EFK stack for the logging of their infrastructure and applications, thus such a solution will be easily implementable in Kubernetes in conjunction with the current ones.

Because of the reasons mentioned above this design will focus on implementing an EFK stack.

Fluentd offers a container which can be deployed as a daemon set in Kubernetes without a lot of extra configuration. The endpoint of the Elasticsearch cluster can be passed to the container in an environment variable. Kubernetes will the make sure a pod runs on every node in the environment. Fluentd will then start scraping the available logs and sending them to the specified Elasticsearch cluster for analysis and storage.

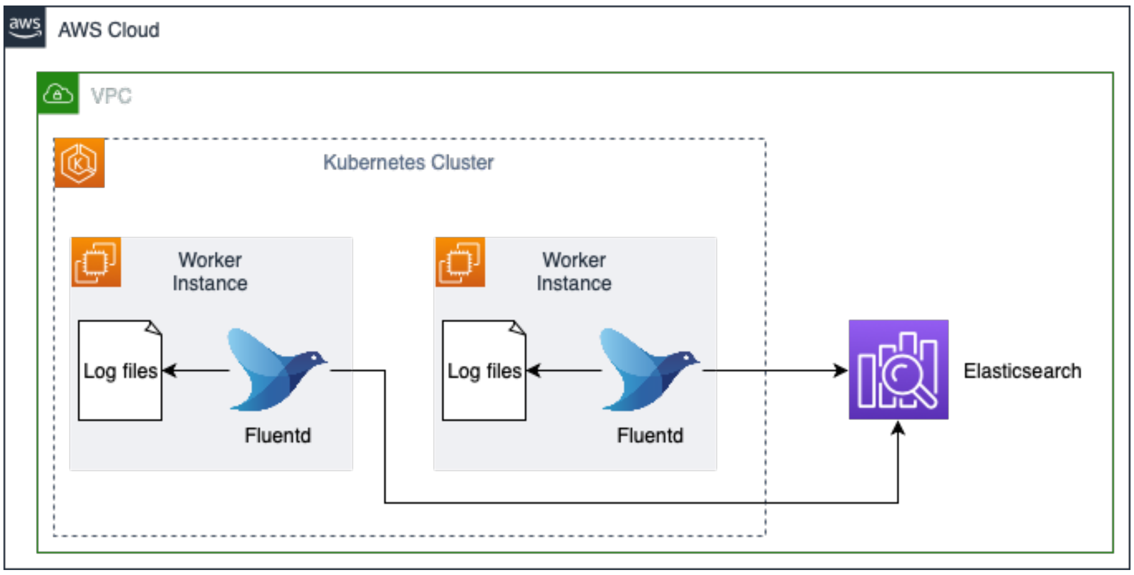


Diagram 8.6.2.1 - Logging pipeline

8.7 CI/CD

Although the general idea of the current CI/CD pipelines for deploying the S3 Passenger components can still be used a change of the tooling and steps will be required to implement it for a Kubernetes environment.

Let's take our sample application from chapter 8.2.2 as an example. We will define the following deployment procedure:

1. Create database
2. Create cache
3. Deploy app B
4. Run database migrations
5. Deploy app A
6. Deploy ingress service

Currently the company uses Ansible to deploy and configure applications. The deploy process is imperative - the steps needed to roll out an application are defined in Ansible Playbooks.

Since the different application configurations are now in Kubernetes and the desired state is defined in deployment files a new process for this declarative deployment strategy will be required.

The default deployment strategy on Kubernetes is the rolling update. (Kubernetes, 2019k) This type of deployment essentially updates one pod at a time, thus keeping the service running. (Kubernetes, 2019c)

There are, of course, many other types of deployments that can be done with Kubernetes. (Tremel, 2017)

9. Design Evaluation

There are two types of testing that will be applied to the design:
- Infrastructure testing
- Application testing

The infrastructure tests focus on Kubernetes itself, whereas the application tests focus on S3 Passenger and how it handles when run in a scheduler.

By using the principles of chaos engineering we can conduct tests that are close to real-world events (server and/or network failures, etc.) and find weaknesses in the design. (Lafeldt, 2019)

Furthermore, performance tests were conducted on the application itself which allows us to see how it behaves under load and check if Kubernetes is taking the proper steps to ensure all services are running properly.

9.1 Chaos Tests

The chaos tests, as the name suggests, were conducted at random times and intervals on random components. The cluster was rigorously tested to see how it responds to random removal of worker nodes and connection outages. The EC2 instances serving as the worker nodes for the cluster were removed at random intervals and were not manually created again.

What was observed was that Kubernetes sees the fact that nodes are missing and deployments are not running because of it. The first step it takes is evaluating the priority of deployments. As already mentioned in this document, if the priority of deployment X is higher than the one of deployment Y and deployment X is not running Kubernetes will stop deployment Y and run deployment X instead. Parallel to this process Kubernetes will also communicate with the ASG to see if the amount of desired nodes needs to be updated and if so - update the amount. In the meantime the ASG would have also already noticed the missing nodes and it wold have started new ones to reach the target amount. When the nodes have started and have registered Kubernetes will start running the rest of the deployments. This process takes in total around 8 minutes.

Test	Description	Result
Worker Node Resilience	Stopping random worker nodes at random intervals.	Kubernetes spins up new nodes and recovers within minutes.
Environment Recovery	Stopping all worker nodes at random intervals.	Kubernetes spins up new nodes and the S3 Passenger environment is fully functional within 8 minutes.

9.2 Performance Tests

Sqills has in-house performance test of the application which generate load on its components. These tests were applied on the prototype to verify that the application will continue working properly in Kubernetes.

During the performance tests it was observed that Kubernetes properly monitors the resource usage of the deployments and it scales them whenever necessary. Furthermore, the worker nodes were scaled as well so that there are enough available resources to run the pods on.

10. Conclusion

The purpose of this project is to deliver a future-proof container orchestration solution to Sqills which will enable the Infra team to host the S3 Passenger platform in a more automated, scalable and modern way. In order to achieve this multiple possible solutions were researched, tested and evaluated during the course of this project. Based on that a solution was chosen - namely Kubernetes. A design of a Kubernetes environment, capable of hosting the S3 Passenger solution, was created and tested.

In the beginning of the project a set of requirements were defined according to the wishes of the teams, company requirements and industry standards. Factors like scalability and maintainability were a top priority for the company. The requirements were prioritised using the MoSCoW method and were used as the main evaluation criteria for the research phase of the project.

After the mentioned above first stage four scheduler technologies were researched and reviewed. Based on that research it was concluded that Kubernetes is the best contender.

On the third stage of the project a Kubernetes environment capable of hosting the S3 Passenger platform was designed. All aspects have been taken into account — from the underlying infrastructure to the deployment of S3 Passenger itself.

As a final step two types of tests were conducted to ensure that the designed infrastructure meets all requirements and standards.

It can be safely concluded that Kubernetes not only meets all necessary requirements, provides a stable and modern platform for hosting applications, it also is perfectly deployable on AWS and the S3 Passenger platform can be hosted on it in a highly available manner.

References

Amazon Web Services. (n.d.). What Is Amazon EKS? Retrieved from https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html

Amazon Web Services. (2017, July). Architecture - Building a Modular and Scalable Virtual Network Architecture with Amazon VPC. Retrieved April 11, 2019, from https://docs.aws.amazon.com/quickstart/latest/vpc/architecture.html?shortFooter=true

Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J., & Google Inc.. (2016, March 2). Borg, Omega, and Kubernetes - ACM Queue. Retrieved from https://queue.acm.org/detail.cfm?id=2898444

Burns, B., Oppenheimer, D., & Google. (2016a). Design Patterns for Container-based Distributed Systems. Retrieved April 1, 2019, from https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/burns

Burns, B., Oppenheimer, D., & Google. (2016b, July 21). Container Design Patterns. Retrieved April 1, 2019, from https://kubernetes.io/blog/2016/06/container-design-patterns/

Burrows, M. (2006). The Chubby lock service for loosely-coupled distributed systems – Google AI. Retrieved from https://ai.google/research/pubs/pub27897

Casalicchio, E. (2017, May 3). Autonomic Orchestration of Containers: Problem Definition and Research Challenges. Retrieved from https://www.researchgate.net/profile/Emiliano_Casalicchio/publication/316703820_Autonomic_Orchestration_of_Containers_Problem_Definition_and_Research_Challenges/links/59c1113aaca272295a097a58/Autonomic-Orchestration-of-Containers-Problem-Definition-and-Research-Challenges.pdf

Ellingwood, J. (2017, November 29). The Docker Ecosystem: Scheduling and Orchestration. Retrieved from https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-scheduling-and-orchestration

Giant Swarm. (2018, August 16). Logging Best Practices for Kubernetes using Elasticsearch, Fluent Bit and Kibana. Retrieved May 21, 2019, from https://itnext.io/logging-best-practices-for-kubernetes-using-elasticsearch-fluent-bit-and-kibana-be9b7398dfee

Grillet, A. (2018, June 1). Comparison of Container Schedulers. Retrieved from https://medium.com/@ArmandGrillet/comparison-of-container-schedulers-c427f4f7421

HashiCorp. (2019a, January 24). Preemption. Retrieved May 7, 2019, from https://www.nomadproject.io/docs/internals/scheduling/preemption.html

HashiCorp. (2019b, January 25). Scheduling. Retrieved from https://www.nomadproject.io/docs/internals/scheduling/scheduling.html

HashiCorp. (2019c, January 30). Schedulers. Retrieved April 1, 2019, from https://www.nomadproject.io/docs/schedulers.html

Hunt, R. (2017, November 29). Introducing AWS Fargate – Run Containers without Managing Infrastructure. Retrieved from https://aws.amazon.com/blogs/aws/aws-fargate/

Johnson, G., Scholes, K., & Whittington, R. (2008, January 21). Exploring Corporate Strategy. Retrieved May 20, 2019, from https://www.researchgate.net/profile/Constantin_Bratianu/post/The_difference_between_the_Concept_of_Strategic_Partnership_and_the_concept_of_Strategic_Relationship/attachment/5a10aa79b53d2f46c7eb03d3/AS%3A562163606409216%401511041656413/download/Johnson-ExploringCorporateStrategy_8Ed_Textbook.pdf

Kitchener, J. (2018, November 6). Kubernetes' scheduling magic revealed. Retrieved from https://www.oreilly.com/ideas/kubernetes-scheduling-magic-revealed

Kubernetes. (2019a, February 1). Pod Priority and Preemption. Retrieved from https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/

Kubernetes. (2019b, March 7). Authenticating. Retrieved May 10, 2019, from https://kubernetes.io/docs/reference/access-authn-authz/authentication/

Kubernetes. (2019c, March 7). Perform Rolling Update Using a Replication Controller. Retrieved May 3, 2019, from https://kubernetes.io/docs/tasks/run-application/rolling-update-replication-controller/

Kubernetes. (2019d, March 8). Kubernetes Components. Retrieved April 29, 2019, from https://kubernetes.io/docs/concepts/overview/components/

Kubernetes. (2019e, March 25). DNS for Services and Pods. Retrieved April 11, 2019, from https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/

Kubernetes. (2019f, March 25). Resource metrics pipeline. Retrieved April 24, 2019, from https://kubernetes.io/docs/tasks/debug-application-cluster/resource-metrics-pipeline/

Kubernetes. (2019g, March 27). Pod Priority and Preemption. Retrieved May 7, 2019, from https://kubernetes.io/docs/concepts/configuration/pod-priority-preemption/

Kubernetes. (2019h, March 31). Pod Overview. Retrieved April 1, 2019, from https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/

Kubernetes. (2019i, April 4). Services. Retrieved April 11, 2019, from https://kubernetes.io/docs/concepts/services-networking/service/

Kubernetes. (2019j, April 15). Concepts Underlying the Cloud Controller Manager. Retrieved April 29, 2019, from https://kubernetes.io/docs/concepts/architecture/cloud-controller/

Kubernetes. (2019k, April 15). Deployments. Retrieved May 3, 2019, from https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

Kubernetes. (2019l, April 15). Logging Architecture. Retrieved April 24, 2019, from https://kubernetes.io/docs/concepts/cluster-administration/logging/

Kubernetes. (2019m, April 19). Tools for Monitoring Resources. Retrieved April 24, 2019, from https://kubernetes.io/docs/tasks/debug-application-cluster/resource-usage-monitoring/

Kubernetes Community. (2018, March 21). Kubernetes Monitoring Architecture. Retrieved April 24, 2019, from https://github.com/kubernetes/community/blob/master/contributors/design-proposals/instrumentation/monitoring_architecture.md

Kubernetes Community. (2019, February 12). EFS Provisioner. Retrieved April 23, 2019, from https://github.com/kubernetes-incubator/external-storage/tree/master/aws/efs

Kubernetes Community - AWS Special Interest Group. (2019, March 1). AWS ALB Ingress Controller. Retrieved April 23, 2019, from https://github.com/kubernetes-sigs/aws-alb-ingress-controller

Lafeldt, M. (2019, April 27). Principles of Chaos Engineering. Retrieved May 17, 2019, from http://principlesofchaos.org

McDonald, C. (2018, May 1). Kubernetes, Kafka Event Sourcing Architecture Patterns and Use Case Examples. Retrieved from https://mapr.com/blog/kubernetes-kafka-event-sourcing-architecture-patterns-and-use-case-examples/

Mendelow, A. (Ed.). (1991). Proceedings of the 2nd International Conference on Information Systems. Cambridge, MA, USA.

Peck, N. (2018, January 26). Task Networking in AWS Fargate. Retrieved from https://aws.amazon.com/blogs/compute/task-networking-in-aws-fargate/

Prometheus. (2018, December 13). Overview. Retrieved April 25, 2019, from https://prometheus.io/docs/introduction/overview/

Schwarzkopf, M. (2013). Omega: flexible, scalable schedulers for large compute clusters – Google AI. Retrieved from https://ai.google/research/pubs/pub41684

Tremel, E. (2017, November 21). Six Strategies for Application Deployment. Retrieved May 8, 2019, from https://thenewstack.io/deployment-strategies/

Vargo, S. (2017, November 27). Load Balancing Strategies for HashiCorp Consul. Retrieved from https://www.hashicorp.com/blog/load-balancing-strategies-for-consul

Verma, A. (2015). Large-scale cluster management at Google with Borg – Google AI. Retrieved from https://ai.google/research/pubs/pub43438

List of Tables

List of Diagrams

Versioning

Version	Date	Changes
1	01/03/2019	First version
2	28/03/2019	Updated report structure, added more research
3	24/04/2019	Added design chapter
4	30/04/2019	Updated chapters based on feedback from supervisors, expanded design chapter
5	24/05/2019	Pre-final draft version
6	03/06/2019	Final draft version
7	16/06/2019	Updated report based on feedback from supervisors
8	17/06/2019	Final version

Appendix

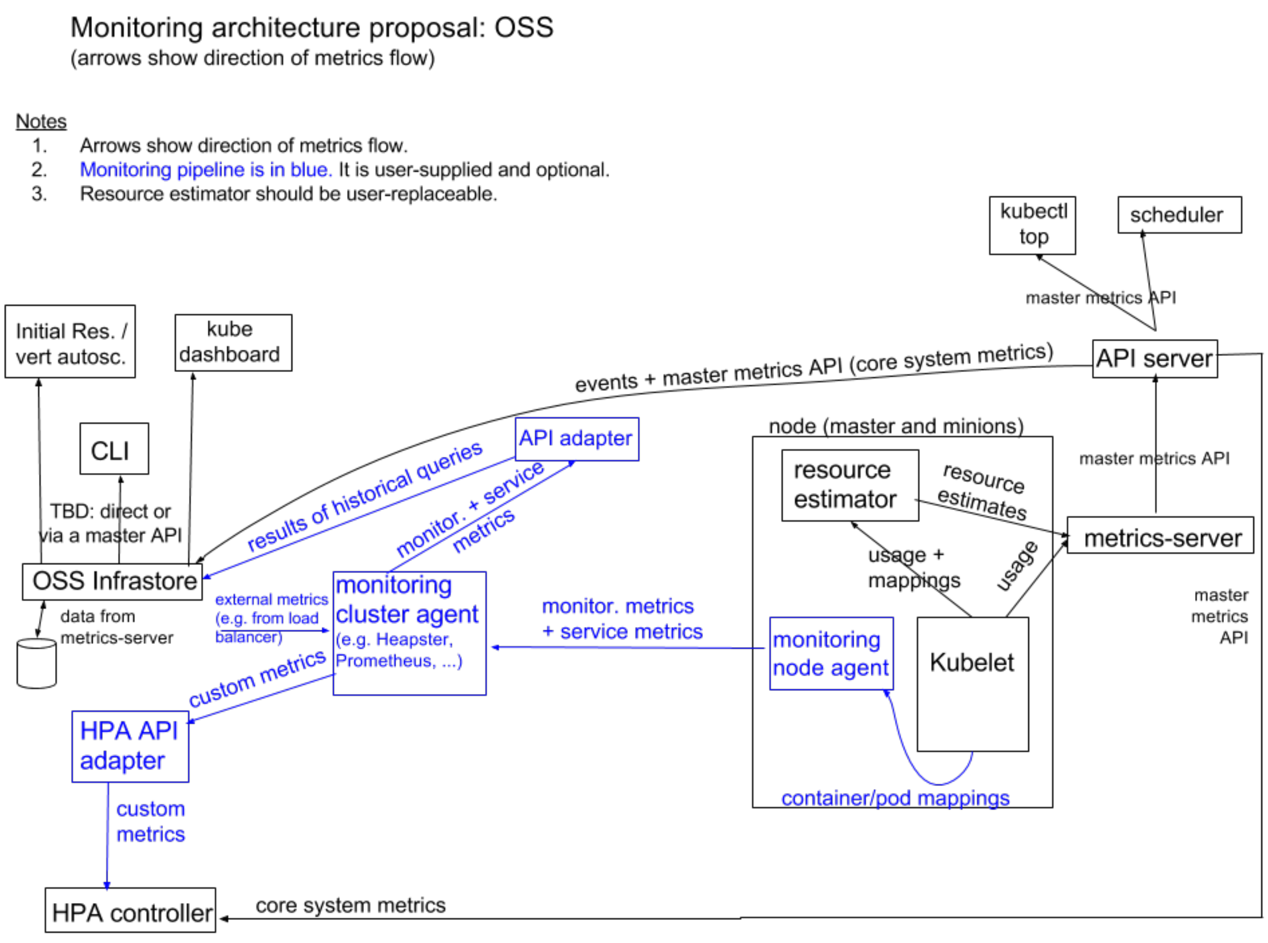
A1. Kubernetes Monitoring Architecture

The diagram below, curtesy of the Kubernetes community, illustrates the Kubernetes monitoring architecture. (Kubernetes Community, 2018)

The two monitoring processes are colour-coded. The black lines being the core monitoring processes and the blue lines the user-specified monitoring pipeline.

As visible on the diagram the API server serves the metrics API which can be consumed by, among other possibilities, kubectl, the scheduler and the HPA controller. Monitoring data by the Kubelet and resource estimator are being sent to the metrics server which makes the metrics API available to the API server.

The user-specified pipeline ties into the core metrics processes by consuming data from core components, like the kubelet. Furthermore a monitoring agent can be installed on the master and slave nodes which can collect even more metrics. Data from those agents is usually sent to a cluster-central monitoring agent (eg. Prometheus) and is exposed via an API adapter to be consumed by other components.


Monitoring architecture proposal: OSS
(arrows show direction of metrics flow)
Notes
1. Arrows show direction of metrics flow.
2. Monitoring pipeline is in blue. It is user-supplied and optional.
3. Resource estimator should be user-replaceable.
kubectl top
scheduler
master metrics API
API server
events + master metrics API (core system metrics)
Initial Res. / vert autosc.
kube dashboard
CLI
TBD: direct or via a master API
API adapter
results of historical queries
monitor. + service metrics
OSS Infrastore
data from metrics-server
external metrics (e.g. from load balancer)
monitoring cluster agent (e.g. Heapster, Prometheus, ...)
custom metrics
HPA API adapter
custom metrics
HPA controller
core system metrics
monitor. metrics + service metrics
node (master and minions)
resource estimator
resource estimates
usage + mappings
usage
metrics-server
master metrics API
master metrics API
monitoring node agent
Kubelet
container/pod mappings