Łódź University of Technology

Faculty of Electrical, Electronic, Computer and Control Engineering

Institute of Applied Computer Science

Institute of Electronics

Implementation of a Highly Scalable and Highly Available Environment for Containerized Applications Based on Kubernetes

Wdrożenie wysokoskalowalnego i wysokodostępnego środowiska

dla aplikacji skonteneryzowanych w oparciu o Kubernetes

Jakub Papuga

Computer Science

PhD Eng. Michał Bujacz

Łódź, 2023

Implementation of a Highly Scalable and Highly Available Environment for Containerized Applications Based on Kubernetes.

Abstract

This thesis presents a comprehensive study on the best practices for containerization and how to implement a Kubernetes cluster for high-availability applications. The main goal of the thesis is to provide practical guidance for engineers and architects who want to design and implement a robust and scalable environment for applications that require high availability.

The paper starts with an overview of containerization, its benefits, and its need in today's technology landscape. It then covers the basics of Kubernetes and its features, including container orchestration, scaling, and self-healing capabilities. The study also covers the various components of a Kubernetes cluster, such as nodes, pods, services, and more.

One of the key aspects of the study is the example of a highly scalable and highly available data ingestion pipeline. This example serves as a practical demonstration of how Kubernetes can be used to create a resilient and scalable environment for data ingestion. The pipeline includes various components such as data collection, data processing, and data storage, and the implementation of each component is discussed in detail.

Keywords:

High Availability, Kubernetes, Docker, High Scalability, Orchestration

Wdrożenie wysokoskalowalnego i wysokodostępnego środowiska dla aplikacji skonteneryzowanych w oparciu o Kubernetes.

Streszczenie

W niniejszej pracy przedstawiono kompleksowy zestaw najlepszych praktyk w zakresie konteneryzacji oraz sposobu wdrożenia klastra Kubernetes dla aplikacji o wysokiej dostępności. Głównym celem pracy jest dostarczenie praktycznych wskazówek dla inżynierów i architektów, którzy chcą zaprojektować i wdrożyć solidne i skalowalne środowisko dla aplikacji wymagających wysokiej dostępności.

Praca rozpoczyna się od przeglądu konteneryzacji, jej korzyści i potrzeby w dzisiejszym krajobrazie technologicznym. Następnie omawia podstawy Kubernetes i jego funkcje, w tym orkiestrację kontenerów, skalowanie i możliwości samoleczenia. Praca obejmuje również różne element klastra Kubernetes takie jak serwery, zasoby typu „Pod" oraz „Service" i inne.

Jednym z kluczowych elementów opracowania jest przykład wysoko skalowalnego i wysoko dostępnego potoku pozyskiwania danych z urządzeń. Przykład ten służy jako praktyczna demonstracja tego, jak Kubernetes może być użyty do stworzenia odpornego i skalowalnego środowiska do przetwarzania danych. System obejmuje różne komponenty, takie jak zbieranie danych, przetwarzanie danych i przechowywanie danych, a implementacja każdego komponentu jest szczegółowo omówiona.

Słowa kluczowe:

Wysoka dostępność, Kubernetes, Docker, wysoka skalowalność, orkiestracja

1. Introduction

The world is increasingly utilizing IoT devices (Gillis, 2022). These devices are transitioning from being used primarily by programmers to water flowers and are now penetrating our lives at every step. While disruption in access to smart speakers, sockets, or even flower watering devices may only slightly decrease the owner's comfort, a disruption in the operation of medical communication devices can lead to potentially life-threatening situations. An example of such a device is "The Band of Life" (originally named in Polish "Opaska Życia") produced by Comarch[1]. In addition to monitoring the user's pulse and activity, this bracelet allows immediate contact with a loved one or a medical rescuer in an emergency, initiated by pressing the SOS button. Devices of this type often rely on centralized services maintained by the manufacturer. Due to the architecture of such systems, even thousands of devices may utilize a single system. Therefore, in addition to ensuring high availability, the ability to scale such a system must also be considered.

The purpose of this study is to implement a highly scalable environment for applications that require high availability. This paper will explain the concept of containerization (Dua et al., 2014), its benefits, and its need in today's technology landscape. The paper then covers the basics of Kubernetes and its features, including container orchestration, scaling, and self-healing capabilities. Various components of the Kubernetes cluster, such as nodes, pods, services, and deployments, will be presented.

This work is of utmost importance for software engineers and architects responsible for designing and implementing systems for IoT devices. This study will help them ensure that the devices they develop are reliable and scalable by providing a comprehensive guide to containerization and high-availability systems.

All software used and its versions, as well as the source code for the custom application, will be included in Appendix 1 for reference and practical implementation purposes.

[1] https://www.comarch.pl/healthcare/produkty/teleopieka/opaska-zycia/

2. Containerization

Containerization is a method of packaging an application and its components, such as libraries, frameworks, and other dependencies, into a single image that can then be run in isolation from the rest of the system. In this context, "the rest of the system" can refer to other similarly packaged applications and the host machine's operating system. However, unlike virtual machines, containers are not self-sufficient. The container image does not include the system's kernel, and each container utilizes the kernel of its host system. This means an image built on Linux cannot be run on a Windows computer, for example. The situation becomes even more complex when considering changes in architecture; for instance, an image built on the x86 platform cannot be run on increasingly popular ARM64 servers (Wang et al., 2020). However, this limitation grants containers a significant advantage over virtual machines in terms of their lightweight nature, size, and compute resource requirements. Creating a dedicated environment for building such images has become very popular due to the savings resulting from better resource utilization (M et al., 2021; Wang et al., 2020) of server resources being many times higher than the cost of such an environment. It is also worth mentioning that the necessity of sharing the kernel does not mean that we are forced to share the entire operating system and its characteristics. A server originally based on Red Hat Enterprise Linux can be the host for a container based on Ubuntu or the very popular, due to its small size, Alpine. That is, on distributions with entirely different philosophies.

To run such a container on a server, we must first build its image and then provide some distribution system for the image. In the world of containers, such distribution systems are called registries. Registries are like libraries that, in addition to the storage and distribution function, can include features such as versioning, access control, and even building pipelines. In the case of Docker, the registry is called Docker Hub (Docker, n.d.). It is an online registry that is public by default but offers the option of creating private repositories for a fee. Another popular registry is the private registry provided by the Kubernetes cluster management system. It is often used in conjunction with the Kubernetes orchestration system.

To execute a container on a server, it is necessary first to build its image and then provide a distribution system for the image. In the world of containers, these distribution systems are called registries. Registries are akin to physical libraries. In addition to their storage and distribution function, they can include features such as versioning, access control, static vulnerability scanning, and removing duplicate image layers.

2.1. Open Container Initiative

To benefit from containerization, three elements are required: an application to build an image, a registry to store the image, and a runtime environment that can create a container based on the image. Docker is by far the most well-known and widely used container platform in the world. The initial release of Docker Engine was in 2013 (Docker Engine Release Notes, 2022). In 2015, the Open Container Initiative (OCI) was launched by Docker, CoreOS, and other leaders in the container industry (About the Open Container Initiative, n.d.) with the goal of establishing open industry standards for containers. As stated on the OCI website:

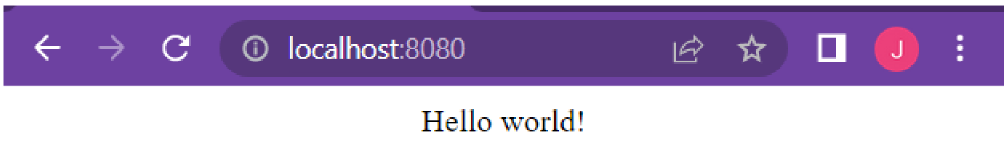
> *The OCI currently contains three specifications: the Runtime Specification (runtime-spec), the Image Specification (image-spec) and the Distribution Specification (distribution-spec). The Runtime Specification outlines how to run a "filesystem bundle" that is unpacked on disk. At a high-level an OCI implementation would download an OCI Image then unpack that image into an OCI Runtime filesystem bundle. At this point the OCI Runtime Bundle would be run by an OCI Runtime. (About the Open Container Initiative, 2022)*

The Open Container Initiative has provided a common ground for all container tools, significantly improving interoperability and accelerating the adoption of containers. Some tools worth mentioning include: Podman[2], a daemonless container engine for developing, managing, and running OCI containers; Buildah[3], a tool for building OCI container images; Buildkit[4], another tool for building OCI container images (used in newer versions of Docker); containerd[5], a high-level container runtime that pulls images from registries and hands them over to a lower-level runtime (initially a part of Docker, later separated out); and runc[6], a low-level container runtime that creates and runs the container process. An image built with Docker can be executed by cri-o just as easily as Podman can execute an image built with Buildah. In contrast, virtual machines still lack proper standardization, and migration from VMWare to Hyper-V or vice versa can be challenging for system administrators. Docker will be used in the following subchapters as it has all the features needed, wrapped in a user-friendly command line tool. This chapter aims to familiarize the reader with containerization rather than compare available platforms and toolchains.

[2] https://podman.io/
[3] https://buildah.io/
[4] https://github.com/moby/buildkit
[5] https://containerd.io/
[6] https://github.com/opencontainers/runc

2.2. Simple "Hello World!" application

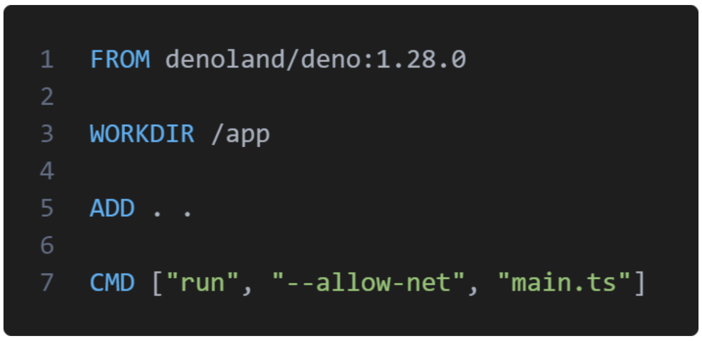
Throughout the following few chapters, we will be using a simple "Hello World!" application made in Deno (1.28.0) as a base. For now, the application only responds to requests to the "/" endpoint with a simple *"Hello World!"* message (see Figure 1).



Hello world!

Figure 1 Hello World! application (version 2.2) accessed from Web Browser

Typically, we would run the application using deno's command line tools (i.e., *deno run main.ts*). We might also be tempted to use our IDE built-in launch functionality. While this is a good approach for the development process, it does not provide a way to distribute the application. We can create a container image by creating a simple Dockerfile:



```
FROM denoland/deno:1.28.0

WORKDIR /app

ADD . .

CMD ["run", "--allow-net", "main.ts"]
```

Snippet 1 Dockerfile version 2.2

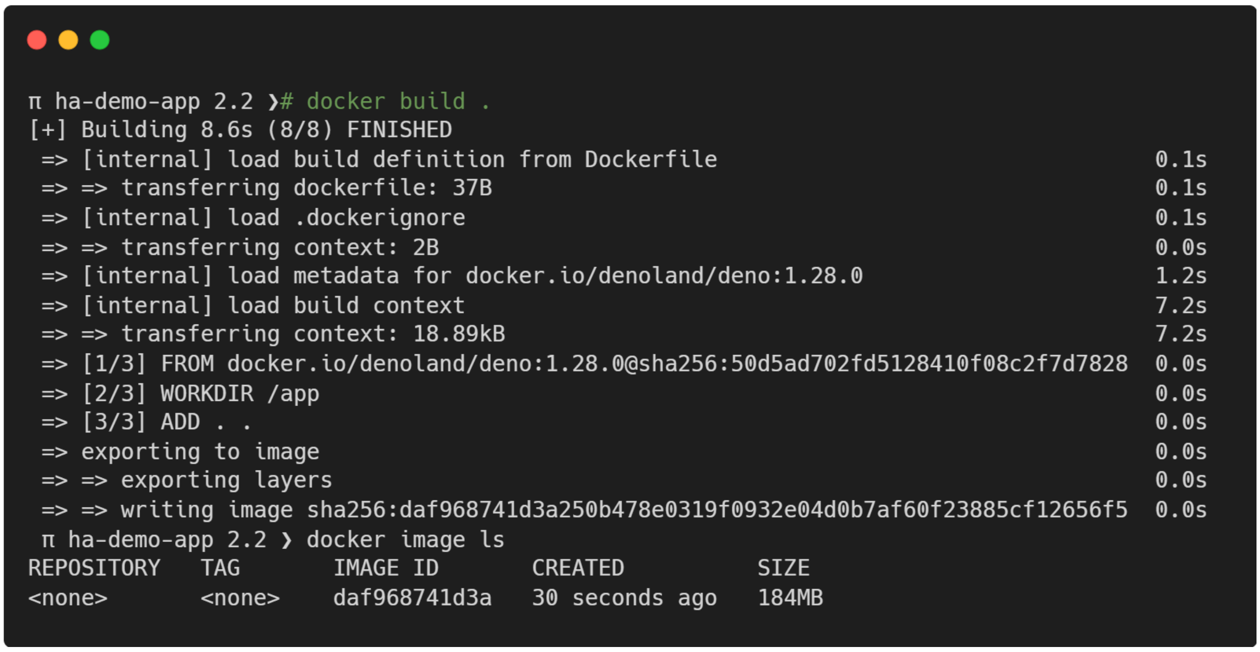
The *FROM* keyword allows us to specify an existing image as the base for our image. The *WORKDIR* keyword specifies a working directory for subsequent commands. To follow best practices and avoid working within the root directory ("/"), we will switch to the "/app" directory. The *ADD* keyword is a more powerful version of *COPY*; it can not only copy local files but also extract tar archives and download files from remote URLs on-the-fly. We will use ADD to copy our source code to the image in our case. The final *CMD* statement specifies the arguments to be used for the *ENTRYPOINT*, which has already been defined in *denoland/deno:1.28.0* by the deno maintainers as deno itself. The combination of ENTRYPOINT and CMD will result in the execution of the "*deno run –allow-net main.ts*" command.

With the above Dockerfile, we can build the image using the "docker build" command.

According to the documentation:

> *The docker build command builds Docker images from a Dockerfile and a "context". A build's context is the set of files located in the specified PATH or URL. The build process can refer to any of the files in the context. For example, your build can use a COPY instruction to reference a file in the context.* (*Docker Build*, n.d.)

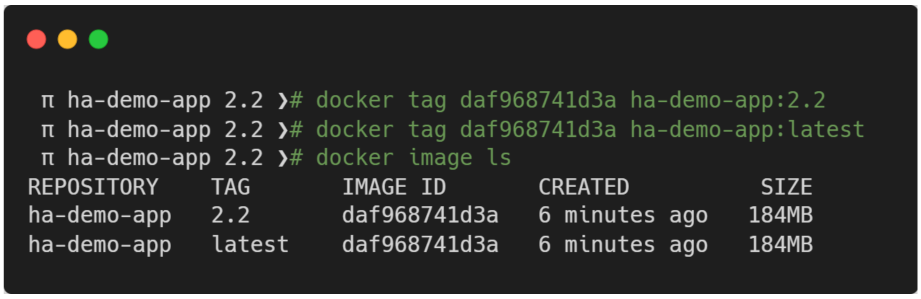
In the case of our "ha-demo-app" case, the context is the folder with the application's source code. Since our command line's working directory is usually the project directory, we can simplify the PATH to "." (see Figure 2).



```
π ha-demo-app 2.2 ❯# docker build .
[+] Building 8.6s (8/8) FINISHED
 => [internal] load build definition from Dockerfile                                     0.1s
 => => transferring dockerfile: 37B                                                      0.1s
 => [internal] load .dockerignore                                                        0.1s
 => => transferring context: 2B                                                          0.0s
 => [internal] load metadata for docker.io/denoland/deno:1.28.0                          1.2s
 => [internal] load build context                                                        7.2s
 => => transferring context: 18.89kB                                                     7.2s
 => [1/3] FROM docker.io/denoland/deno:1.28.0@sha256:50d5ad702fd5128410f08c2f7d7828  0.0s
 => [2/3] WORKDIR /app                                                                   0.0s
 => [3/3] ADD . .                                                                        0.0s
 => exporting to image                                                                   0.0s
 => => exporting layers                                                                  0.0s
 => => writing image sha256:daf968741d3a250b478e0319f0932e04d0b7af60f23885cf12656f5  0.0s
 π ha-demo-app 2.2 ❯ docker image ls
REPOSITORY   TAG       IMAGE ID       CREATED          SIZE
<none>       <none>    daf968741d3a   30 seconds ago   184MB
```

Figure 2 Building ha-demo-app (version 2.2) using the docker build command

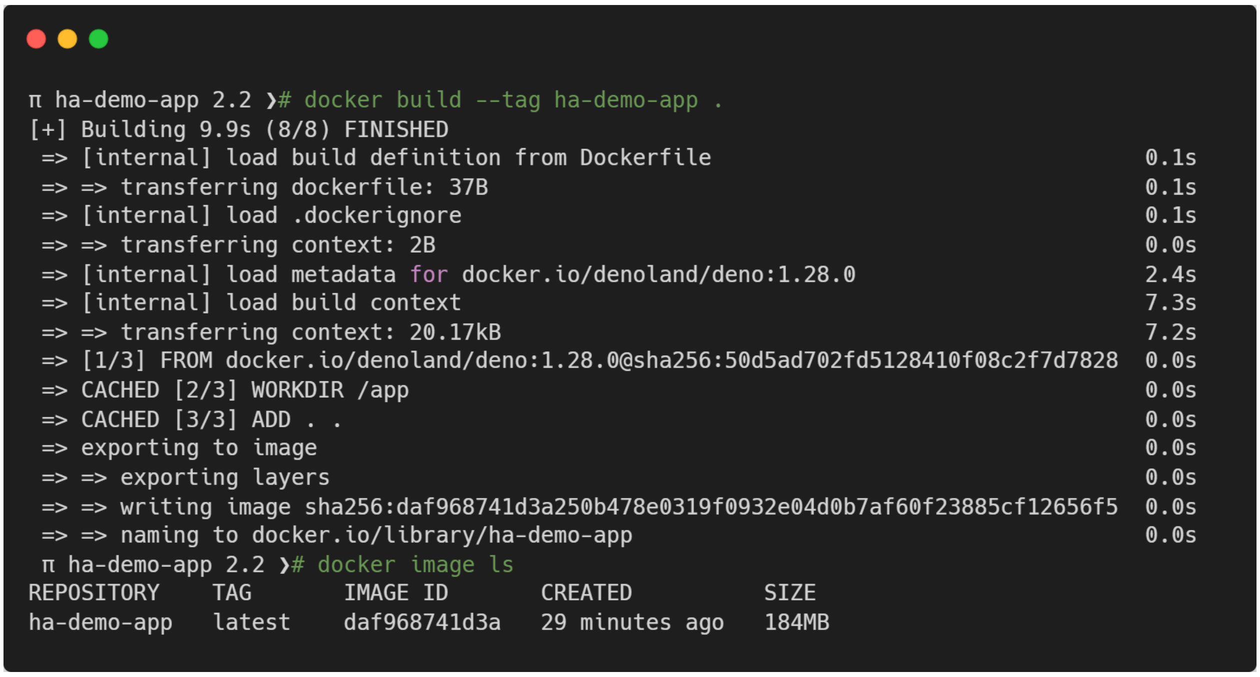
Docker images are identified by their hash, which can be seen in the *IMAGE ID* column in Figure x. While hashes help identify images, they are difficult for humans to understand. When we anticipate that humans will interact with our image, we should name it using the "docker tag" command. It is also possible to give an image multiple tags, as shown in Figure 3.



```
 π ha-demo-app 2.2 ❯# docker tag daf968741d3a ha-demo-app:2.2
 π ha-demo-app 2.2 ❯# docker tag daf968741d3a ha-demo-app:latest
 π ha-demo-app 2.2 ❯# docker image ls
REPOSITORY    TAG      IMAGE ID       CREATED         SIZE
ha-demo-app   2.2      daf968741d3a   6 minutes ago   184MB
ha-demo-app   latest   daf968741d3a   6 minutes ago   184MB
```

Figure 3 Manually tagging the ha-demo-app image with version 2.2

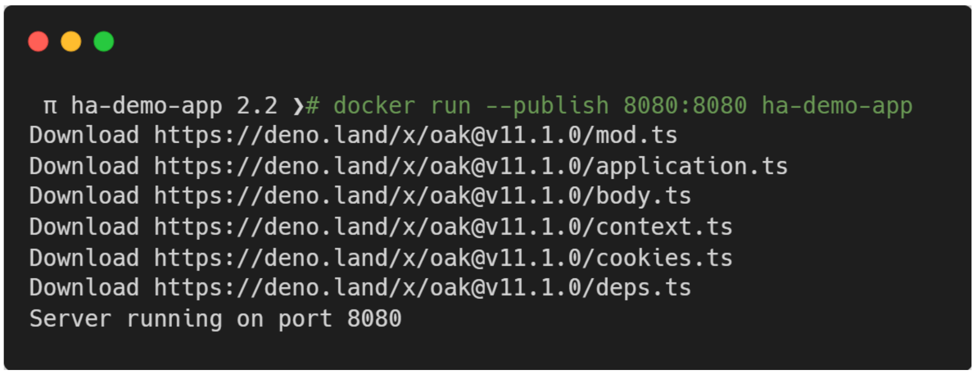
We can also tag our image during the build stage by using the "--tag" or " -t" argument with the "docker build" command. If we do not specify a tag, the "latest" tag will be used by default, as shown in Figure 4.



```
π ha-demo-app 2.2 ❯# docker build --tag ha-demo-app .
[+] Building 9.9s (8/8) FINISHED
 => [internal] load build definition from Dockerfile                          0.1s
 => => transferring dockerfile: 37B                                           0.1s
 => [internal] load .dockerignore                                             0.1s
 => => transferring context: 2B                                               0.0s
 => [internal] load metadata for docker.io/denoland/deno:1.28.0               2.4s
 => [internal] load build context                                             7.3s
 => => transferring context: 20.17kB                                          7.2s
 => [1/3] FROM docker.io/denoland/deno:1.28.0@sha256:50d5ad702fd5128410f08c2f7d7828  0.0s
 => CACHED [2/3] WORKDIR /app                                                 0.0s
 => CACHED [3/3] ADD . .                                                      0.0s
 => exporting to image                                                        0.0s
 => => exporting layers                                                       0.0s
 => => writing image sha256:daf968741d3a250b478e0319f0932e04d0b7af60f23885cf12656f5  0.0s
 => => naming to docker.io/library/ha-demo-app                                0.0s
 π ha-demo-app 2.2 ❯# docker image ls
REPOSITORY    TAG       IMAGE ID       CREATED          SIZE
ha-demo-app   latest    daf968741d3a   29 minutes ago   184MB
```

Figure 4 Tagging ha-demo-app image during build time

We can create a container from our image using the "docker run" command. Since our application is a web application, we may want to publish some container ports to the host using the "--publish" or " -p" argument. As with the "docker build" command, if we do not specify the image tag, the "latest" tag will be used by default, as shown in Figure 5.



```
 π ha-demo-app 2.2 ❯# docker run --publish 8080:8080 ha-demo-app
Download https://deno.land/x/oak@v11.1.0/mod.ts
Download https://deno.land/x/oak@v11.1.0/application.ts
Download https://deno.land/x/oak@v11.1.0/body.ts
Download https://deno.land/x/oak@v11.1.0/context.ts
Download https://deno.land/x/oak@v11.1.0/cookies.ts
Download https://deno.land/x/oak@v11.1.0/deps.ts
Server running on port 8080
```

Figure 5 Running ha-demo-app (version 2.2) with published ports

2.3. Image Layers

Thus far, we have been using a basic Dockerfile that copies the application's source code. In this chapter, we will enhance our image to make it suitable for production use.

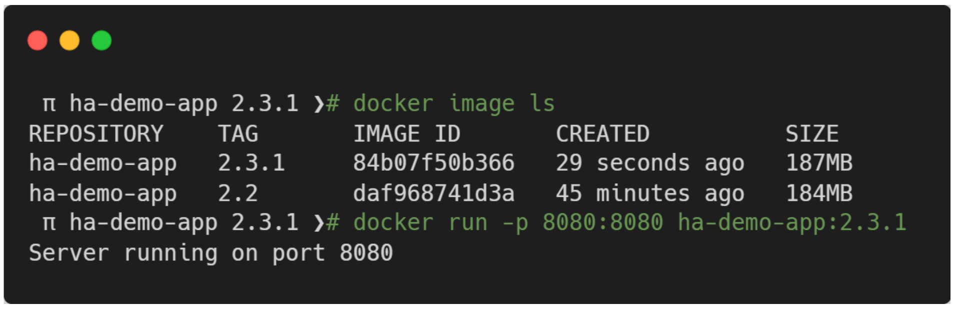
2.3.1. Include dependencies

In the previous chapter, we saw that Deno downloaded dependencies when we ran the container (as shown in Figure 5). This occurred because we only copied the application's source code to the image. This approach is not suitable for an air-gapped production environment. Additionally, the dependencies may change over time, and the application may fail to run. To address these issues, we can download the dependencies during the image creation process rather than during the application runtime. Deno provides the "deno cache" command to download dependencies at any time, and we can leverage this functionality. Using the RUN keyword, we can execute arbitrary commands during the image build process. By doing this, we can ensure that the dependencies are available when the application is run.

```
FROM denoland/deno:1.28.0

WORKDIR /app

ADD . .

RUN deno cache --lock locj.json deps.ts

CMD ["run", "--allow-net", "main.ts"]
```

Snippet 2 Dockerfile with dependency caching (version 2.3.1)

As the new image will contain the application dependencies, the image size is expected to increase slightly. However, this change will allow the application to run without downloading any libraries from the internet, as shown in Figure 6.

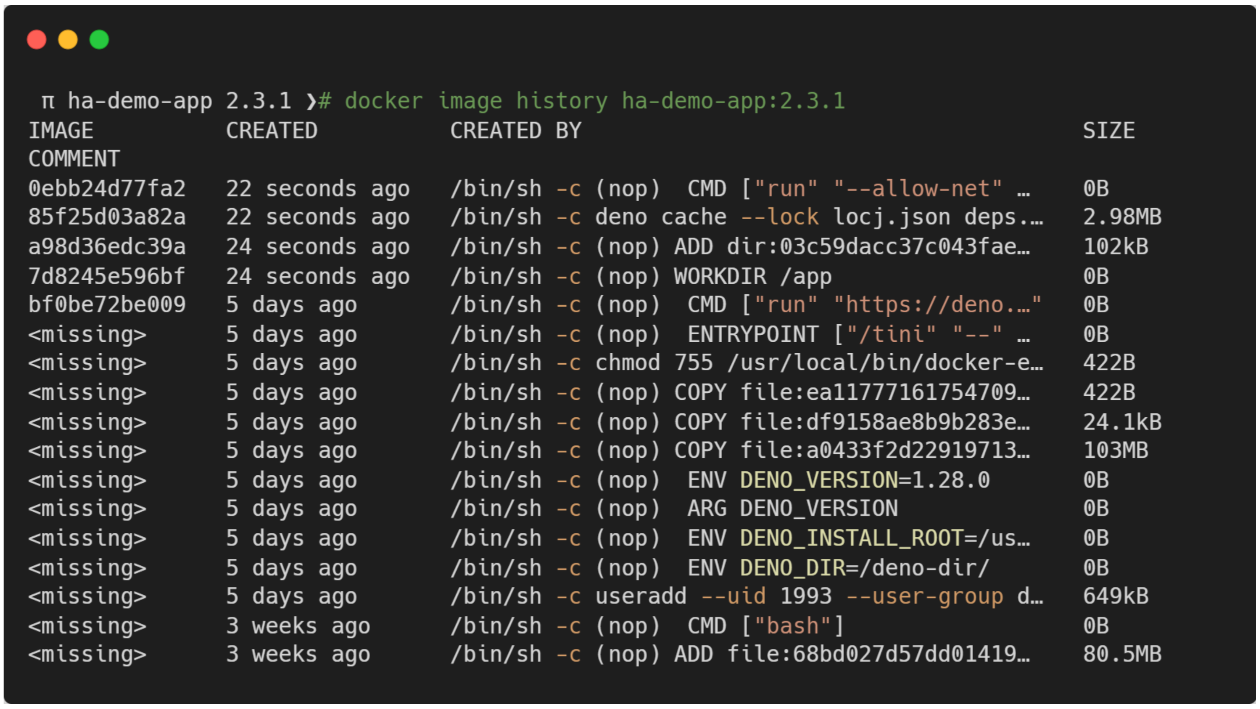


```
 π ha-demo-app 2.3.1 ❯# docker image ls
REPOSITORY    TAG       IMAGE ID       CREATED          SIZE
ha-demo-app   2.3.1     84b07f50b366   29 seconds ago   187MB
ha-demo-app   2.2       daf968741d3a   45 minutes ago   184MB
 π ha-demo-app 2.3.1 ❯# docker run -p 8080:8080 ha-demo-app:2.3.1
Server running on port 8080
```

Figure 6 Running ha-demo-app (version 2.3.1)

2.3.2. Caching

Now is an excellent time to introduce the concept of layers. We can view the layers of an image by using the "docker history" command, as shown in Figure 7. Every instruction within the Dockerfile that modifies the filesystem creates a new layer.



```
 π ha-demo-app 2.3.1 ❯# docker image history ha-demo-app:2.3.1
IMAGE          CREATED          CREATED BY                                      SIZE
COMMENT
0ebb24d77fa2   22 seconds ago   /bin/sh -c (nop)  CMD ["run" "--allow-net" …   0B
85f25d03a82a   22 seconds ago   /bin/sh -c deno cache --lock locj.json deps.…   2.98MB
a98d36edc39a   24 seconds ago   /bin/sh -c (nop) ADD dir:03c59dacc37c043fae…    102kB
7d8245e596bf   24 seconds ago   /bin/sh -c (nop) WORKDIR /app                   0B
bf0be72be009   5 days ago       /bin/sh -c (nop)  CMD ["run" "https://deno.…"   0B
<missing>      5 days ago       /bin/sh -c (nop)  ENTRYPOINT ["/tini" "--" …   0B
<missing>      5 days ago       /bin/sh -c chmod 755 /usr/local/bin/docker-e…   422B
<missing>      5 days ago       /bin/sh -c (nop) COPY file:ea11777161754709…    422B
<missing>      5 days ago       /bin/sh -c (nop) COPY file:df9158ae8b9b283e…    24.1kB
<missing>      5 days ago       /bin/sh -c (nop) COPY file:a0433f2d22919713…    103MB
<missing>      5 days ago       /bin/sh -c (nop)  ENV DENO_VERSION=1.28.0       0B
<missing>      5 days ago       /bin/sh -c (nop)  ARG DENO_VERSION              0B
<missing>      5 days ago       /bin/sh -c (nop)  ENV DENO_INSTALL_ROOT=/us…    0B
<missing>      5 days ago       /bin/sh -c (nop)  ENV DENO_DIR=/deno-dir/       0B
<missing>      5 days ago       /bin/sh -c useradd --uid 1993 --user-group d…   649kB
<missing>      3 weeks ago      /bin/sh -c (nop)  CMD ["bash"]                  0B
<missing>      3 weeks ago      /bin/sh -c (nop) ADD file:68bd027d57dd01419…    80.5MB
```

Figure 7 History of image ha-demo-app (version 2.3.1)

Starting from the bottom, we can see an image with ID *bf0be72be009*, which is the hash of *denoland/deno:1.28.0* that we specified in the FROM keyword in our Dockerfile. While we are missing the intermediate layers, we can still see how the Deno maintainers created the image. The layers created *three weeks ago* are likely the underlying Linux distribution files that the Deno maintainers used as a base. In the layers created *five days ago*, we can see that the Deno maintainers started by creating a new user with UID 1993, added some metadata with environmental variables, copied the Deno runtime, and set the default ENTRYPOINT and CMD for the container. The layers created seconds ago were created by us based on the Dockerfile. We changed the working directory to */app*, added our source code, downloaded the dependencies, and overwrote the default arguments for the ENTRYPOINT using the CMD instruction.

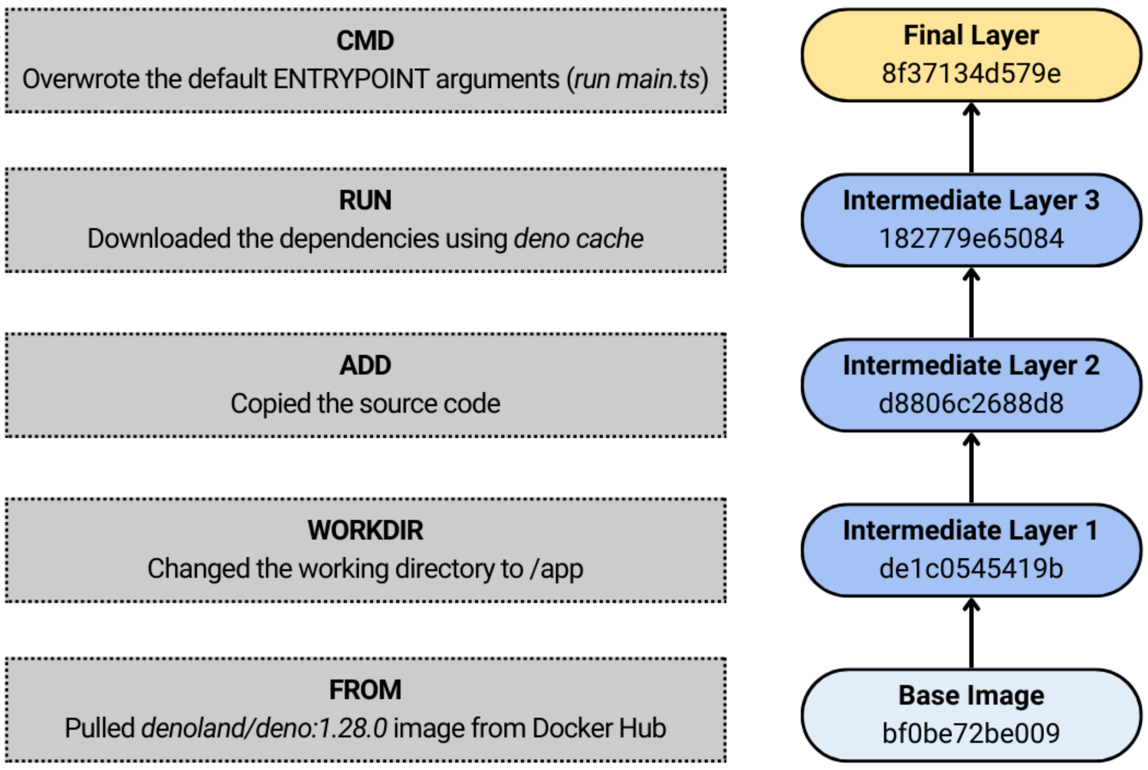
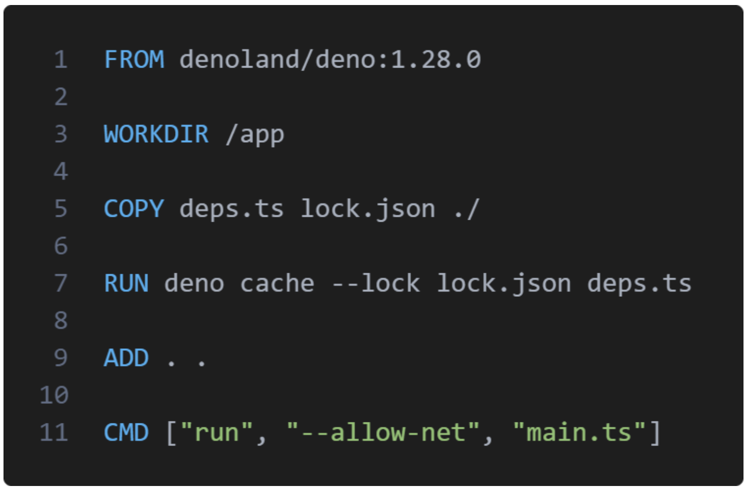

CMD
Overwrote the default ENTRYPOINT arguments (*run main.ts*)
RUN
Downloaded the dependencies using *deno cache*
ADD
Copied the source code
WORKDIR
Changed the working directory to /app
FROM
Pulled *denoland/deno:1.28.0* image from Docker Hub
Final Layer
8f37134d579e
Intermediate Layer 3
182779e65084
Intermediate Layer 2
d8806c2688d8
Intermediate Layer 1
de1c0545419b
Base Image
bf0be72be009

Figure 8 Graphical representation of image history

It is important to note that image layers are essentially diffs, with the newer layer containing information about the differences compared to the parent layer. By incorporating caching into our workflow, we can leverage this information and optimize the build times of our images. Our source code may change frequently, but our dependencies remain consistent. By separating our *ADD* layer into two layers - one for copying dependencies and one for copying the application source code - we can avoid downloading dependencies every time the source code is modified and instead utilize cached dependencies. This can be achieved through modifications to the Dockerfile, as demonstrated in Snippet 3.



```dockerfile
FROM denoland/deno:1.28.0

WORKDIR /app

COPY deps.ts lock.json ./

RUN deno cache --lock lock.json deps.ts

ADD . .

CMD ["run", "--allow-net", "main.ts"]
```

Snippet 3 Dockerfile with layer caching (version 2.3.2)

Henceforth, dependencies will be cached unless the contents of the deps.ts file are modified.

Note: Newer versions of Docker include a new tool called Buildkit for building images, which does not retain intermediate images. As a result, the appearance of Figure 7 may differ when using Buildkit. It is possible to revert to the original Docker build tool by exporting the DOCKER_BUILDKIT=0 environment variable or editing the daemon.json file. For up-to-date information, refer to Docker's BuildKit documentation[7].

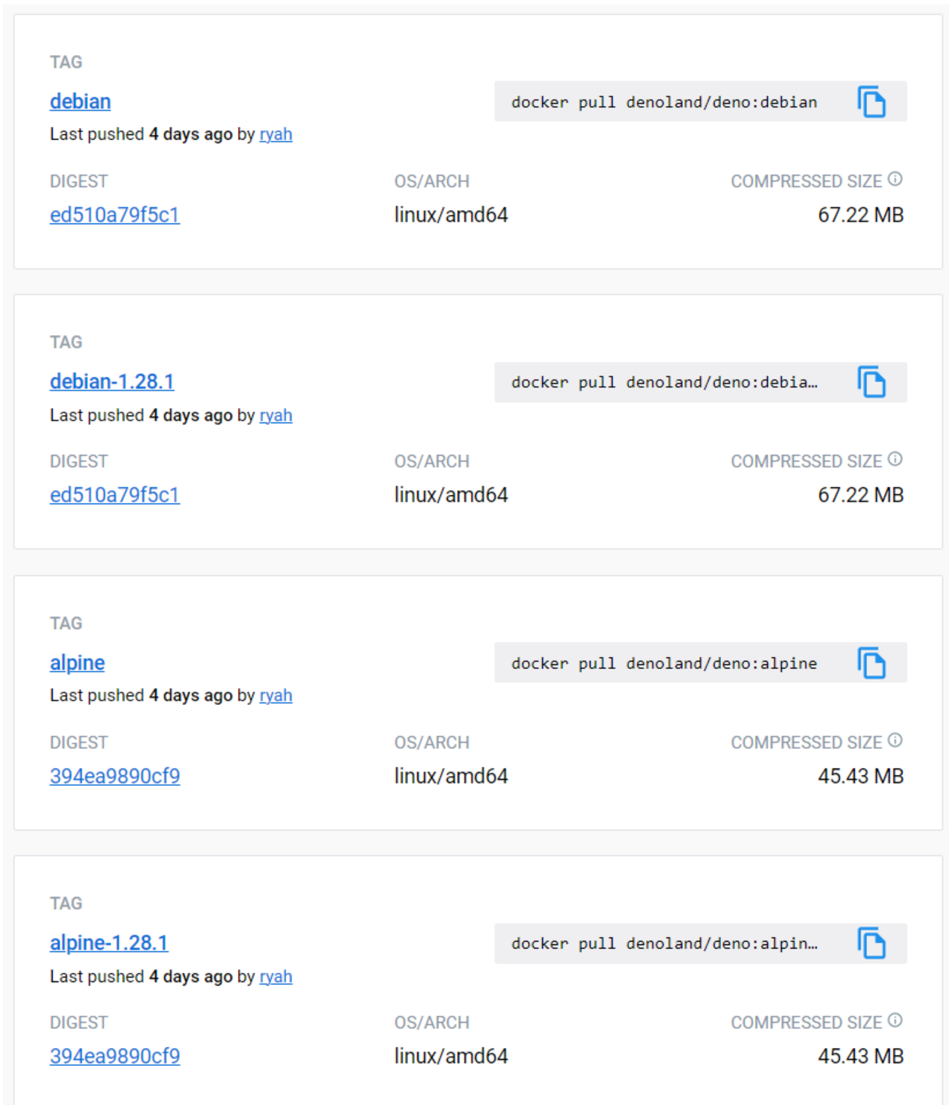
2.3.3. Specific image versions

We have previously adhered to the best practice of using as specific images as possible, but it is worth explaining the reasoning behind this rule.

The use of specific images ensures that the same version of the code and its dependencies are used every time, reducing the risk of inconsistencies in the application's behavior. This is particularly important in production environments, where a single change to the codebase could cause significant problems for the users. By using specific images, the application can be deployed with confidence, knowing that the same image will be used every time.

It is important to note that image tags are not necessarily tied to a specific image ID. If we examine Deno's Docker Hub (Docker's default image registry), we can observe multiple tags associated with the same image.

[7] https://docs.docker.com/build/buildkit/


TAG
debian
Last pushed **4 days ago** by ryah

```
docker pull denoland/deno:debian
```

DIGEST	OS/ARCH	COMPRESSED SIZE
ed510a79f5c1	linux/amd64	67.22 MB

TAG
debian-1.28.1
Last pushed **4 days ago** by ryah

```
docker pull denoland/deno:debia…
```

DIGEST	OS/ARCH	COMPRESSED SIZE
ed510a79f5c1	linux/amd64	67.22 MB

TAG
alpine
Last pushed **4 days ago** by ryah

```
docker pull denoland/deno:alpine
```

DIGEST	OS/ARCH	COMPRESSED SIZE
394ea9890cf9	linux/amd64	45.43 MB

TAG
alpine-1.28.1
Last pushed **4 days ago** by ryah

```
docker pull denoland/deno:alpin…
```

DIGEST	OS/ARCH	COMPRESSED SIZE
394ea9890cf9	linux/amd64	45.43 MB

Figure 9 Different tags available on Docker Hub

In Figure 10, we can see that tags *debian* and *debian-1.28.1* are bound to the same image with ID *ed510a79f5c1*. We can deduce from this information that the *debian* tag will always point to the newest version of Deno. As time passes, we might want to push a patch or introduce new functionality to our application. We may not want our build to fail or, even worse, have our application misbehave only because the version of our underlying images has silently changed with breaking changes in the newer version. We should always be pointing to as specific an image as possible. If the maintainers of a particular project follow Semantic Versioning[8], it is a good practice to point to an image with specific MAJOR and MINOR versions. It is generally accepted to use a tag without a PATCH version (provided that maintainers provide such a tag), as patch releases should not introduce a breaking change, only bug fixes.

[8] https://semver.org/

2.3.4. Image security

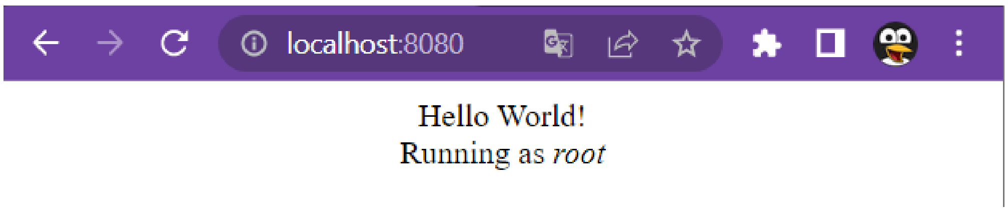
By default, all processes within a container are run by the *root* user. This violates the principle of least privilege (PoLP) (What Is the Principle of Least Privilege (POLP), 2022). In case our application gets compromised, we do not want the attacker to be able to start undesirable processes in our container. Beyond modifying our configuration, the attacker might be able to install additional software to exploit our infrastructure further. Moreover, if a container runtime vulnerability is discovered, running the container as an unprivileged user will prevent the attacker from gaining elevated permissions on the host.

Running as non-root requires a couple of additional steps inside our Dockerfile. We must ensure that the specified user exists (and if not, create one). Furthermore, we need to provide the user with appropriate file system permissions. In our demo application, this can be achieved in the following way:

```dockerfile
FROM denoland/deno:1.28.0

WORKDIR /app

RUN useradd --no-create-home demo-app \
    && chown -R demo-app /app \
    && chown -R demo-app /deno-dir

USER demo-app

COPY deps.ts lock.json ./

RUN deno cache --lock lock.json deps.ts

ADD . .

CMD ["run", "--allow-net", "--allow-run", "main.ts"]
```

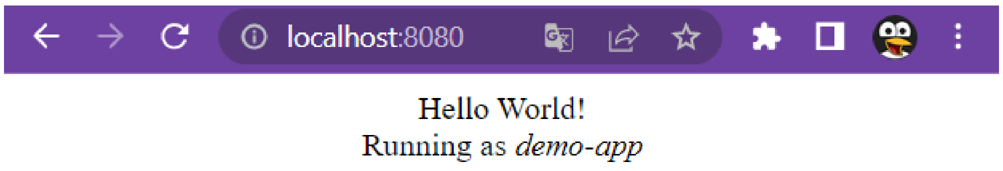
Snippet 4 Dockerfile with defined USER (version 2.3.4)

To visualize the changes, the demo application has been extended with *whoami* functionality displayed below "*Hello World!*". Please see the figures below (Figure 11, Figure 12).



Hello World!
Running as *root*

Figure 10 Application running as root (before changes)



Hello World!
Running as *demo-app*

Figure 11 Application running as deno-app (after changes)

2.3.5. Shrinking image footprint

We can optimize the image size to make our future deployments more snappy. Lowe image size should decrease the time needed to download the image from the registry by the destination server. One of the simplest ways to achieve that is to replace the base image with a more lightweight one. In the case of Deno, the maintainers have prepared multiple base images: Debian (approx. 69MB), Ubuntu (approx. 65MB), CentOS (approx. 118MB), and Alpine (approx. 46MB). So far, we have been using the default "1.28.0" image based on Debian.

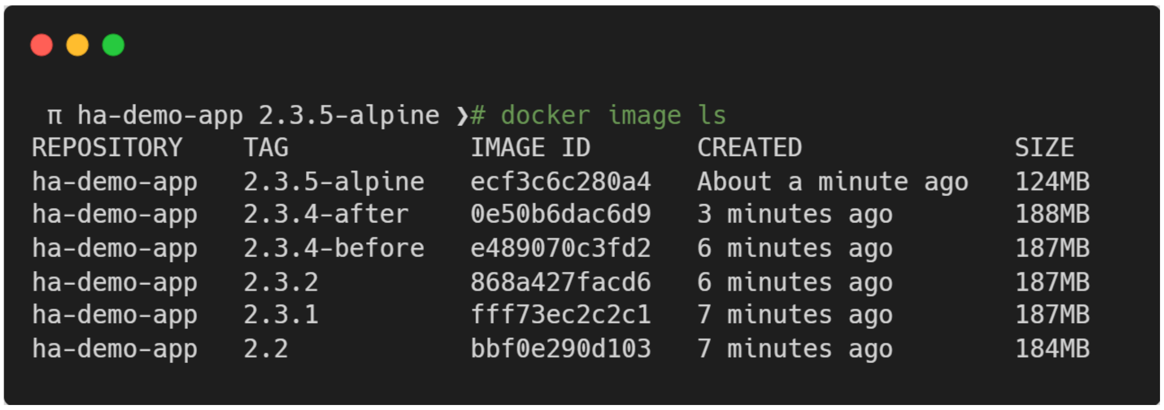
Replacing the base distribution with a more lightweight one has another significant benefit - decreased vulnerability plane. Smaller distributions come with lower amounts of packages that an attacker could potentially exploit.

Since we are not using any functionality specific to Debian, we can swap the image for the smallest distribution supported by Deno - Alpine. This can be done by modifying the *FROM* keyword in our Dockerfile:

```dockerfile
FROM denoland/deno:alpine-1.28.0

WORKDIR /app

RUN adduser -D -H demo-app \
    && chown -R demo-app /app \
    && chown -R demo-app /deno-dir

USER demo-app

COPY deps.ts lock.json ./

RUN deno cache --lock lock.json deps.ts

ADD . .

CMD ["run", "--allow-net", "--allow-run", "main.ts"]
```

Snippet 5 Dockerfile with Alpine as the base image (version 2.3.5-alpine)

As a result, our image size has decreased from 188MB to 124MB (see Figure 12).



```
 π ha-demo-app 2.3.5-alpine ❯# docker image ls
REPOSITORY    TAG            IMAGE ID       CREATED              SIZE
ha-demo-app   2.3.5-alpine   ecf3c6c280a4   About a minute ago   124MB
ha-demo-app   2.3.4-after    0e50b6dac6d9   3 minutes ago        188MB
ha-demo-app   2.3.4-before   e489070c3fd2   6 minutes ago        187MB
ha-demo-app   2.3.2          868a427facd6   6 minutes ago        187MB
ha-demo-app   2.3.1          fff73ec2c2c1   7 minutes ago        187MB
ha-demo-app   2.2            bbf0e290d103   7 minutes ago        184MB
```

Figure 12 Image sizes after changing the base image to Alpine (version 2.3.5-alpine)

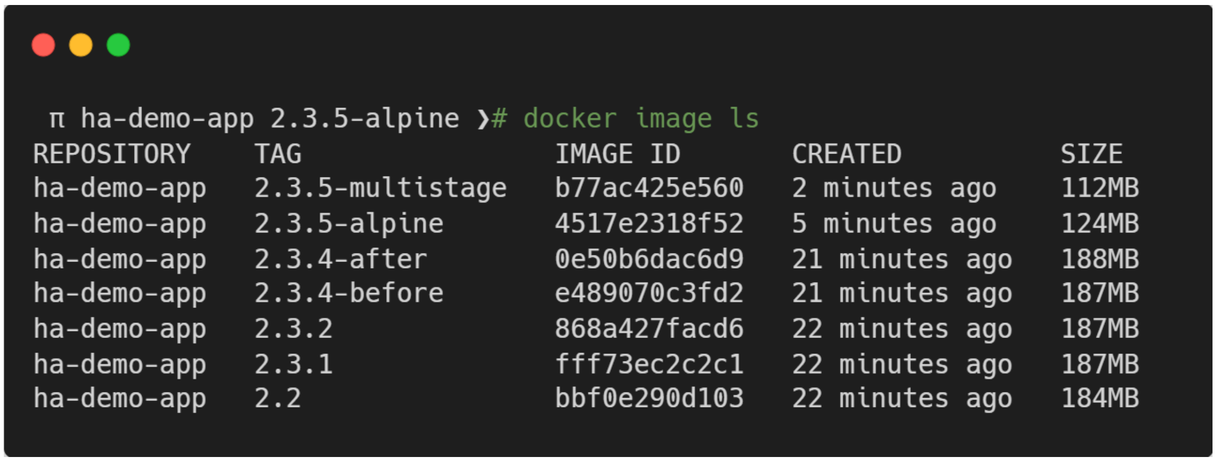
Many languages can be compiled and run without the need for compiler presence. We can divide our Dockerfile into two stages - one for compiling the application and the second for running it. Only the second stage will be kept for distribution. In the container world, we call such an approach a *multistage build*.

Deno is an interpreted language, and the interpreter has to be present during the execution of the application; however, Deno can be "compiled" into a self-contained executable. While we will not see a drastic change in the image size compared to traditionally compiled languages such as C, C++, C# (gigabytes down to megabytes), or Golang (hundreds of megabytes to a dozen megabytes), we should still be able to cut down on some unnecessary parts of the base image. This can be achieved by introducing another *FROM* keyword to the Dockerfile. To be able to copy the files from the previous stage, we need to name the stages with the *AS* keyword:

```dockerfile
FROM denoland/deno:alpine-1.28.0 as builder
WORKDIR /app
COPY deps.ts lock.json ./
RUN deno cache --lock lock.json deps.ts
ADD . .
RUN deno compile --output ha-demo-app --allow-net --allow-run main.ts

FROM alpine:3.17 as runner
WORKDIR /app
RUN adduser -D -H demo-app \
    && chown -R demo-app /app
USER demo-app
COPY --from=builder /app/ha-demo-app /app/ha-demo-app
CMD ["/app/ha-demo-app"]
```

Snippet 6 Multistage Dockerfile example (version 2.3.5-multistage)

Notice that we have moved the user creation to the "runner" stage, as we only care about the user during runtime, not compilation time. The resulting image is 12MB smaller than the previous one (see Figure 13).



```
 π ha-demo-app 2.3.5-alpine ❯# docker image ls
REPOSITORY     TAG                  IMAGE ID       CREATED          SIZE
ha-demo-app    2.3.5-multistage     b77ac425e560   2 minutes ago    112MB
ha-demo-app    2.3.5-alpine         4517e2318f52   5 minutes ago    124MB
ha-demo-app    2.3.4-after          0e50b6dac6d9   21 minutes ago   188MB
ha-demo-app    2.3.4-before         e489070c3fd2   21 minutes ago   187MB
ha-demo-app    2.3.2                868a427facd6   22 minutes ago   187MB
ha-demo-app    2.3.1                fff73ec2c2c1   22 minutes ago   187MB
ha-demo-app    2.2                  bbf0e290d103   22 minutes ago   184MB
```

Figure 13 Image sizes after implementing multistage build (version 2.3.5-multistage)

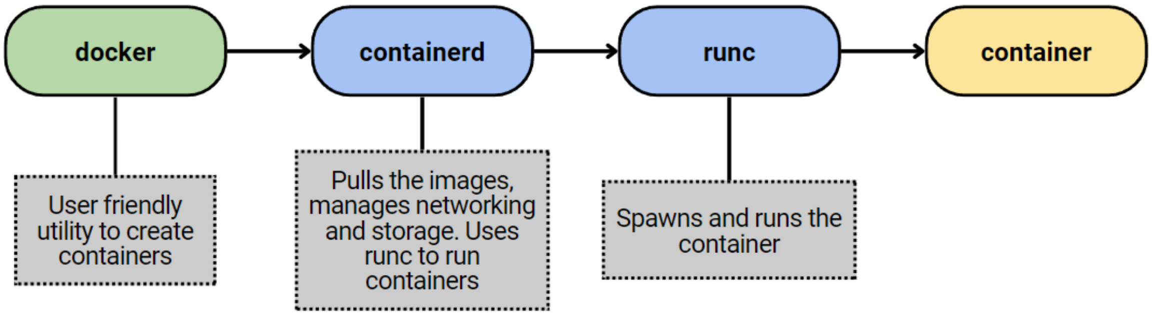
Opinion: Avoid using SCRATCH or BIN images. Scratch containers are not only hard to debug by DevOps teams in production environments (missing shell, package manager), but they also have multiple pitfalls which may prevent the application from running. Such pitfalls include missing proper user management, important folders (e.g., /tmp), timezone info, and even CA certificates. If possible, use distro-based or so-called distroless[9] images.

2.3.6. Useful metadata

Dockerfile specification has multiple instructions that can be considered metadata. The ones worth mentioning are *LABEL* and *EXPOSE*. The LABEL instruction can set arbitrary labels, though it is often used to indicate the author of the generated images. The EXPOSE instruction informs the runtime that the container listens on the specified port. The EXPOSE instruction does not actually publish the port. It functions as a documentation between the person who builds the image and the person who runs the container about which ports are intended to be published. If we pass the "--publish" flag to "docker run" without specifying the exact ports, docker will automatically map the port from EXPOSE instruction to a randomly available port on the host.

[9] https://github.com/GoogleContainerTools/distroless

Note: Some Dockerfiles may contain the MAINTAINER instruction. It is currently considered deprecated in favor of more generic LABEL instruction. However, the principle stays the same.


docker
containerd
runc
container
User friendly utility to create containers
Pulls the images, manages networking and storage. Uses runc to run containers
Spawns and runs the container

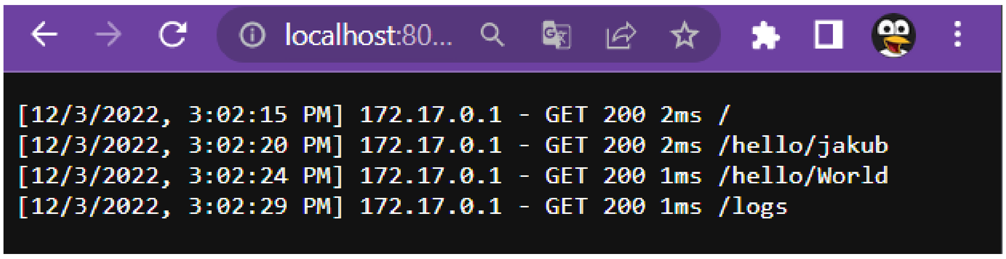
```
FROM denoland/deno:alpine-1.28.0 as builder
WORKDIR /app
COPY deps.ts lock.json ./
RUN deno cache --lock lock.json deps.ts
ADD . .
RUN deno compile --output ha-demo-app --allow-net --allow-run main.ts

FROM alpine:3.17 as runner
LABEL org.opencontainers.image.authors mrpsycho@mrpsycho.pl
EXPOSE 8080
WORKDIR /app
RUN adduser -D -H demo-app \
    && chown -R demo-app /app
USER demo-app
COPY --from=builder /app/ha-demo-app /app/ha-demo-app
CMD ["/app/ha-demo-app"]
```

Snippet 7 Example of Dockerfile with EXPOSE and LABEL instructions (version 2.3.6)

2.4. Volumes

We start with a fresh copy of our image every time we create a container. By default, containers have no data persistence. Since we may want to persist the data generated by our application (logs, database contents), we have to use volumes.

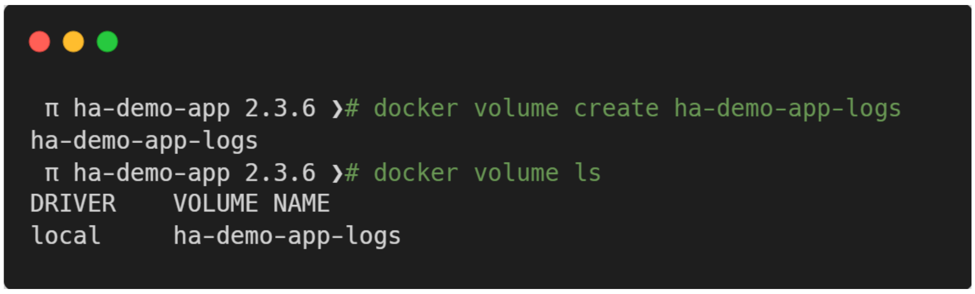
Our demo application has been extended for demonstration purposes with a logger and two new endpoints /hello/:name (where *:name* is a variable) and /logs. The logs are kept in a file. If we run the container and visit a few pages, we see our visit history in the logs (Figure 15).



```
[12/3/2022, 3:02:15 PM] 172.17.0.1 - GET 200 2ms /
[12/3/2022, 3:02:20 PM] 172.17.0.1 - GET 200 2ms /hello/jakub
[12/3/2022, 3:02:24 PM] 172.17.0.1 - GET 200 1ms /hello/World
[12/3/2022, 3:02:29 PM] 172.17.0.1 - GET 200 1ms /logs
```

Figure 14 /logs endpoint in a web browser

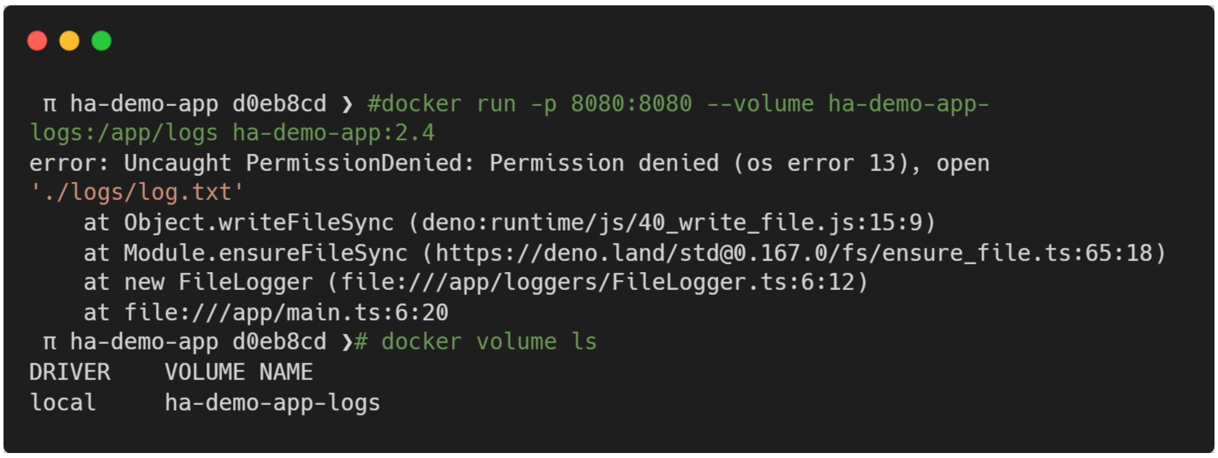
If we now recreate the container, the data will be lost. To persist the logs, we must create a volume and mount it inside the container. A volume can be either created explicitly or automatically. To create a volume explicitly, use "*docker volume create*" (see Figure 16).



```
 π ha-demo-app 2.3.6 ❯# docker volume create ha-demo-app-logs
ha-demo-app-logs
 π ha-demo-app 2.3.6 ❯# docker volume ls
DRIVER    VOLUME NAME
local     ha-demo-app-logs
```

Figure 15 Creation of a docker volume

To create a volume automatically, it is enough just to mount it to the container using "*docker run*" (see Figure 17).



```
 π ha-demo-app d0eb8cd ❯ #docker run -p 8080:8080 --volume ha-demo-app-
logs:/app/logs ha-demo-app:2.4
error: Uncaught PermissionDenied: Permission denied (os error 13), open
'./logs/log.txt'
    at Object.writeFileSync (deno:runtime/js/40_write_file.js:15:9)
    at Module.ensureFileSync (https://deno.land/std@0.167.0/fs/ensure_file.ts:65:18)
    at new FileLogger (file:///app/loggers/FileLogger.ts:6:12)
    at file:///app/main.ts:6:20
 π ha-demo-app d0eb8cd ❯# docker volume ls
DRIVER    VOLUME NAME
local     ha-demo-app-logs
```

Figure 16 Automatically creating a volume with docker run

In figure 16 above, we got a PermissionDenied error. The error indicates that the application has no permission to the logs folder. This is because we are running as a non-root user inside the container, and docker's volume has been mounted with root-only permissions. As of December 2022, it is impossible to mount the volume as a user other than root (please see issue #2259 on https://github.com/moby/moby). Our only option is to adjust the volume permissions during the runtime. The *demo-app* user does not have permission to change the volume ownership; however, Linux native binaries can be run with elevated privileges. The *setuid* instruction tells the Linux kernel to run an executable as the owner of the executable instead of the current user. The following Go[10] code will be used to change the */app/logs* folder ownership (see Snippet 8).

```go
func main() {
    userName := "demo-app"
    groupName := "demo-app"
    folderName := "/app/logs"

    fmt.Println("Changing permissions for folder: " + folderName)

    myUser, err := user.Lookup(userName)
    if err != nil {
        log.Fatalf("User lookup failed: %s", err)
    }
    myGroup, err := user.LookupGroup(groupName)
    if err != nil {
        log.Fatalf("Group lookup failed: %s", err)
    }

    myUserInt, _ := strconv.Atoi(myUser.Uid)
    myGroupInt, _ := strconv.Atoi(myGroup.Gid)

    err = os.Chown(folderName, myUserInt, myGroupInt)
    if err != nil {
        log.Fatalf("Chown failed: %s", err)
    }
    fmt.Println("Successfully changed permissions")
}
```

Snippet 8 Go function to adjust permissions for deno-app user

The code will be compiled into a binary in a separate stage inside the Dockerfile, copied into the final stage, and given the *setuid* flag using the "*chmod* command (see Snippet 9 and Snippet 10).

[10] https://go.dev/

```
FROM golang:1.19-alpine3.16 as gocompiler
WORKDIR /go/src/adjust_permissions
COPY adjust_permissions.go main.go
RUN go mod init adjust_permissions
RUN go build
```

Snippet 9 gocompiler stage in Dockerfile (version 2.4)

```
FROM frolvlad/alpine-glibc:alpine-3.16 as runner
LABEL org.opencontainers.image.authors mrpsycho@mrpsycho.pl
EXPOSE 8080
WORKDIR /app
RUN adduser -D demo-app \
    && chown -R demo-app /app
COPY --from=gocompiler /go/src/adjust_permissions/adjust_permissions .
RUN chmod u+s adjust_permissions
USER demo-app
COPY --from=builder /app/ha-demo-app /app/ha-demo-app
CMD ["/bin/sh", "-c", "/app/adjust_permissions && /app/ha-demo-app"]
```

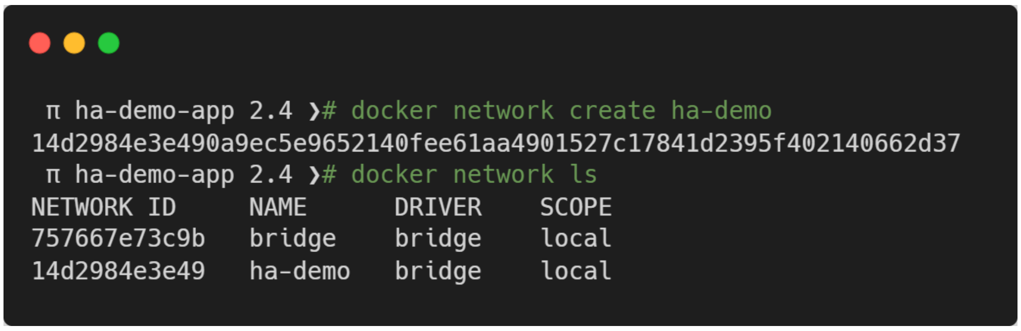
Snippet 10 runner stage in Dockerfile (version 2.4)

We should now be able to run our Deno project.

Warning: Do not create a reusable *setuid* binary that calls another script unless you can guarantee that an attacker cannot modify the script. Be aware of any race conditions that can be exploited.

2.5. Networking

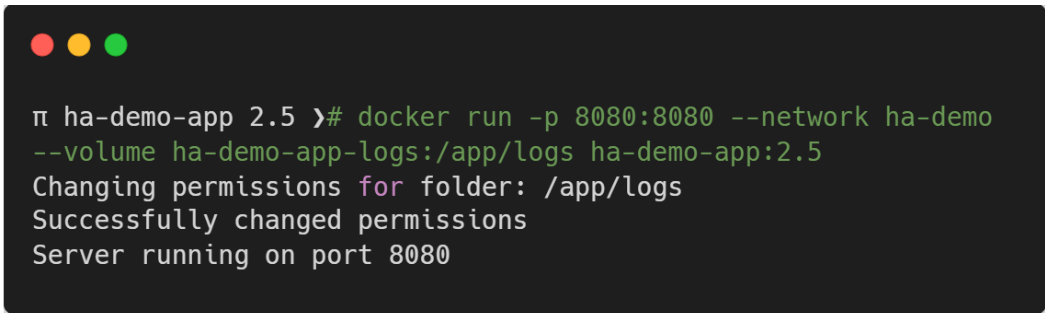
If we want our containers to be able to connect to each other, we have to put them within the same network. Containers within the same network can communicate freely using each other's names. By default, docker puts all containers within the same network. We can reduce our potential attack surface by separating our workloads into separate networks. A network can be created using the "docker network create" command (see Figure 18).



```
 π ha-demo-app 2.4 ❯# docker network create ha-demo
14d2984e3e490a9ec5e9652140fee61aa4901527c17841d2395f402140662d37
 π ha-demo-app 2.4 ❯# docker network ls
NETWORK ID     NAME      DRIVER    SCOPE
757667e73c9b   bridge    bridge    local
14d2984e3e49   ha-demo   bridge    local
```

Figure 17 Manual creation of docker network

To run a container inside a specific network, the "--network" flag has to be passed to the "docker run" command (see Figure 18).



```
π ha-demo-app 2.5 ❯# docker run -p 8080:8080 --network ha-demo
--volume ha-demo-app-logs:/app/logs ha-demo-app:2.5
Changing permissions for folder: /app/logs
Successfully changed permissions
Server running on port 8080
```

Figure 18 Running ha-demo-app with network and volume (version 2.5)

The ha-demo-app has been extended to include an Elasticsearch logger for demonstration purposes. Elasticsearch is a part of the Elastic Stack, a data analysis platform. In addition to the Elasticsearch database, we will also use Kibana, another tool from the Elastic family, to visualize the data. The final network should resemble the following (see Figure 19).

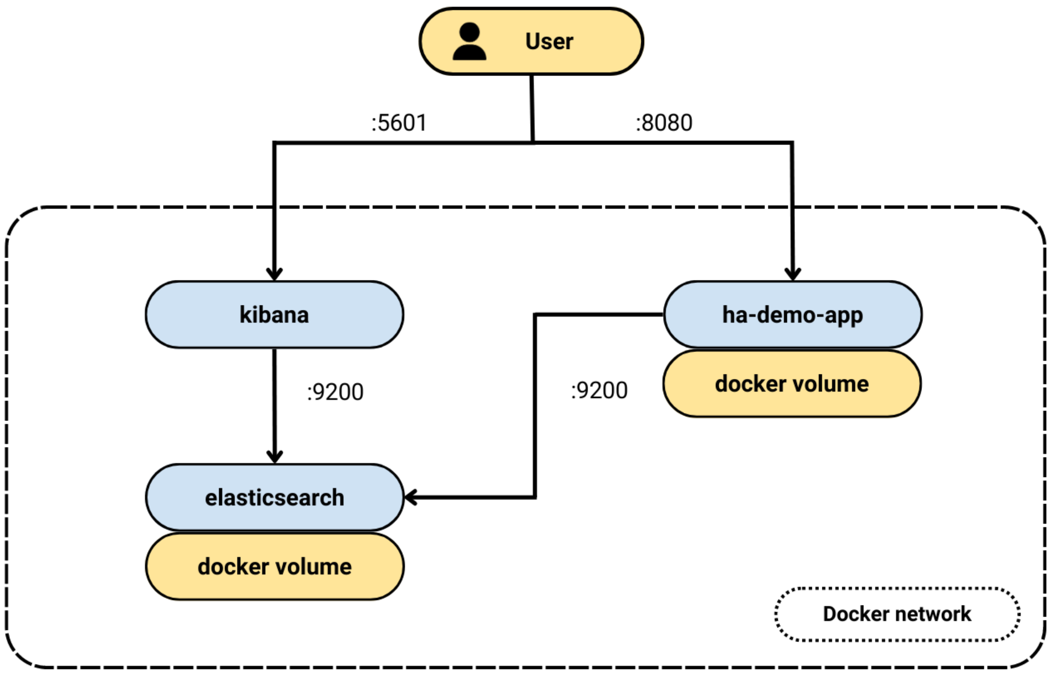


Figure 19 Visual representation of network with ha-demo-app and elastic stack

2.6. Declarative approach

So far, all the examples have been presented using imperative methods. Imperative configuration involves creating resources directly at the command line, which can be difficult to manage and maintain. This approach can make it challenging to recreate an imperatively created state, as it can be hard to keep track of the exact steps that were taken. Additionally, rolling back to a previous state could be problematic, as it may be challenging to recreate the previous configuration accurately. Overall, using an imperative approach can be tedious, error-prone, and may not be as flexible or scalable as a declarative approach.

Declarative configuration involves defining resources and their desired state through a manifest file. It enables the specification of application services, networks, and volumes configuration in a clear manner. Therefore, it enhances understanding and maintenance of the application configuration and facilitates scaling and modification as required. Keeping the manifest file in a version-controlled repository allows changes to the application configuration to be tracked, and previous versions can be easily rolled back if needed. The declared state of the application can then be managed and set up automatically by tools like Docker Compose.

```yaml
version: "3.8"
services:
  ha-demo-app:
    build:
      context: .
    networks:
      - ha-demo
    ports:
      - 8080:8080
    volumes:
      - ha-demo-app-logs:/app/logs
  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:8.5.0
    environment:
      - xpack.security.enabled=false
      - "discovery.type=single-node"
    networks:
      - ha-demo
  kibana:
    image: docker.elastic.co/kibana/kibana:8.5.0
    environment:
      - ELASTICSEARCH_HOSTS=http://elasticsearch:9200
    networks:
      - ha-demo
    depends_on:
      - elasticsearch
    ports:
      - 5601:5601
networks:
  ha-demo:
volumes:
  ha-demo-app-logs:
```

Snippet 11 Docker compose file for ha-demo-app (version 2.5)

Note: By default, Docker Compose creates a separate network for each project. The network is named based on the "project name," derived from the Compose file's directory. If you plan to have multiple projects that need to communicate with each other, it may be necessary to create a network beforehand and connect the containers within each project to that network. For more information, please refer to the documentation on networking in Docker Compose[11] and the Docker Compose file reference[12].

[11] https://docs.docker.com/compose/networking/
[12] https://docs.docker.com/compose/compose-file/compose-file-v3/

3. Orchestration

Orchestration is a way of managing a large environment of software and hardware components that make up a computing system, such as a network of servers. The role of an orchestrator is to maintain a desired state with as little human interaction as possible. The maintenance typically involves monitoring various components and making adjustments as necessary to keep them in line with the desired configuration. It may include starting and stopping services, scaling up or down the resources allocated to a particular component, or implementing other changes.

An orchestrator typically uses some form of automation, though it is essential to understand that orchestration is not automation. Automation is a task that is only concerned with the present state of the environment. An orchestrator continuously checks if the actual state of an environment matches the desired state. If any external event impacts the environment's actual state, the orchestrator's job is to overcome the problem and bring back the state to a desired state. An example of such an event could be a failed node. In such a situation, an orchestrator should reschedule the application to a different node.

The following chapters will focus on Kubernetes - one of the most popular and influential container orchestrators (Schmeling & Dargatz, 2022).

There are several benefits to using orchestration with Kubernetes. First, it allows for efficient resource management by automatically scheduling and deploying containers on available resources, which can help to optimize the use of computing resources and reduce the need for manual intervention.

Second, orchestration can improve the scalability of applications by making it easy to deploy and manage large numbers of containers, especially in situations where applications may need to scale up or down quickly to meet changing demand.

Third, orchestration can help improve applications' reliability and availability by automatically detecting and replacing failed containers and providing mechanisms for rolling out updates and rollbacks to ensure that applications remain available and stable.

Note: In the following chapters, the term *deployment* and *Deployment* will be used extensively. To clarify, "*Deployment*" with a capital "D" refers to a resource object in Kubernetes. In contrast, "*deployment*" with a lowercase "d" refers to the general process of releasing and updating an application or service in a specific environment.

3.1. Introduction

In Kubernetes, there are two main types of nodes: *master nodes* and *worker nodes*. Master nodes are the "control center" of the Kubernetes cluster. They manage the cluster and expose a RESTful API for the management of the cluster. Master nodes are responsible for scheduling and deploying containers onto worker nodes, and they also monitor the cluster's health and the containers running in the cluster.

The master node comprises several components: the *API server*, *the scheduler*, and the *controller manager*. The *API server* is the main entry point for all administrative tasks and is responsible for the cluster's overall state. The *scheduler* is responsible for deciding which worker node should run a given container. The *controller manager* is responsible for managing the system's state and ensuring that the desired state of the cluster matches the actual state. In a *stacked* topology, the master nodes also run an instance of *etcd* - a database where information about cluster state is stored.

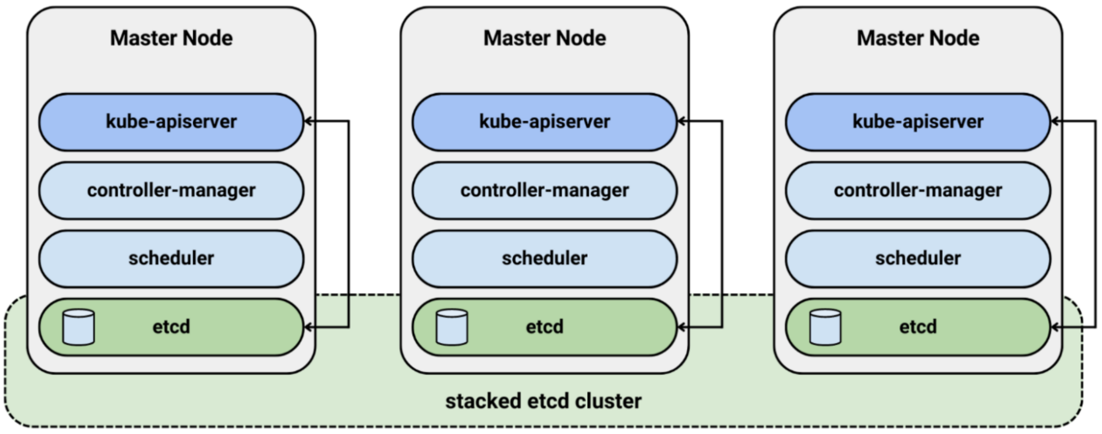


Figure 20 Visual representation of components in a master Kubernetes node

Worker nodes run the containers and communicate with the master node to receive instructions. They have a few components, including the *kubelet*, the primary agent communicating with the master node, and the container runtime responsible for running the containers.

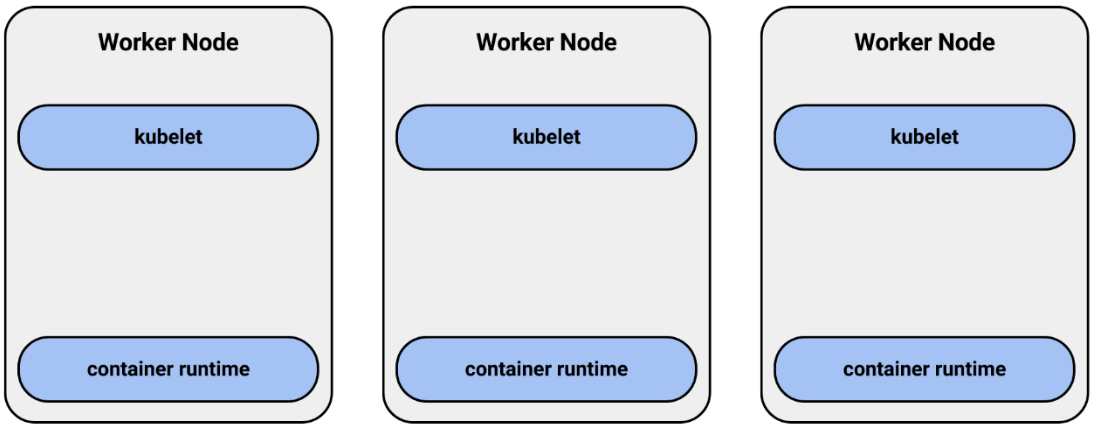


Figure 21 Visual representation of components in a worker Kubernetes node

Since master nodes are responsible for managing the state of the cluster, running multiple master nodes increases the fault tolerance of the cluster. Suppose one of the master nodes fails or becomes unavailable. In that case, the other master nodes can continue to coordinate the cluster and ensure that the containers and services continue to run without interruption.

Not all components of the Kubernetes master node can run in an active-active mode. Only one *controller manager* and one *scheduler* can be active at a given time. Since no data persistence is involved, every process should behave the same way the leader is elected on a first-come basis. On startup, the controller manager and the scheduler try to become a leader by creating a lock on an *Endpoint* object in Kubernetes for a fixed amount of time (by default: 15 seconds). The active instance then renews the lock periodically (by default: every 10 seconds), while the other instance remains in standby mode, periodically trying to acquire the lock (by default: every 2 seconds). If the previous leader fails to renew their lock, another instance can become the new leader.

Etcd is another component that requires special attention when designing a cluster. Since it is a database, keeping a consensus across all the instances is the highest priority. Etcd uses The Raft Consensus Algorithm (Diego Ongaro, John Ousterhout, 2014). The Raft algorithm works by electing a leader among the servers in the system and then using that leader to coordinate and replicate operations across the servers. The leader is responsible for receiving client requests, appending them to its log, and then replicating those logs to the other servers in the system. The servers in the system use a voting process to ensure that only a single leader is active at any given time. They also use a quorum-based approach to ensure that a majority of the servers agree on the contents of the log before it is considered committed. Using the Raft algorithm, a distributed system can ensure that all of its servers have a consistent view of the data, even in the face of network partitions and other failures.

This can help to prevent data inconsistencies and other problems, and it can also make it easier to implement fault-tolerant distributed systems.

The quorum must consist of a majority of the instances; therefore, the number of necessary instances can be expressed using the following formula:

$$Quorum = N/2 + 1$$

Where *N* is the number of total etcd instances.

Table 1 Recommended number of instances in a highly available cluster

Instances	Quorum	Fault tolerance	Split-brain possibility	Recommended
1	1	0	n/a	Yes*
2	2	0	Yes	No*
3	2	1	No	Yes
4	3	1	Yes	No
5	3	2	No	Yes
6	4	2	Yes	No
7	4	3	No	Yes

* no etcd high availability possible

A split-brain situation is where instances are separated into two groups of the same size, and no quorum can be established (see Figure 22). This may happen during network segmentation. To avoid split-brain situations, the number of instances should be odd.

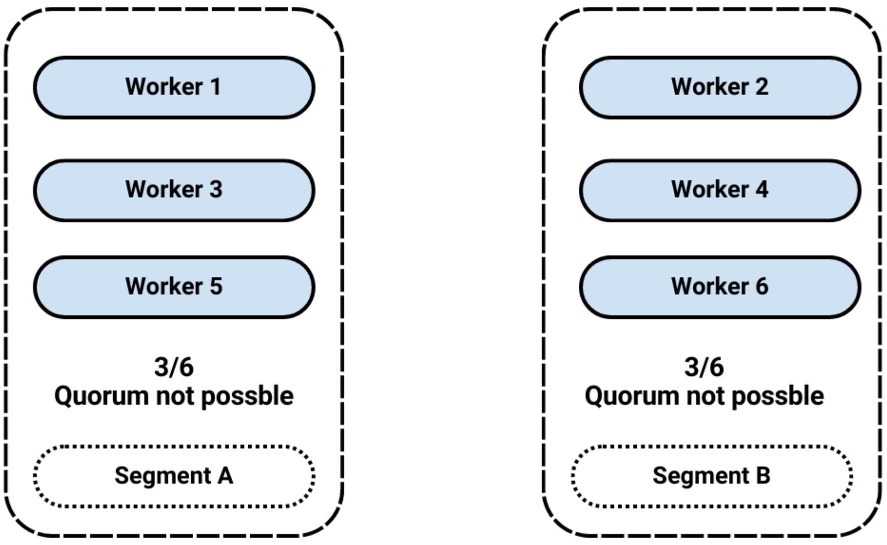


Figure 22 Visual representation of a segmentation

Note: In more sophisticated clusters, multiple schedulers can be used. However, only one scheduler can be used by a single deployment, and the deployment must specify which scheduler should be used. Multiple schedulers are used to provide multiple scheduling logic.

3.2. Container Runtime Interface

To run containers, Kubernetes needs a container runtime. Since Kubernetes is an open-source project maintained and developed by many companies and individuals, it was crucial to provide an unbiased interface between Kubernetes and container runtimes available on the market. Similarly to the runtime specifications settled by the Open Container Initiative, a Container Runtime Interface[13] specification was established to provide a standard set of APIs for interacting with container runtimes, allowing for greater flexibility and interoperability in the Kubernetes ecosystem.

So far, we have thought of Docker as a container runtime engine. When using the docker run command, the command is actually being internally forwarded to the Docker daemon, which invokes containerd, which calls runc. The following Figure represents the projects involved in running a container with Docker:

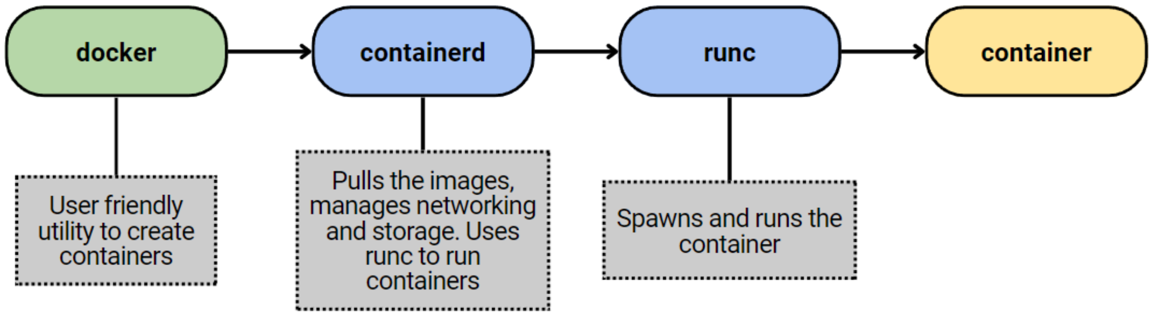

docker
containerd
runc
container
User friendly utility to create containers
Pulls the images, manages networking and storage. Uses runc to run containers
Spawns and runs the container

Figure 23 Visual representation of internal Docker calls

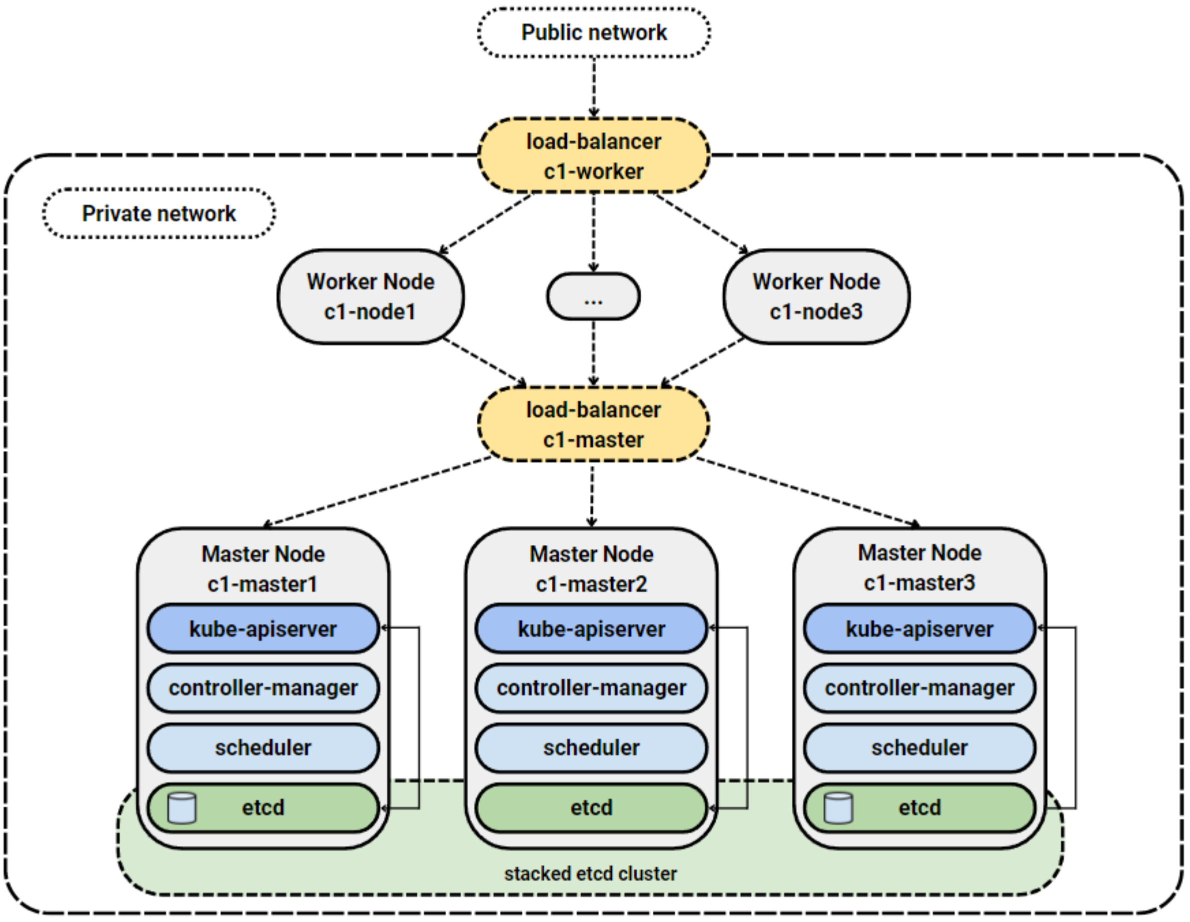
Essentially, docker is just a human-friendly interface for *containerd* and therefore does not conform to the Container Runtime Interface specification. Those human-friendly enhancements are not only unnecessary for Kubernetes but also create another abstraction layer that must be worked around. Moreover, the centralized nature of Docker Engine (single daemon) does not allow for the parallelity required in large clusters.

With that in mind, we will install just the *containerd* on our nodes as it is CRI compatible by itself. Kubernetes themselves will provide a human-friendly interface.

[13] https://github.com/kubernetes/community/blob/master/contributors/devel/sig-node/container-runtime-interface.md

3.3. Cluster creation

Throughout this chapter, we will create a production-grade Kubernetes cluster in an on-premise environment. To create the cluster, we will use a first-party tool called kubeadm. Familiarity with your choice's Linux distribution and basic networking knowledge will be necessary. The finished cluster will resemble the following:



3.3.1. Requirements

To successfully create a cluster, the following criteria must be met:

- Full network connectivity among all machines in the cluster.
- Unique hostname, MAC address, and product_uuid for every node.
- Swap disabled on every node.

If a firewall is present in the network, certain ports have to be open. Refer to the current *Ports and Protocols* [14]reference page.

The following assumptions have been made:

- Every node has a domain name, and a DNS server is present (if no DNS server is available, entries to */etc/hosts* will be sufficient).
- A load balancer is available (if no hardware load balancer is available, a software-defined HTTP(s) load balancer like *nginx* is sufficient).
- Every node has access to the internet during cluster creation.

3.3.2. Environment

For demonstration purposes, an on-premises-like infrastructure has been created within Hetzner Cloud[15]. Hetzner has been chosen based on the following criteria:

- private networking,
- managed load balancers,
- Container Storage Interface driver (discussed in 3.4),
- ready to use Terraform provider,
- speedy VM provisioning,
- custom cloud-init configuration,
- automatic backups,
- hourly billing with no commitment.

[14] https://kubernetes.io/docs/reference/networking/ports-and-protocols/
[15] https://www.hetzner.com/cloud

The following resources have been utilized to create the environment within Hetzner Cloud. A Terraform script can be found in Appendix 1.

Table 2 Resources created in Hetzner Cloud

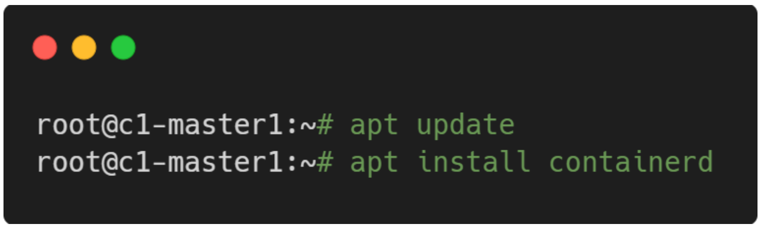
Type	Name	Product	Additional information	Price (ex. VAT)
VM	pfsense*	CX11	Ubuntu 22.04, SSH server	€ 3.29/mo
VM	c1-master1	CPX11	Ubuntu 22.04, master node	€ 3.85/mo
VM	c1-master2	CPX11	Ubuntu 22.04, master node	€ 3.85/mo
VM	c1-master3	CPX11	Ubuntu 22.04, master node	€ 3.85/mo
VM	c1-node1	CX21	Ubuntu 22.04, worker node	€ 4.85/mo
VM	c1-node2	CX21	Ubuntu 22.04, worker node	€ 4.85/mo
VM	c1-node3	CX21	Ubuntu 22.04, worker node	€ 4.85/mo
Load Balancer	c1-master	LB11	Used by Kubernetes API server on master nodes	€ 5.39/mo
Load Balancer	c1-worker	LB11	Used by Ingress on workers (discussed in 3.6)	€ 5.39/mo
Network	Kubernetes	N/A	A private network for internal communication	€ 0.00/mo
Placement group	ha-demo-kubernetes	N/A	Type: spread	€ 0.00/mo
			Total:	€ 40.17/mo € 0.0757/hr

The virtual machine *pfsense* will be used as a DNS server and a virtual router. It does not play a role in the Kubernetes cluster and can be skipped in on-premise environments. A Terraform configuration file is available at the end of this document

Note: Throughout the following chapters, some commands will be given. Be aware that those may vary depending on your Linux distribution choice. In this document, Ubuntu will be assumed.

3.3.3. Node preparations

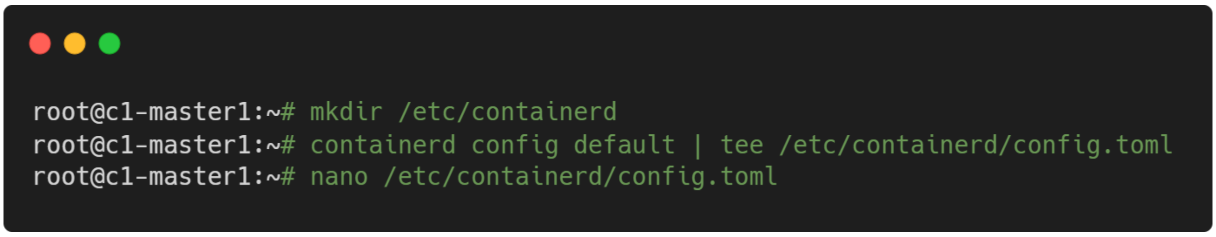
To be able to run containers, we need a container runtime engine. As discussed in chapter 3.2, we will continue with containerd, a lower-level container runtime compared to Docker that conforms to Container Runtime Interface specification. This step will vary based on the preferred Linux distribution, though containerd should be available in most distribution package repositories. For Ubuntu, the following command will install containerd:



```
root@c1-master1:~# apt update
root@c1-master1:~# apt install containerd
```

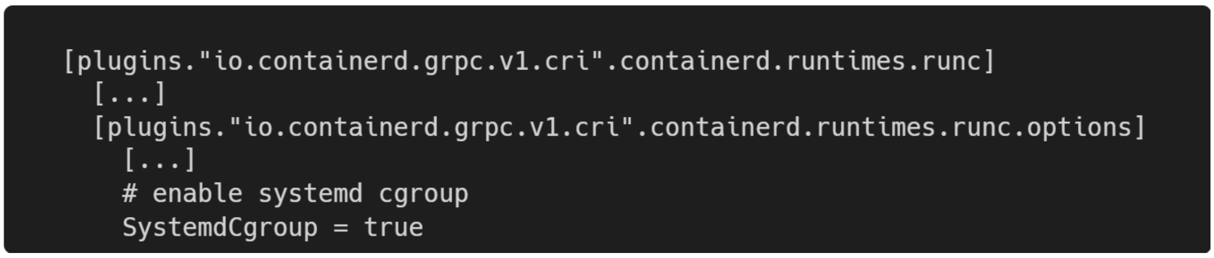
Figure 24 Installing containerd

Control groups, also known as *cgroups*, are used in Linux to manage resources allocated to processes. The kubelet and container runtime need to use control groups to enforce resource limits for pods and containers, such as CPU and memory requests and limits. To do this, they need to use a *cgroup driver*. The kubelet and container runtime must use the same cgroup driver and be configured in the same way. The *cgroupfs* driver is the default cgroup drive. However, the cgroupfs driver is not recommended when using the systemd init system because systemd expects only one cgroup manager. Since Ubuntu is a systemd-based distribution, it is necessary to configure containerd to use systemd as its cgroup driver. This can be accomplished by generating a default containerd configuration file and modifying it in the following way:



```
root@c1-master1:~# mkdir /etc/containerd
root@c1-master1:~# containerd config default | tee /etc/containerd/config.toml
root@c1-master1:~# nano /etc/containerd/config.toml
```

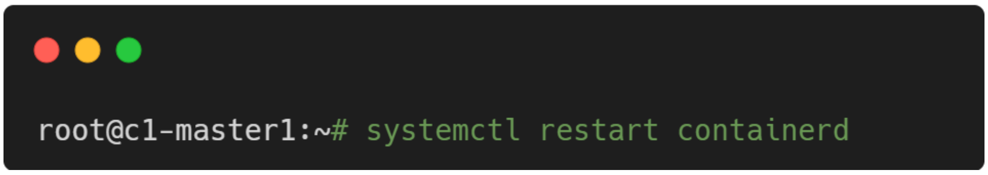
Figure 25 Generating configuration for containerd



```
[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc]
  [...]
  [plugins."io.containerd.grpc.v1.cri".containerd.runtimes.runc.options]
    [...]
    # enable systemd cgroup
    SystemdCgroup = true
```

Figure 26 Modyfing containerd to enable systemd cgroups

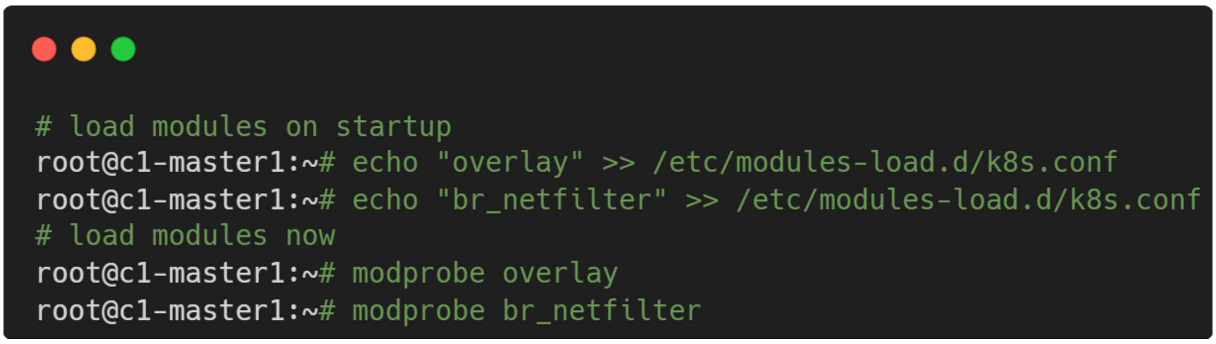
To apply the changes, we have to restart the containerd



```
root@c1-master1:~# systemctl restart containerd
```

Figure 27 Restarting containerd to apply the new configuration

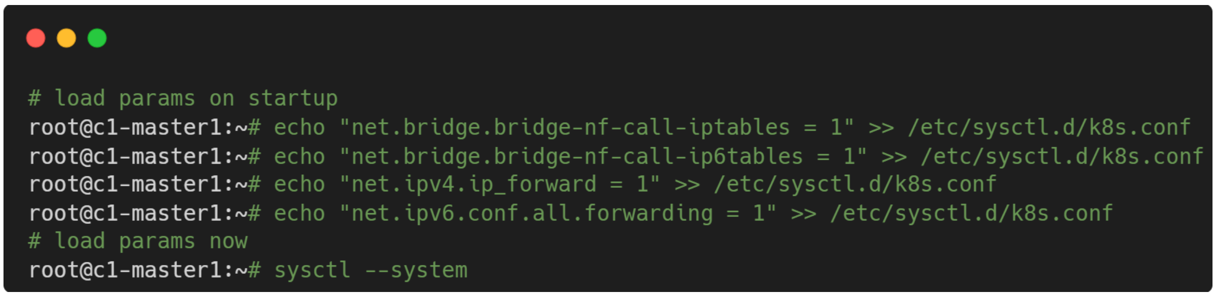
Kubernetes requires two kernel modules to be enabled - *overlay* and *br_netfilter*. This can be achieved in the following manner:



```
# load modules on startup
root@c1-master1:~# echo "overlay" >> /etc/modules-load.d/k8s.conf
root@c1-master1:~# echo "br_netfilter" >> /etc/modules-load.d/k8s.conf
# load modules now
root@c1-master1:~# modprobe overlay
root@c1-master1:~# modprobe br_netfilter
```

Figure 28 Enabling overlay and br_netfilter kernel modules

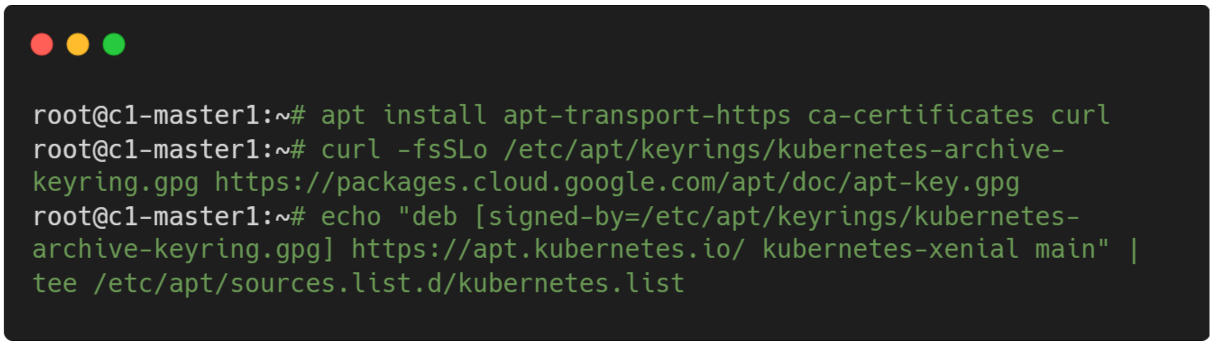
Kubernetes also require three sysctl params that enable IPv4 forwarding visible to iptables:



```
# load params on startup
root@c1-master1:~# echo "net.bridge.bridge-nf-call-iptables = 1" >> /etc/sysctl.d/k8s.conf
root@c1-master1:~# echo "net.bridge.bridge-nf-call-ip6tables = 1" >> /etc/sysctl.d/k8s.conf
root@c1-master1:~# echo "net.ipv4.ip_forward = 1" >> /etc/sysctl.d/k8s.conf
root@c1-master1:~# echo "net.ipv6.conf.all.forwarding = 1" >> /etc/sysctl.d/k8s.conf
# load params now
root@c1-master1:~# sysctl --system
```

Figure 29 Enabling IPv4 forwarding

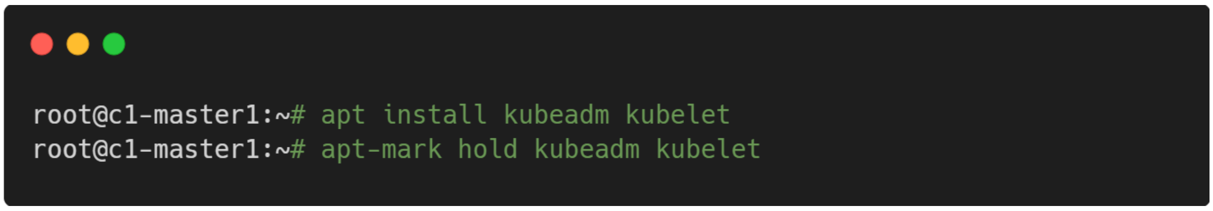
To bootstrap a Kubernetes cluster with *kubeadm*, all nodes must have installed kubeadm and kubelet. The recommended way to do this is to add the Kubernetes package repository to the package manager. This can be done in the following manner:



```
root@c1-master1:~# apt install apt-transport-https ca-certificates curl
root@c1-master1:~# curl -fsSLo /etc/apt/keyrings/kubernetes-archive-
keyring.gpg https://packages.cloud.google.com/apt/doc/apt-key.gpg
root@c1-master1:~# echo "deb [signed-by=/etc/apt/keyrings/kubernetes-
archive-keyring.gpg] https://apt.kubernetes.io/ kubernetes-xenial main" |
tee /etc/apt/sources.list.d/kubernetes.list
```

Figure 30 Adding Kubernetes package repository

We can now proceed and install both *kubeadm* and *kubelet*. Since both kubeadm and kubelet require a special procedure to update, putting them in a hold state is recommended, so they will not be considered when performing an automatic upgrade of all packages.



```
root@c1-master1:~# apt install kubeadm kubelet
root@c1-master1:~# apt-mark hold kubeadm kubelet
```

Figure 31 Installing kubeadm and kubelet

It is necessary to install and configure containerd, kubeadm, and kubelet on all cluster nodes. Repeat the above steps as many times as necessary. You may also want to install kubectl on your computer at this stage. It is a command line tool for interacting with Kubernetes clusters.

3.3.4. Cluster Networking

To create a highly available cluster, we need multiple *etcd* instances. In a stacked topology, the *etcd* instances will be created as *static pods*. Static pods are containers running inside a cluster, except that they are created and managed directly by the kubelet rather than the Kubernetes API server. A network plugin is necessary to establish communication between pods in a cluster.

Similarly to Container Runtime Interface, a Container Network Interface defines a set of standards for configuring network interfaces for containers, including how to assign IP addresses, route traffic between containers, and how to connect containers to external networks. There are many CNI plugins available on the market. Some popular ones include

Calico[16], Flannel[17], Weavenet[18] , and Cilium[19]. In an article from Mehndiratta, the differences between available CNI plugins have been discussed (2021). Mehndiratta summarized the differences in a single table:

Table 3 Comparision of available CNI plugins (Mehndiratta, 2021)

CNI/Feature	Flannel	Calico	Cilium	Weavenet
Encapsulation and Routing	VxLAN	IPinIP, BGP, eBPF	VxLAN	VxLAN
Support for Network Policies	No	Yes	Yes	Yes
Datastore used	etcd	etcd	etcd	No
Encryption	Yes	Yes	Yes	Yes
Ingress Support	No	Yes	Yes	Yes
Enterprise Support	No	Yes	No	Yes

Since we will be using Ingress to access our application from outside of the cluster, we will continue with Calico as our networking plugin. Calico is also feature rich and has a relatively low entry level.

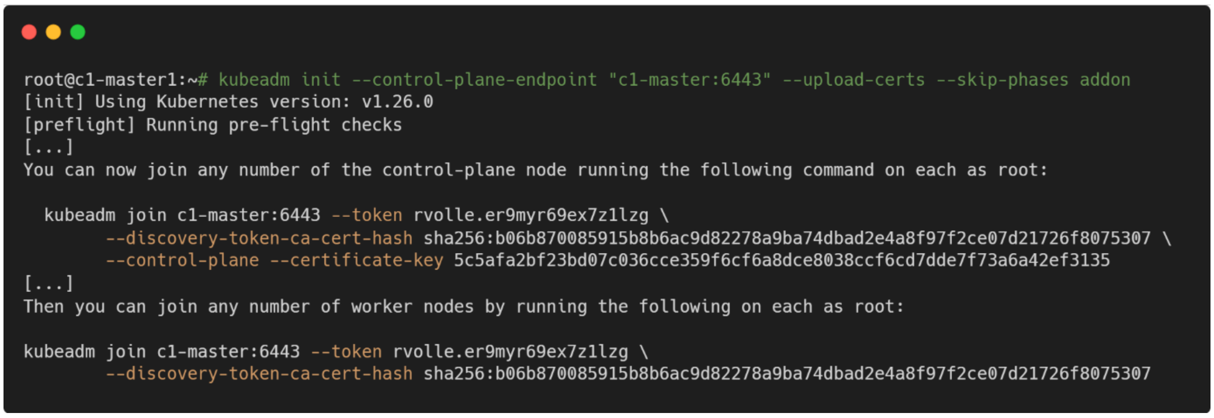
Note: Mehndiratta's article also mentions the Canal[20] CNI plugin, though it is no longer supported or maintained by its creators.

3.3.5. Cluster Initialization

To initialize a Kubernetes cluster, we have to issue the kubeadm init command on one of our master nodes. The cluster initialization is divided into multiple phases by kubeadm. The addon phase requires the Kubernetes API server to be accessible. The load balancer in use only targets servers considered healthy; in essence, they must have answered to one or more health checks. Since the health checking operation takes a while, it may happen that the addon phase will fail, and the cluster will not be properly initialized. To overcome this problem, we can tell kubeadm to skip this phase and run it manually later.

[16] https://www.tigera.io/project-calico/
[17] https://github.com/flannel-io/flannel
[18] https://www.weave.works/oss/net/
[19] https://cilium.io/
[20] https://github.com/projectcalico/canal

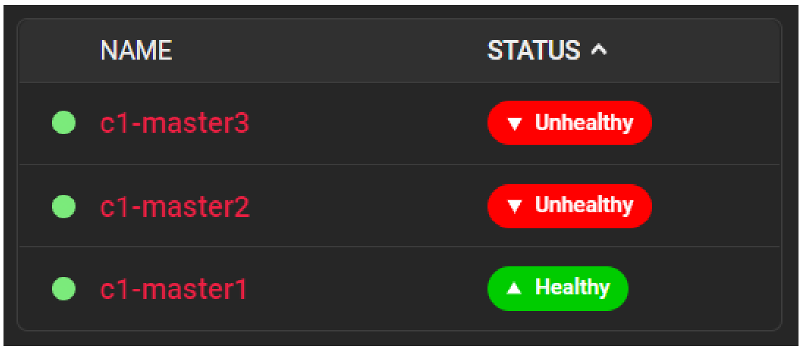
Moreover, to create a highly available cluster, we have to pass the *--control-plane-endpoint* argument with a DNS name of our master load balancer. Using the --upload-certs argument, we can also configure the cluster to share the generated certificates across all control-plane nodes automatically. By default, Kubernetes will use the 10.96.0.0/12 subnet as *service cidr*. At the end, kubeadm will return a command that can be used on other nodes to join the cluster.



```
root@c1-master1:~# kubeadm init --control-plane-endpoint "c1-master:6443" --upload-certs --skip-phases addon
[init] Using Kubernetes version: v1.26.0
[preflight] Running pre-flight checks
[...]
You can now join any number of the control-plane node running the following command on each as root:

  kubeadm join c1-master:6443 --token rvolle.er9myr69ex7z1lzg \
        --discovery-token-ca-cert-hash sha256:b06b870085915b8b6ac9d82278a9ba74dbad2e4a8f97f2ce07d21726f8075307 \
        --control-plane --certificate-key 5c5afa2bf23bd07c036cce359f6cf6a8dce8038ccf6cd7dde7f73a6a42ef3135
[...]
Then you can join any number of worker nodes by running the following on each as root:

kubeadm join c1-master:6443 --token rvolle.er9myr69ex7z1lzg \
        --discovery-token-ca-cert-hash sha256:b06b870085915b8b6ac9d82278a9ba74dbad2e4a8f97f2ce07d21726f8075307
```

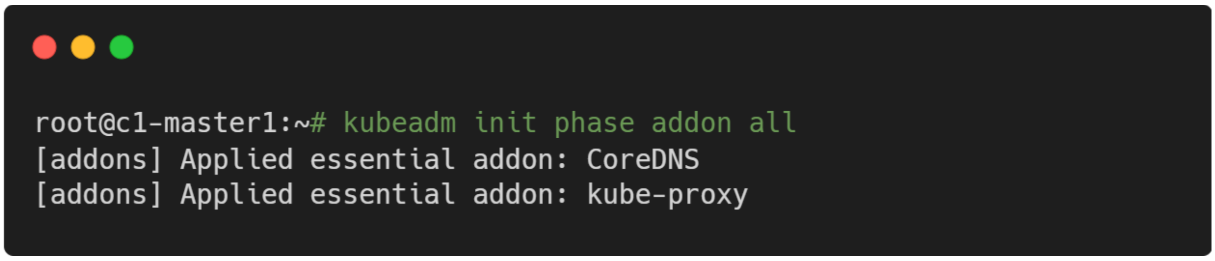
Figure 32 Initializing Kubernetes cluster

Once the first target of the load balancer becomes *healthy,* we can proceed with the *addon* phase installation:



NAME	STATUS ^
● c1-master3	▼ Unhealthy
● c1-master2	▼ Unhealthy
● c1-master1	▲ Healthy

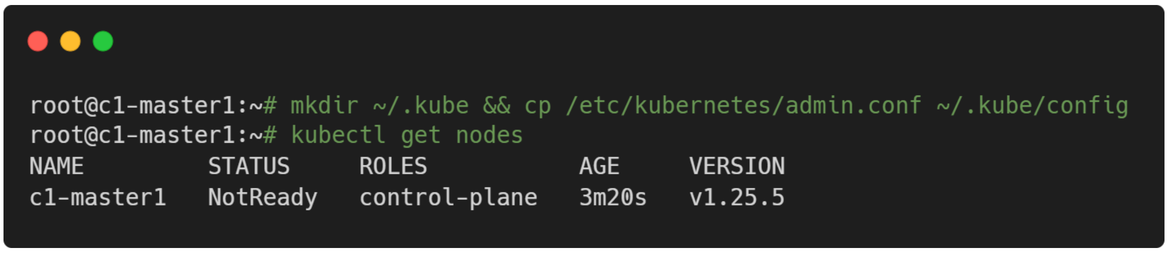
Figure 33 Load balancer status after initializing the first master node



```
root@c1-master1:~# kubeadm init phase addon all
[addons] Applied essential addon: CoreDNS
[addons] Applied essential addon: kube-proxy
```

Figure 34 Initializing addons

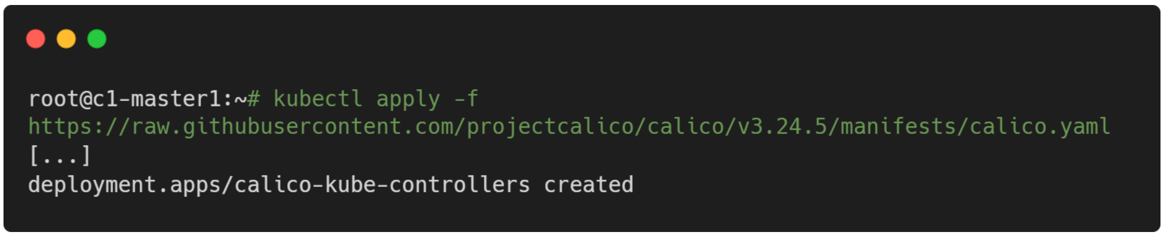
We can now copy the generated configuration file and access the Kubernetes cluster.



```
root@c1-master1:~# mkdir ~/.kube && cp /etc/kubernetes/admin.conf ~/.kube/config
root@c1-master1:~# kubectl get nodes
NAME         STATUS     ROLES           AGE     VERSION
c1-master1   NotReady   control-plane   3m20s   v1.25.5
```

Figure 35 Accessing cluster for the first time

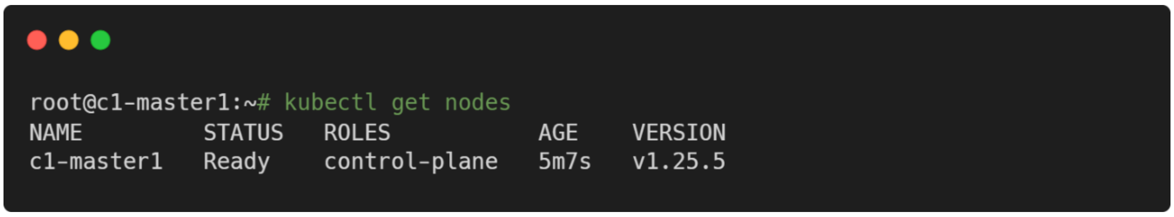
The node is not yet in a *Ready* state, as we still must install the Container Network Interface plugin. This step will vary depending on the CNI plugin of choice. For Calico, the procedure to install the CNI on clusters with less than 50 nodes is to apply one manifest:



```
root@c1-master1:~# kubectl apply -f
https://raw.githubusercontent.com/projectcalico/calico/v3.24.5/manifests/calico.yaml
[...]
deployment.apps/calico-kube-controllers created
```

Figure 36 Applying Calico

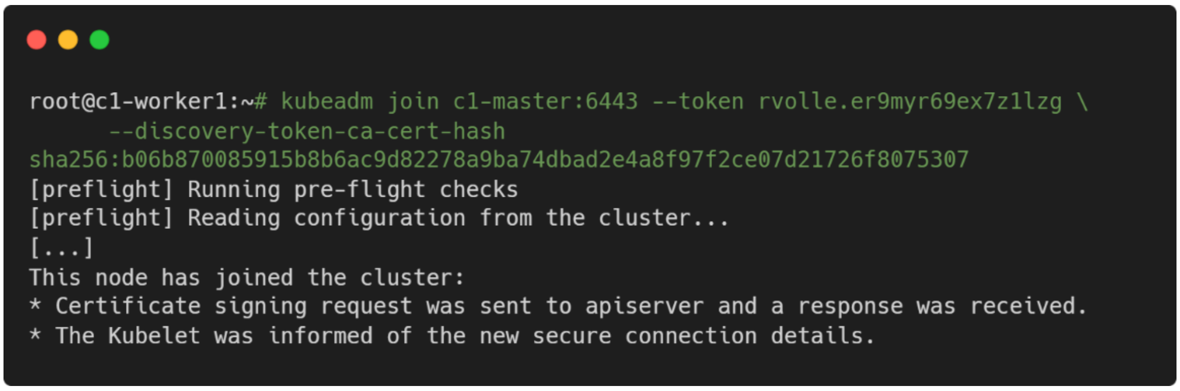
A few minutes later, the node should become *Ready*:



```
root@c1-master1:~# kubectl get nodes
NAME         STATUS   ROLES           AGE    VERSION
c1-master1   Ready    control-plane   5m7s   v1.25.5
```

Figure 37 Verifying state of the first master node

We can now add other nodes using the *kubeadm join* command given in Figure 32. On a worker node, the result should resemble the following:



```
root@c1-worker1:~# kubeadm join c1-master:6443 --token rvolle.er9myr69ex7z1lzg \
      --discovery-token-ca-cert-hash
sha256:b06b870085915b8b6ac9d82278a9ba74dbad2e4a8f97f2ce07d21726f8075307
[preflight] Running pre-flight checks
[preflight] Reading configuration from the cluster...
[...]
This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.
```

Figure 38 Joining other nodes to the cluster

After a while, the *kubectl get nodes should report all nodes as Ready:*

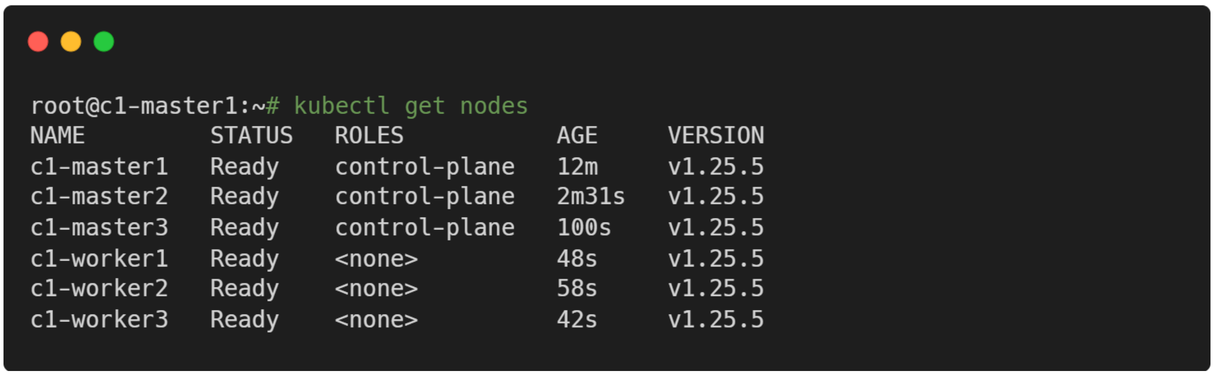

```
root@c1-master1:~# kubectl get nodes
NAME          STATUS   ROLES           AGE     VERSION
c1-master1    Ready    control-plane   12m     v1.25.5
c1-master2    Ready    control-plane   2m31s   v1.25.5
c1-master3    Ready    control-plane   100s    v1.25.5
c1-worker1    Ready    <none>          48s     v1.25.5
c1-worker2    Ready    <none>          58s     v1.25.5
c1-worker3    Ready    <none>          42s     v1.25.5
```

Figure 39 Inspecting the state of all nodes

Note: By default, Kubernetes will create services in the 10.96.0.0/12 subnet, and the pod subnet will be determined by the CNI plugin (Calico uses 192.168.0.0/16). To avoid network overlap, changing these based on the organization's existing infrastructure may be necessary. During cluster initialization, the --service-cidr and --pod-network-cidr arguments can be passed to kubeadm. It is important to note that the --pod-network-cidr is merely metadata that may or may not be read by the CNI driver. Further information can be found in the documentation of the CNI driver.

3.4. Persistent Volumes

When implementing a cluster infrastructure utilizing persistent storage, it is vital to ensure that the data is made accessible to all nodes in the system. Various distributed storage platforms are available in the market, with some of the more popular on-premise options being Ceph [21]and GlusterFS[22]. In addition, there are also emerging technologies, such as Rook[23], which are gaining traction in the field. Most major cloud providers offer a means of installing a Container Storage Interface (CSI) driver or provide pre-configured Storage Classes. In cases where automatic provisioning is not required, a simple Network File System (NFS) share may be a viable solution.

In Docker, two key terms related to data persistence are *volume* and *storage driver*. The storage driver manages the lifecycle of a volume, which can be mounted to a container. Kubernetes distinguishes between *Persistent Volumes, Persistent Volume Claims,* and *Storage Classes*. A

[21] https://ceph.io/en/
[22] https://www.gluster.org/
[23] https://rook.io/

Persistent Volume is a piece of storage that has been provisioned and made available to the cluster. In contrast, a Persistent Volume Claim is a request for storage with certain specifications such as capacity, access modes, or storage class. Storage classes describe the characteristics of a particular class of storage. They can be used to specify things such as the type of storage (e.g., SSD or HDD), the level of durability or availability, and the performance characteristics of the storage. Storage Classes can be bound to a *provisioner*.

The Container Storage Interface (CSI) serves as a specification for communication between the orchestrator and the storage provider, similar in function to the Container Runtime and Container Network Interfaces. This specification enables storage providers to create plugins capable of dynamic storage provisioning.

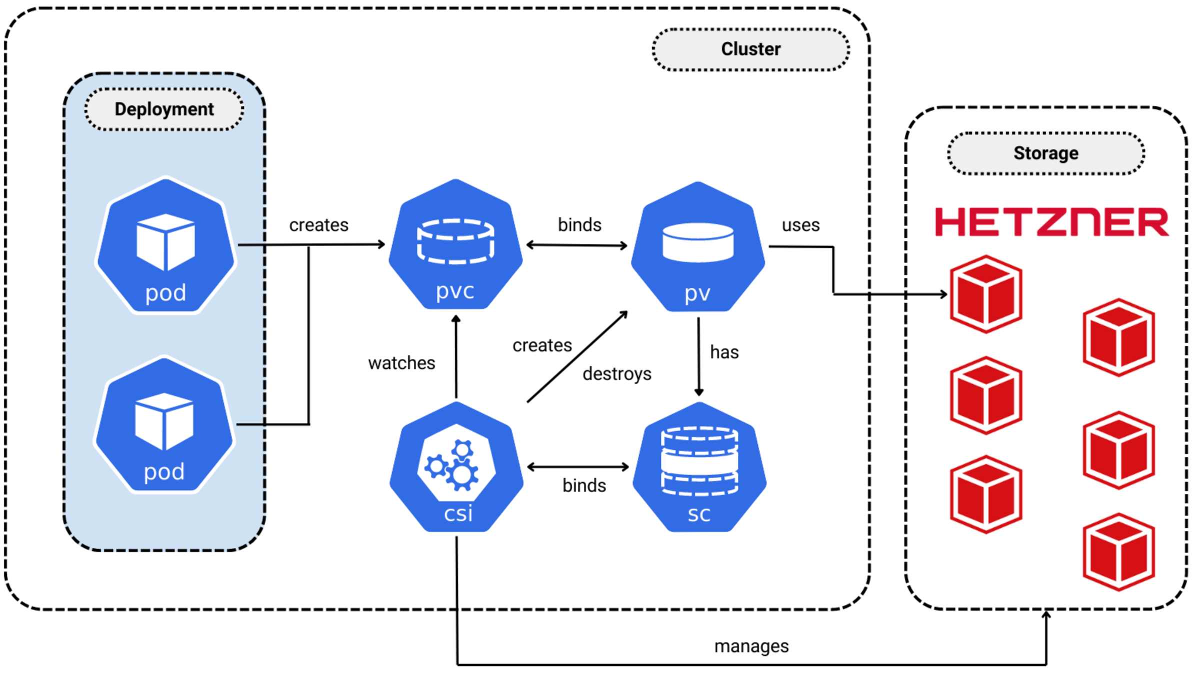
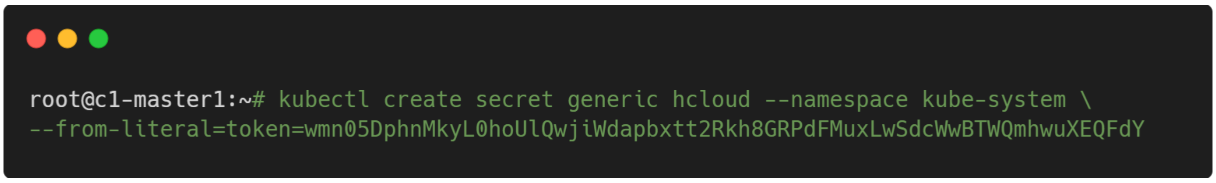


Figure 40 Visual representation of storage components in Kubernetes

The installation of Hetzner's CSI driver is a two-step process - creating a secret containing the API key and applying a single manifest. The manifest will create a Storage Class with the name *hcloud-volumes*, a *Service Account* with *Cluster Role* granting permission to watch and modify objects related to Persistent Volumes. A Deployment will then use the Service Account with a

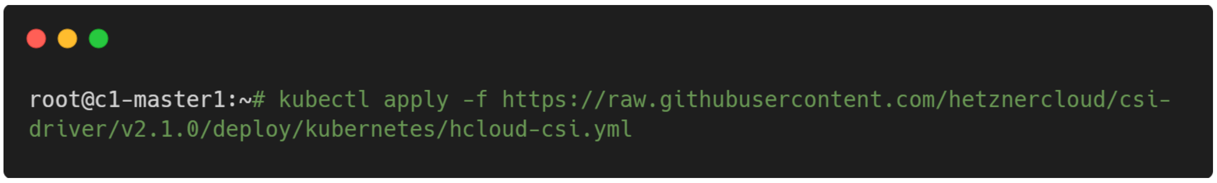
CSI controller and a *DaemonSet* that will run an agent on every node. The secret can be created in the following way:



```
root@c1-master1:~# kubectl create secret generic hcloud --namespace kube-system \
--from-literal=token=wmn05DphnMkyL0hoUlQwjiWdapbxtt2Rkh8GRPdFMuxLwSdcWwBTWQmhwuXEQFdY
```

Figure 41 Creating secret for Hetzner Cloud API

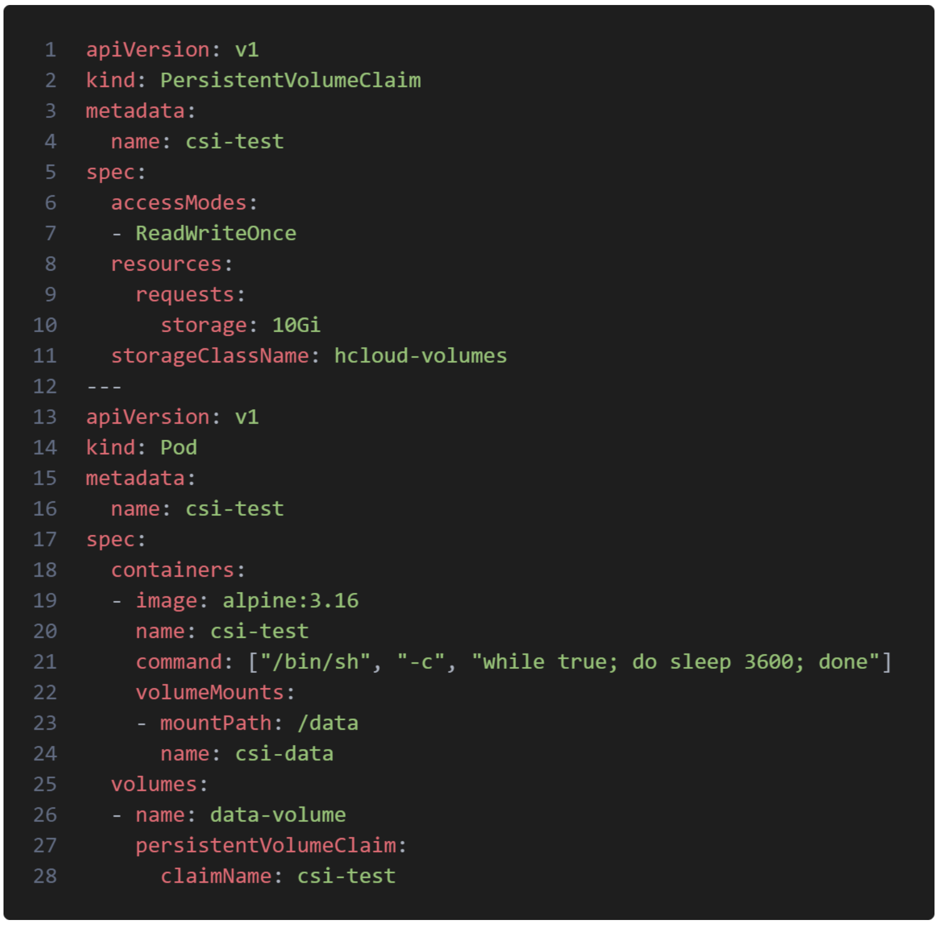
Furthermore, the manifest can be applied similarly:



```
root@c1-master1:~# kubectl apply -f https://raw.githubusercontent.com/hetznercloud/csi-
driver/v2.1.0/deploy/kubernetes/hcloud-csi.yml
```

Figure 42 Applying Hetzer Cloud CSI Driver

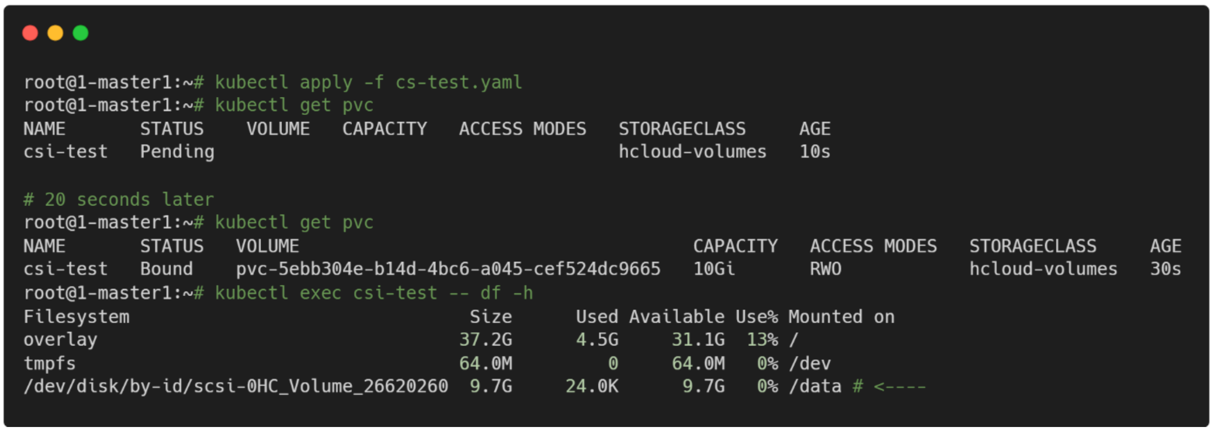
To test if the volume is being provisioned, we can create a PVC and a Pod mounting that volume:



```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: csi-test
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: hcloud-volumes
---
apiVersion: v1
kind: Pod
metadata:
  name: csi-test
spec:
  containers:
  - image: alpine:3.16
    name: csi-test
    command: ["/bin/sh", "-c", "while true; do sleep 3600; done"]
    volumeMounts:
    - mountPath: /data
      name: csi-data
  volumes:
  - name: data-volume
    persistentVolumeClaim:
      claimName: csi-test
```

Snippet 12 Pod with PersistentVolumeClaim example

We can apply the manifest from Snippet 12 using kubectl:



```
root@1-master1:~# kubectl apply -f cs-test.yaml
root@1-master1:~# kubectl get pvc
NAME       STATUS    VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS     AGE
csi-test   Pending                                      hcloud-volumes   10s

# 20 seconds later
root@1-master1:~# kubectl get pvc
NAME       STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS     AGE
csi-test   Bound    pvc-5ebb304e-b14d-4bc6-a045-cef524dc9665   10Gi       RWO            hcloud-volumes   30s
root@1-master1:~# kubectl exec csi-test -- df -h
Filesystem                            Size      Used Available Use% Mounted on
overlay                              37.2G      4.5G     31.1G  13% /
tmpfs                                64.0M         0     64.0M   0% /dev
/dev/disk/by-id/scsi-0HC_Volume_26620260  9.7G     24.0K      9.7G   0% /data # <----
```

Figure 43 Verifying creation of Persistent Volume

3.5. Pod versus Deployment

Pods in Kubernetes are not managed in the sense that they do not have any built-in self-healing or rescheduling capabilities. Once a Pod is created, it will continue to run until it is manually deleted or encounters an error that causes it to crash. If Pod crashes or becomes unavailable, it will not be automatically rescheduled or replaced.

To achieve high availability, it is necessary to use a higher-level resource such as a *Deployment*. A Deployment allows to specify the desired number of replicas for an application and automatically creates, updates, and deletes Pods as needed to maintain that desired state. Moreover, deployments keep track of the applications' different versions and allow them to quickly roll back to a previous version if needed.

Behind the scenes, Deployment uses another mechanism called *ReplicaSet*. The replica set monitors and maintains the desired number of *replicas*. Deployment uses multiple ReplicaSets to keep track of versions, scaling them accordingly.

Updating a Deployment in Kubernetes typically works by using a rolling update strategy. A rolling update allows updating an application by gradually rolling out the new version to a subset of replicas at a time while keeping the rest of the replicas running the previous version. Rolling releases allow for a more controlled and predictable update process and minimize the risk of disruptions to the application. The process typically works as follows:

The Deployment is updated with the new version of the application and the desired number of replicas.

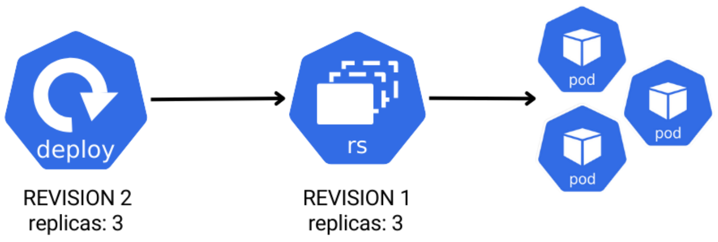


Figure 44 Updated Deployment with old ReplicaSet

The Deployment controller then creates a ReplicaSet with a newer application version.

As new replicas are created, the Deployment controller begins to scale down the replica running the previous version.

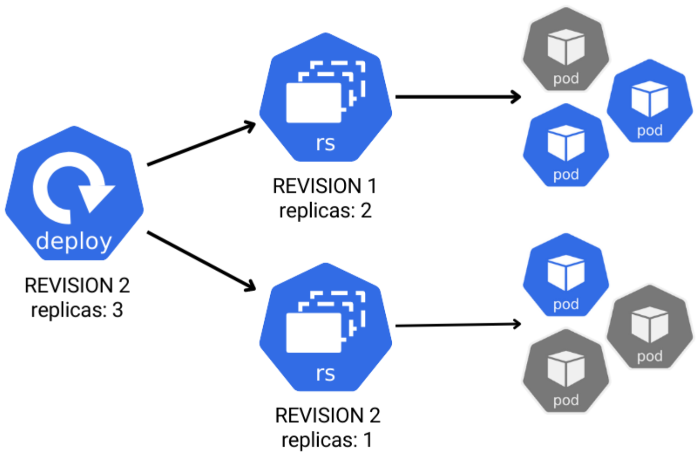


Figure 45 Rolling release procedure

This process continues until the new ReplicaSet reaches the desired number of replicas.

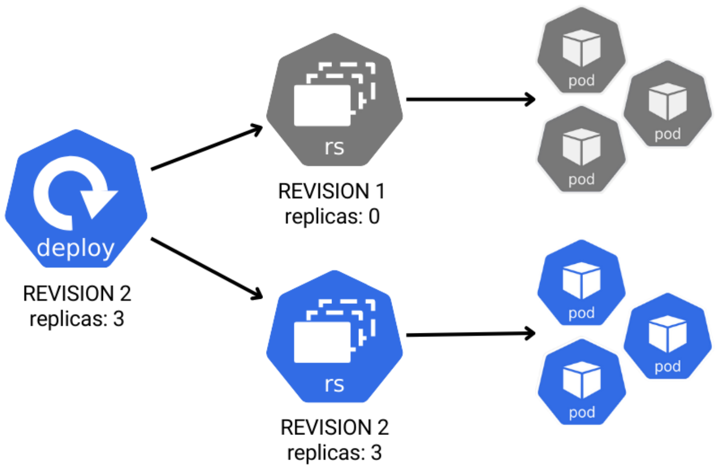


Figure 46 Updated Deployment with new ReplicaSet

If any issues arise during the update process, the Deployment controller will roll back to the previous version.

3.6. High Availability of the "Hello World!" application

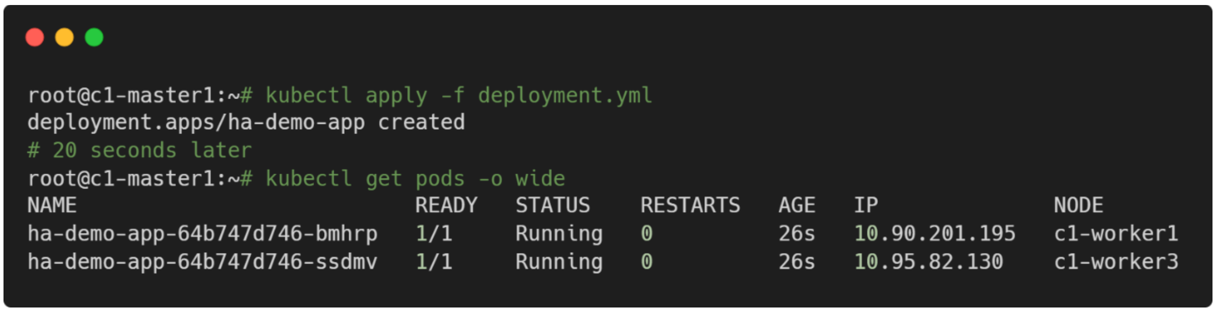
When an application is deployed in Kubernetes using a Deployment resource, it will automatically benefit from high availability as Kubernetes ensures that the desired number of replicas of the application is running at all times, even in case of node failure or application crash. To increase the high availability even further, we can run the application in multiple replicas, preferably on separate machines, so that a node failure does not impact the availability of the application.

The following manifest will create a Deployment of the "Hello World!" application with two replicas. The *podAntiAffinity* will try to distribute the pods among available nodes.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ha-demo-app
  labels:
    app: "ha-demo-app"
spec:
  replicas: 2
  selector:
    matchLabels:
      app: "ha-demo-app"
  template:
    metadata:
      labels:
        app: "ha-demo-app"
    spec:
      affinity:
        podAntiAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 20
            podAffinityTerm:
              labelSelector:
                matchExpressions:
                - key: app
                  operator: In
                  values:
                  - "ha-demo-app"
              topologyKey: "kubernetes.io/hostname"
      containers:
      - image: registry.gitlab.com/mrpsycho5/ha-demo-app:3.6
        name:  ha-demo-app
        ports:
        - name: www
          containerPort: 8080
```

Snippet 13 Deployment manifest for ha-demo-app

We can apply the deployment using kubectl in the following way:



```
root@c1-master1:~# kubectl apply -f deployment.yml
deployment.apps/ha-demo-app created
# 20 seconds later
root@c1-master1:~# kubectl get pods -o wide
NAME                           READY   STATUS    RESTARTS   AGE   IP              NODE
ha-demo-app-64b747d746-bmhrp   1/1     Running   0          26s   10.90.201.195   c1-worker1
ha-demo-app-64b747d746-ssdmv   1/1     Running   0          26s   10.95.82.130    c1-worker3
```

Figure 47 Applying the ha-demo-app Deployment

3.7. Services

Services are used to expose multiple Pods running inside the cluster with a stable endpoint. When multiple pods are available, a service works as a load balancer targeting only healthy Pods. Kubernetes automatically creates a DNS entry for each service and allows other components in the cluster to easily discover and communicate with the Pods associated with the service. Services make it easy to scale and update the underlying Pods without affecting the communication to the service. Moreover, Services can redirect ports.

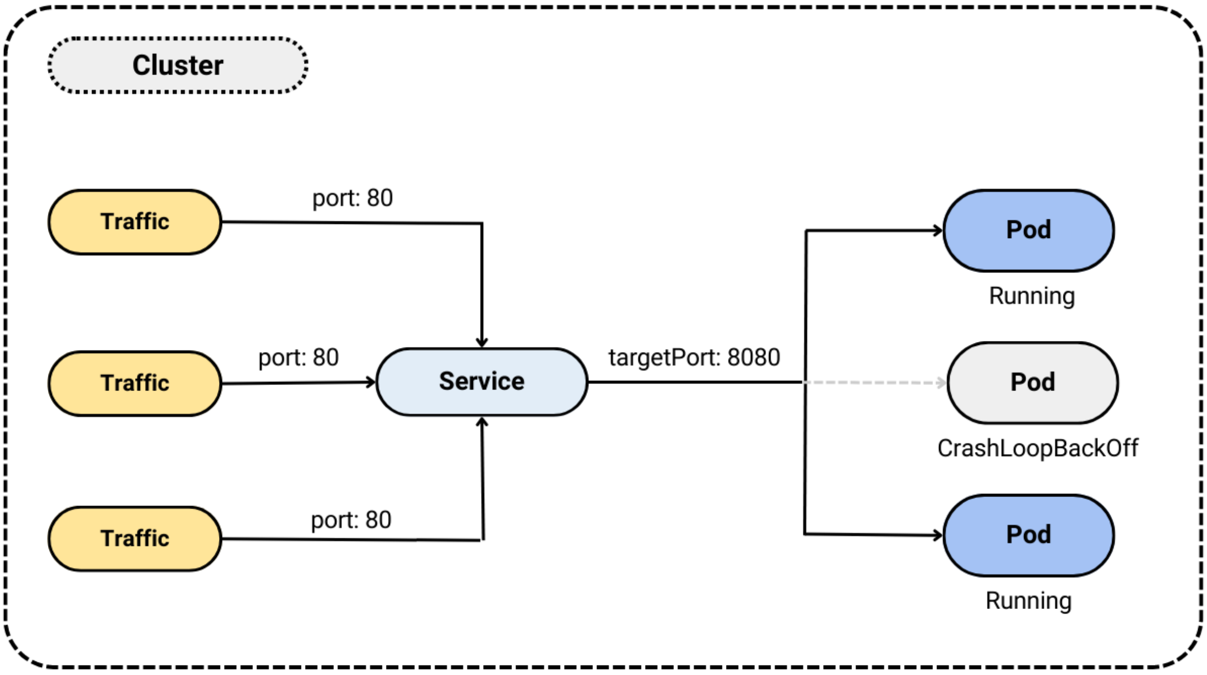
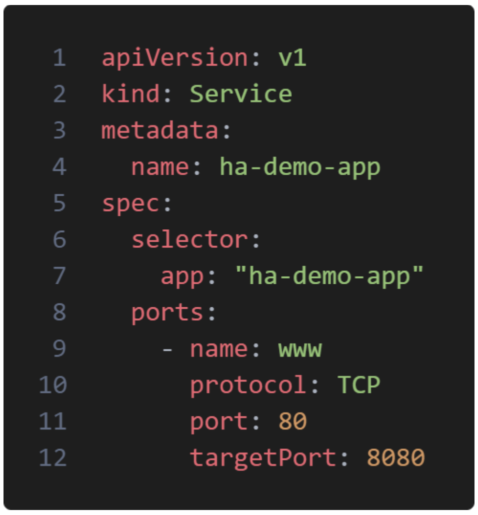


Figure 48 Visual representation of a Service

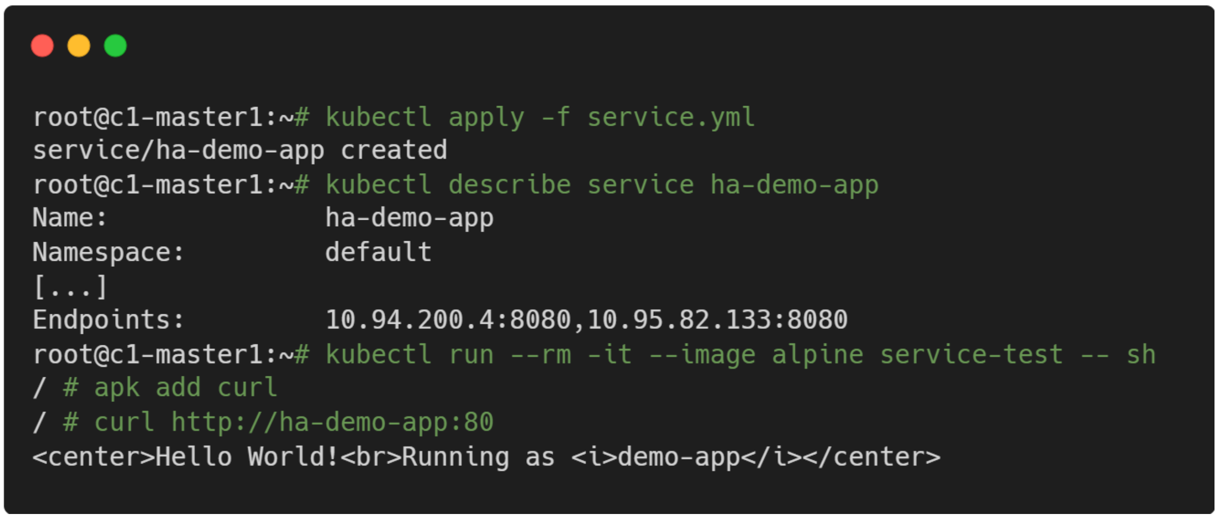
Services target pods based on labels. In Snippet 13, we have added the label *app=ha-demo-app* to the pods inside the deployment. We can now use those labels as a selector for our service:



```yaml
apiVersion: v1
kind: Service
metadata:
  name: ha-demo-app
spec:
  selector:
    app: "ha-demo-app"
  ports:
    - name: www
      protocol: TCP
      port: 80
      targetPort: 8080
```

Snippet 14 Service manifest for ha-demo-app

After applying that manifest, we can *describe* the service object to verify that it has two *endpoints*. If we create another pod in the cluster, we should also be able to access our application through the service.



```
root@c1-master1:~# kubectl apply -f service.yml
service/ha-demo-app created
root@c1-master1:~# kubectl describe service ha-demo-app
Name:              ha-demo-app
Namespace:         default
[...]
Endpoints:         10.94.200.4:8080,10.95.82.133:8080
root@c1-master1:~# kubectl run --rm -it --image alpine service-test -- sh
/ # apk add curl
/ # curl http://ha-demo-app:80
<center>Hello World!<br>Running as <i>demo-app</i></center>
```

Figure 49 Applying the ha-demo-app Service

3.8. Ingress

Ingress allows external traffic to be routed to one or more services within the cluster. It acts as a reverse proxy, routing incoming traffic to the appropriate service based on rules defined in the Ingress resource. Additionally, Ingress can be used to provide SSL/TLS termination.

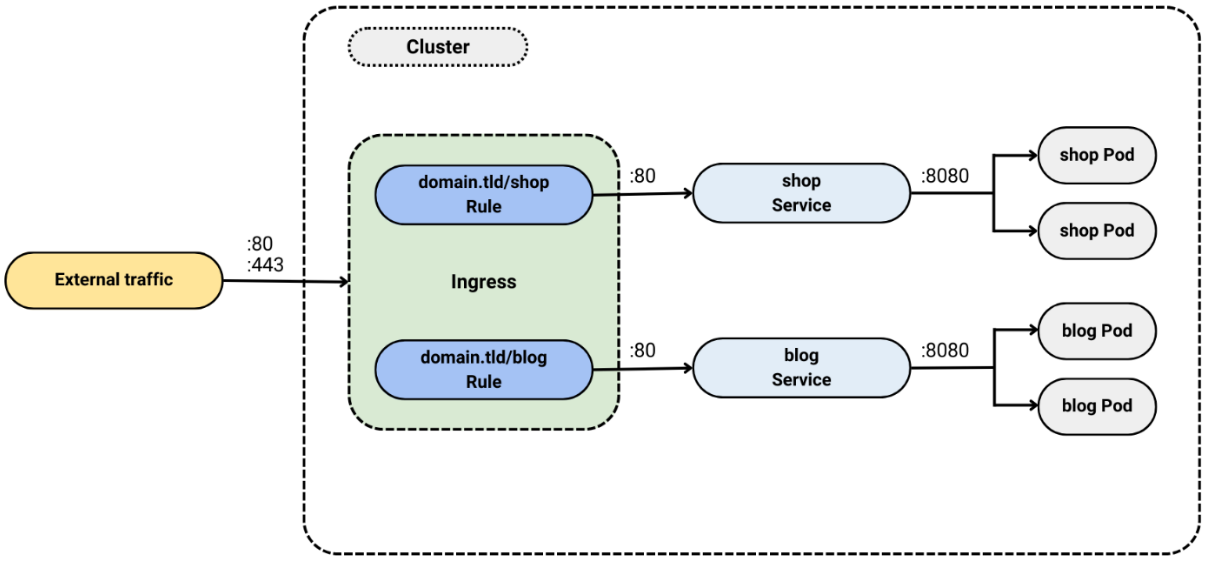


Figure 50 Visual representation of how Ingress routes external traffic to multiple services

Kubernetes defines rules for Ingress resources, but like with other components, it is completely agnostic on how the controller is implemented. An Ingress controller is a software component that runs within the Kubernetes cluster and is responsible for enforcing the rules defined in Ingress resources. The Ingress controller listens for changes to the Ingress resources and updates its configuration accordingly. There are numerous Ingress controllers available on the market[24]. The controller market is rapidly evolving, and research is usually necessary to determine which controller is most suitable for an organization. Learnk8s[25] has created an exhaustive Comparison of Kubernetes Ingress controllers. While it is technically possible to deploy multiple controllers in a single cluster[26], it is necessary to include additional fields to the Ingress Resource starting from the first Ingress Resource.

[24] https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/
[25] https://learnk8s.io/research
[26] https://kubernetes.github.io/ingress-nginx/user-guide/multiple-ingress/

3.8.1. NGINX Ingress Controller

We will be continuing with the community-maintained NGINX Ingress Controller[27]. The installation process is similar to both the CNI plugin and CSI driver. Essentially, a manifest file has to be applied. However, we will have to modify the manifest due to our on-premise characteristics. The modified Ingress Controller will resemble the following:

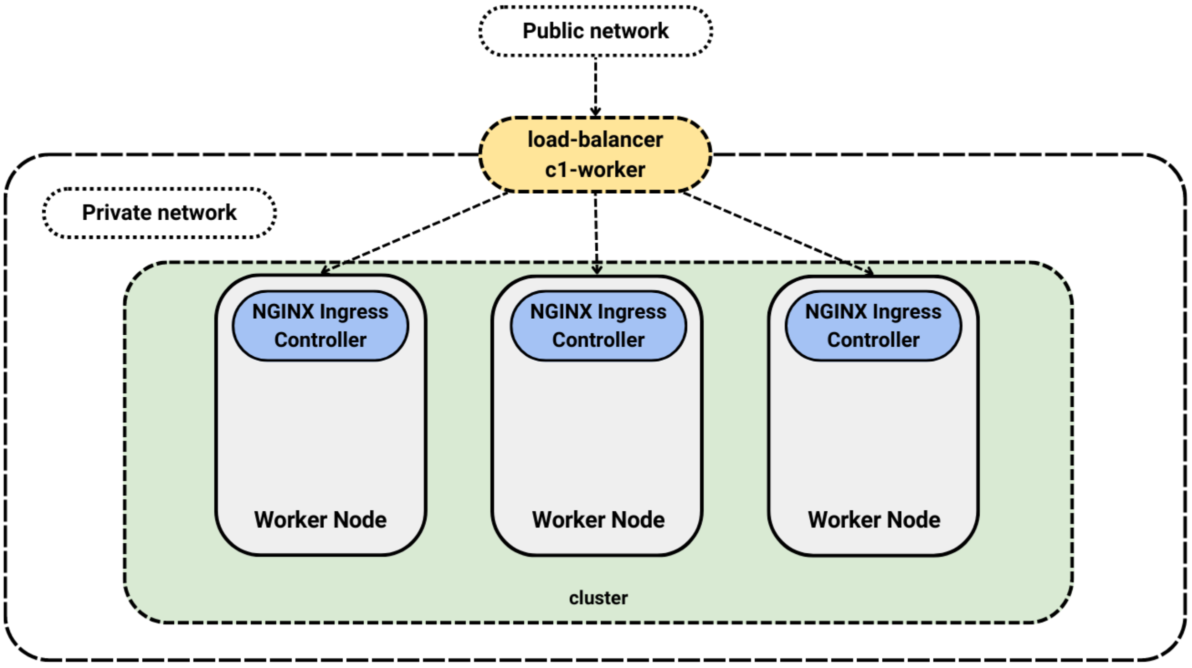
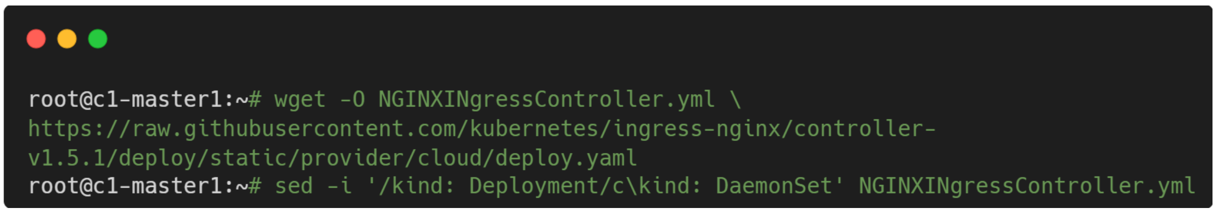


Figure 51 Visual representation of Ingress Controllers

By default, the manifest will create a Deployment with a single ingress controller replica. Since our initial cluster goal was to create a highly available environment, we want a controller present on every node. This can be achieved by simply swapping Deployment for a DaemonSet:



```
root@c1-master1:~# wget -O NGINXINgressController.yml \
https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-
v1.5.1/deploy/static/provider/cloud/deploy.yaml
root@c1-master1:~# sed -i '/kind: Deployment/c\kind: DaemonSet' NGINXINgressController.yml
```

Figure 52 Changing the NGINX Ingress Controller to work as a DaemonSet

The default deployment will also create a Service of type *LoadBalancer*. Such services depend on an external provider and are generally only available in cloud environments. We will expose

[27] https://kubernetes.github.io/ingress-nginx/

the controllers using a Service of *NodePort* type. This can be done by editing the manifest file in the following way:

```yaml
apiVersion: v1
kind: Service
metadata:
  [...]
  name: ingress-nginx-controller
  namespace: ingress-nginx
spec:
  [...]
  ports:
  - appProtocol: http
    name: http
    port: 80
    protocol: TCP
    targetPort: http
    nodePort: 30080 # add nodePort for http
  - appProtocol: https
    name: https
    port: 443
    protocol: TCP
    targetPort: https
    nodePort: 30443 # add nodePort for https
  [...]
  type: NodePort # modify from LoadBalancer
```

Snippet 15 Exposing NGINX Ingress Controller via NodePort

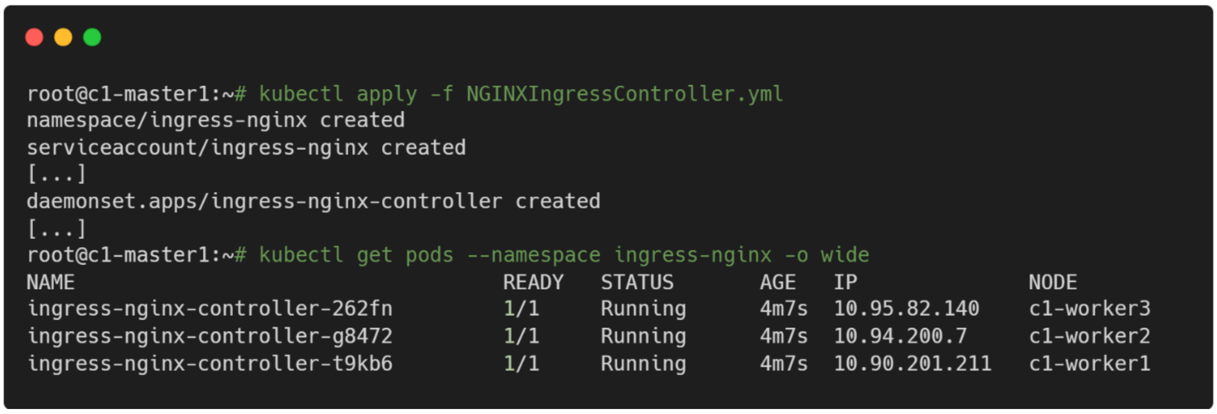
The *30080* and *30443* ports were used as, by default, only ports in the range 30000 and 32767 can be assigned to NodePort.

Since all requests to the ingress controllers will be made via the load balancer, it is necessary to configure the proxy protocol on both sides to have accurate data on the requests. The Ingress Controller configuration is stored in a *ConfigMap* resource called *ingress-nginx-controller.* Therefore the manifest file can be edited in the following way to enable full support of proxy protocol:

```yaml
apiVersion: v1
data:
  allow-snippet-annotations: "true"
  compute-full-forwarded-for: "true" # add this line
  use-forwarded-headers: "true" # add this line
  use-proxy-protocol: "true" # add this line
kind: ConfigMap
metadata:
  [...]
  name: ingress-nginx-controller
  namespace: ingress-nginx
```

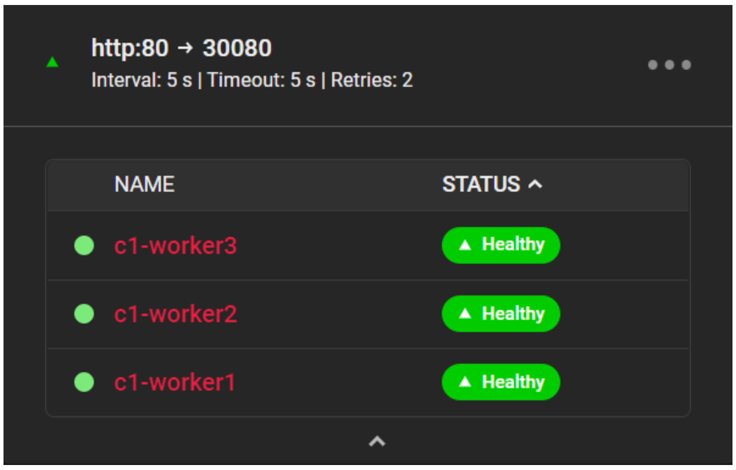
Snippet 16 Enabling Proxy Protocol support for NGINX Ingress Controller

We can now apply the manifest. After a few minutes, the pods should become *Running,* and the load balancer should become healthy.



```
root@c1-master1:~# kubectl apply -f NGINXIngressController.yml
namespace/ingress-nginx created
serviceaccount/ingress-nginx created
[...]
daemonset.apps/ingress-nginx-controller created
[...]
root@c1-master1:~# kubectl get pods --namespace ingress-nginx -o wide
NAME                               READY   STATUS     AGE   IP             NODE
ingress-nginx-controller-262fn     1/1     Running    4m7s  10.95.82.140   c1-worker3
ingress-nginx-controller-g8472     1/1     Running    4m7s  10.94.200.7    c1-worker2
ingress-nginx-controller-t9kb6     1/1     Running    4m7s  10.90.201.211  c1-worker1
```

Figure 53 Verifying the status of NGINX Ingress Controller



http:80 → 30080
Interval: 5 s | Timeout: 5 s | Retries: 2

NAME	STATUS
c1-worker3	Healthy
c1-worker2	Healthy
c1-worker1	Healthy

Figure 54 Veryfing the state of Hetzner Load Balancer

It should be noted that the community NGINX Ingress Controller should not be confused with F5's NGINX Ingress Controller. Care should be taken when reading the documentation to ensure the correct one is read.

It should also be noted that the included Terraform scripts will only create an HTTP load balancer. To create an HTTPS load balancer, a Service of TLS Termination type must be created, and a certificate uploaded via Hetzner's GUI.

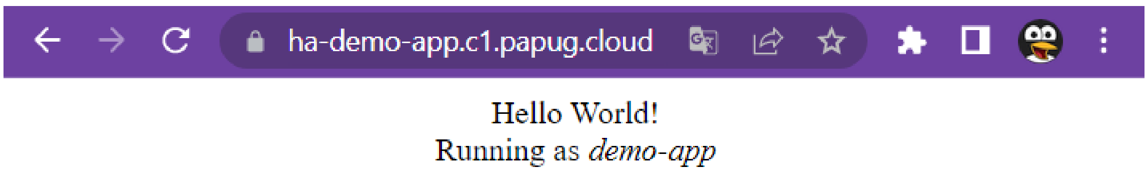
3.8.2. Ingress Resource

The following manifest can be used to expose our demo application:

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ha-demo-app
  annotations:
    kubernetes.io/ingress.class: "nginx"
spec:
  rules:
  - host: ha-demo-app.c1.papug.cloud
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: ha-demo-app
            port:
              number: 80
```

Snippet 17 Ingress manifest for ha-demo-app

After applying the manifest, we should be able to access the application from outside of our cluster, provided the DNS entries are correct.



```
Hello World!
Running as demo-app
```

Figure 55 Accessing ha-demo-app from a web browser via load balancer address

4. Example scenario

This chapter aims to propose a fault-tolerant data processing pipeline. Such a pipeline shall ensure that data can still be processed despite failures or errors. One example of its usage could be in a medical setting where critical data is received from various medical equipment.

4.1. Data processing pipeline

The proposed data processing pipeline consists of external devices, a data ingestion API, a message broker, a data analyzer, a database, and a user interface.

The external devices will be emulated for demonstration purposes by a data generator application. The application will emulate one hundred devices at the same time. Each "device" will establish its TCP connection with the data ingestion API and send generated data every ten seconds. When the connection is dropped, the application will try to reestablish the connection and resend the data. The time between attempts to send the data will grow linearly to a maximum of 10 seconds. A maximum of ten retries will be attempted.

The data ingestion API will provide the external devices with a secure interface (HTTPS) to send the data and then forward the received data to RabbitMQ's queue. Once the data is inserted into the RabbitMQ's queue, it will be considered as received, and a successful response will be sent back to the device.

The data ingestion API would be an excellent place to implement device authorization functionality if needed. Such functionality could be used, for example, for licensing purposes.

The purpose of the message broking system (implemented in RabbitMQ) is to ensure that every message received will be consumed (i.e., analyzed and saved by the data analyzer). Without a message broker, the received data could be lost during an application crash or a node failure. Moreover, the queue size could also give us valuable insight into the performance of our pipeline.

The data analyzing application will save the received data into a database and analyze the last two hundred entries to determine if the few recent entries are anomalies. If an anomaly is detected, it will be saved into a separate table in the database.

CockroachDB will be used as a database. It is a modern distributed SQL database designed to be a highly scalable and strongly-consistent key-value store. It is also highly resilient to disk and machine failures.

The anomaly API will provide a way to read the recorded anomalies. It will also provide a website with a user interface for demonstration purposes.

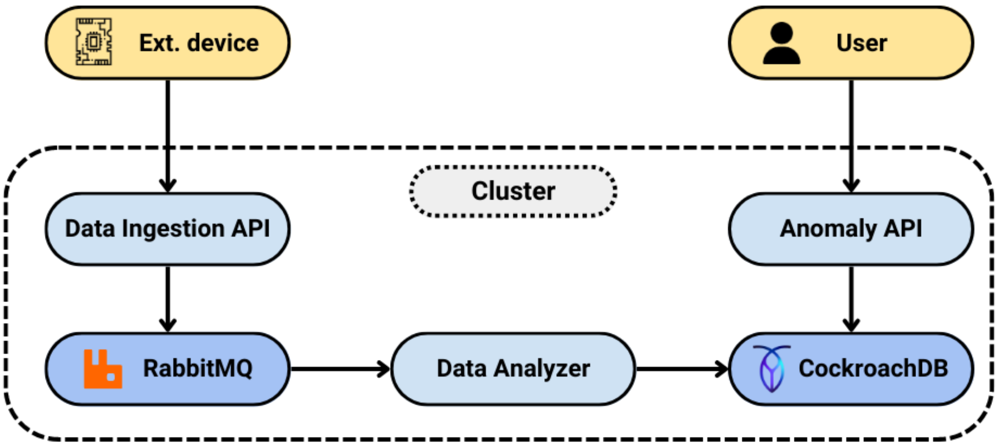


Figure 56 Visualization of the data processing pipeline

4.2. High Availability and Fault Tolerance

To achieve high availability of our applications, we will run them in two replicas. The replicas will be distributed across different nodes in the cluster. The second replica will immediately take over if the application crashes or a node fails. The Kubernetes cluster will also make sure to recreate the failed replica on a different node.

Since both RabbitMQ and CockroachDB are stateful applications, we will be using their built-in clustering functionality. Both applications provide Kubernetes operators to manage the clusters.

We will also use kube-state-metrics[28] together with Prometheus[29] and Grafana[30] to monitor the resource usage of the applications. Since the monitoring is not a critical part of the data processing pipeline, each application will run with a single replica.

[28] https://github.com/kubernetes/kube-state-metrics
[29] https://prometheus.io/
[30] https://grafana.com/

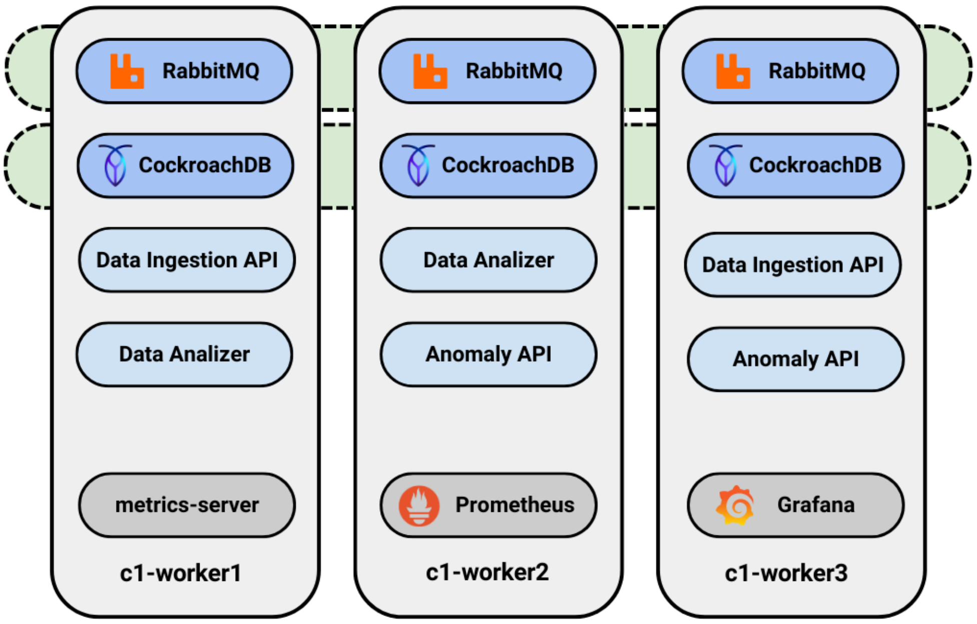


Figure 57 Cluster view of the data processing pipeline

4.3. Pod failure

4.3.1. Data Ingestion API failure

The Data Ingestion API is the entry point for our data processing API. Therefore making sure it is always available is crucial. When a pod crashes, it will automatically be removed from Service's endpoint, and no further data will be routed to it. To test the time it takes Kubernetes to remove a pod from service, a */kill* endpoint was added to the Data Ingestion API. Once a request is sent to that endpoint, the application will exit without sending a response. From Kubernetes' view, that will be considered an application crash. The request will be sent by another pod over cluster IP so that the NGINX Ingress Controller will not reattempt the request. To monitor the changes in real-time, console logging with the number of attempted retries has been added to the data generator.

During testing, it was not possible to observe more than two retries. Each retry is postponed by a second longer than the previous, meaning that the failover procedure took no more than four seconds.

4.3.2. RabbitMQ failure

RabbitMQ offers highly available clustering, though the default RabbitMQ's queues reside only on one node. In a highly available environment, it is crucial to use a queue that is being replicated across all nodes in the cluster. The *quorum queue*[31] offers such functionality. With the quorum queue, any pod (including the one that is internally classified as queue leader) can be stopped, and the cluster will continue to work without dropping a single message or request.

To test the queue's high availability, the container should be forcefully stopped on a node rather than via the *kubectl delete pod command*. This is because the kubectl delete pod command will issue a SIGTERM signal to gracefully shut down the process before force-killing the container. The container can be forcefully stopped by issuing a SIGKILL signal manually (*kill -9 <process id>*) to the container's process from the node, or one can use *crictl* to do the same (*crictl stop --timeout 0 <container id>*).

4.3.3. Data Analyzer failure

Since all messages are stored in RabbitMQ's queue until their consumption is acknowledged, the Data Analyzer can be stopped anytime. If a message is delivered, but its consumption is not acknowledged (i.e., the application crashed), RabbitMQ will attempt to redeliver the message to the next consumer (i.e., another data analyzer replica).

4.3.4. CockroachDB failure

Similarly to RabbitMQ, CockroachDB uses internal clustering. A majority of replicas have to acknowledge every data modification within the database. This ensures that no data can be lost in a single replica failure. However, it is possible to lose a connection during an operation. Therefore, if no message broking system is implemented, the client application should be capable of reconnecting to CockroachDB.

4.3.5. Alerts API failure

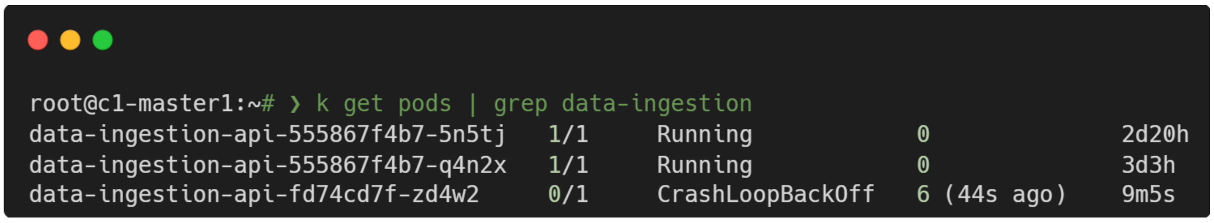
The Alerts API is an entirely stateless application used for read-only access to the database. Therefore no special measurements have to be taken.

[31] https://www.rabbitmq.com/quorum-queues.html

4.3.6. Update failure

To protect against faulty deployments, a *livenessProbe*[32] should be used. It gives Kubernetes additional insight into whether the application is behaving as expected. The liveness probe can be implemented in three ways: command execution within the pod; http request; establishment of tcp connection.

For demonstration purposes, the Data Ingestion API was extended with livenessProbe and then a faulty version was deployed. The rolling update process discussed in chapter 3.5 makes sure that the previous deployment stays running.



```
root@c1-master1:~# ❯ k get pods | grep data-ingestion
data-ingestion-api-555867f4b7-5n5tj   1/1     Running          0             2d20h
data-ingestion-api-555867f4b7-q4n2x   1/1     Running          0             3d3h
data-ingestion-api-fd74cd7f-zd4w2     0/1     CrashLoopBackOff 6 (44s ago)   9m5s
```

Figure 58 Failed deployment of Data Ingestion API

4.4. Node failure

After a node failure it will take about one minute for Kubernetes to mark a node as *NotReady*. After the default eviction timeout[33], that is five minutes, pods running on that node will be rescheduled on different nodes. During this time it is possible that some requests will be forwarded to the failed node. Such requests should be repeated until they hit a healthy node.

Note: If you plan to bring multiple nodes down at a time for maintenance, consider implementing PodDistruptionBudget[34] to gracefully drain the nodes without impacting the applications.

[32] https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/
[33] https://kubernetes.io/docs/concepts/architecture/nodes/#condition
[34] https://kubernetes.io/docs/tasks/run-application/configure-pdb/

5. Monitoring

When specifying a Pod manifest, it is possible to specify CPU/RAM *requests* and *limits*. Requested resources are guaranteed, whereas limits define the maximum amount of resources a pod can consume. For example, if a pod is defined with a request of 128MB of memory, the Kubernetes scheduler will make sure that the server on which a Pod will be scheduled will have at least 128MB of memory available for that Pod. The Pod may consume more than 128MB of memory unless a limit is also defined. When a Pod tries more than the allowed amount of memory, the process will be terminated with an *OutOfMemory* error.

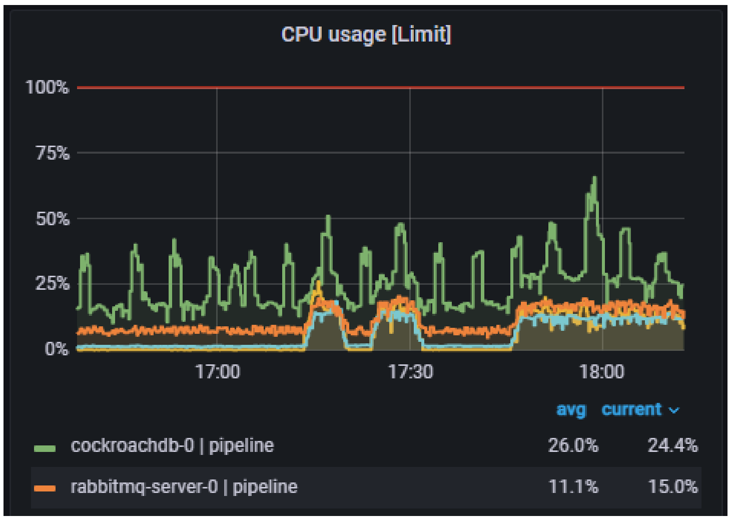
Every kubelet exposes metrics about containers running on a node via built-in cAdvisor[35]. The metrics can be accessed via Kubernetes API, and Prometheus can periodically scan them. Since every request to the Kubernetes API has to be authenticated and authorized, it is recommended that a ServiceAccount with adequate permissions is created. Example manifest and instructions can be found in the *metrics-components* repository in Appendix 1.

To further extend monitoring capabilities, kube-state-metrics can provide Prometheus metrics with additional information on the state of the entire Kubernetes cluster (e.g., number of Deployments, number of healthy/unhealthy Pods).

[35] https://github.com/google/cadvisor

5.1. CPU consumption

To measure CPU consumption over time, two metrics have to be used, that is *container_cpu_usage_seconds_total* and *container_spec_cpu_period.* Together with *container_spec_cpu_quota* a graph of current CPU usage vs. CPU limit can be plotted.

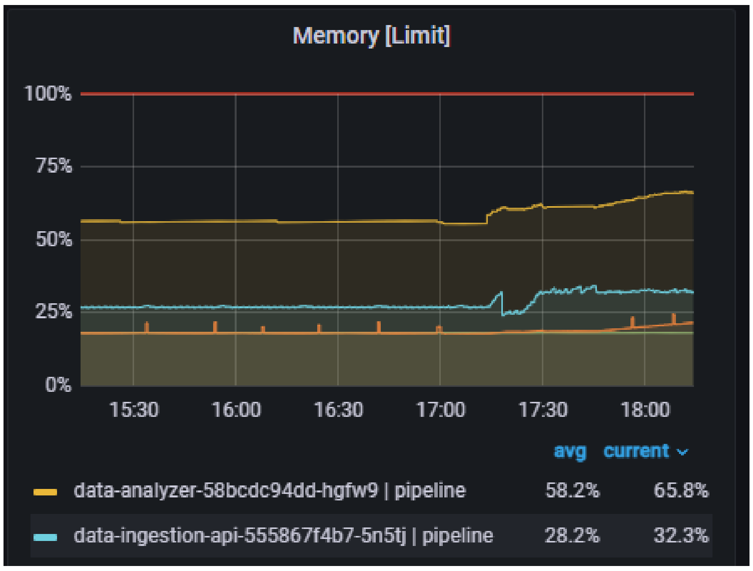


		avg	current
🟩	cockroachdb-0 | pipeline	26.0%	24.4%
🟧	rabbitmq-server-0 | pipeline	11.1%	15.0%

Figure 59 CPU consumption plotted in Grafana

5.2. RAM usage

The current RAM usage is given via *container_memory_usage_bytes* metric. It can be combined with *container_spec_memory_limit_bytes* to plot current RAM usage vs. RAM usage.

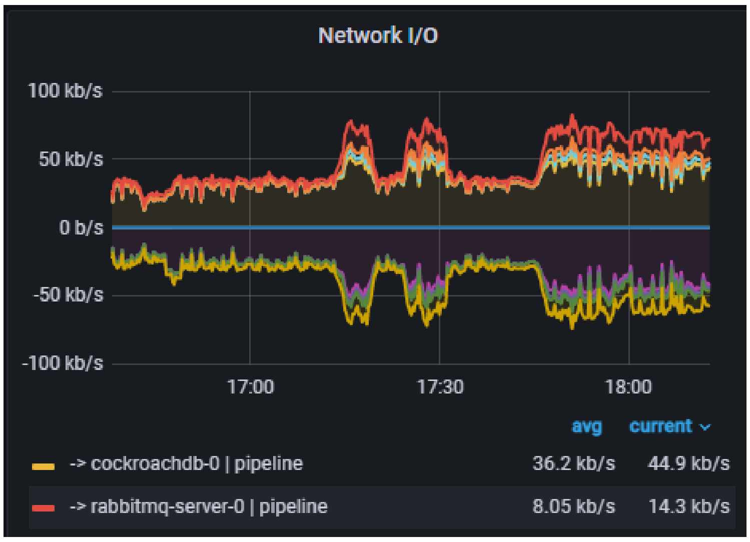


		avg	current
🟨	data-analyzer-58bcdc94dd-hgfw9 | pipeline	58.2%	65.8%
🟦	data-ingestion-api-555867f4b7-5n5tj | pipeline	28.2%	32.3%

Figure 60 RAM usage plotted in Grafana

5.3. Network I/O

The container's network metrics are separated into receive (*container_network_receive_bytes_total)* *and* transmit (*container_network_transmit_bytes_total*) metrics. When combined, they can be used to plot the total network traffic.



	avg	current
-> cockroachdb-0 | pipeline	36.2 kb/s	44.9 kb/s
-> rabbitmq-server-0 | pipeline	8.05 kb/s	14.3 kb/s

Figure 61 Network I/O plotted in Grafana

A Grafana's dashboard JSON definition of the above graphs can be found in the *metrics-components* repository in Appendix 1.

6. Scalability

All components of the proposed data processing pipeline except for RabbitMQ and CockroachDB can be classified as stateless and, therefore, can be scaled horizontally nearly infinitely.

CockroachDB has been built from the ground up to allow for scalability, and clusters can be scaled horizontally by manipulating the number of nodes.

The proposed RabbitMQ scheme emphasizes consistency over scalability. To improve RabbitMQ's scalability, federation[36] can be implemented.

The applications have been written in a single-threaded manner. Therefore each replica has limited throughput. The author suggests coding applications multi-threadedly whenever high throughput is necessary.

Kubernetes clusters can also be scaled to enormous sizes. According to the Kubernetes documentation[37]:

> *Kubernetes is designed to accommodate configurations that meet all of the following criteria:*
>
> *- No more than 110 pods per node*
>
> *- No more than 5,000 nodes*
>
> *- No more than 150,000 total pods*
>
> *- No more than 300,000 total containers*

It is worth noting that the author has successfully run a cluster with nearly three hundred pods on a single node in a production environment with some additional commitment (notably, all the pods had defined resource requests and limits).

[36] https://www.rabbitmq.com/distributed.html
[37] https://kubernetes.io/docs/setup/best-practices/cluster-large/

7. Summary

The thesis is a comprehensive guide to implementing highly scalable and highly available environments for applications based on Kubernetes. A production-grade cluster was described and implemented with a focus on high reliability for mission-critical applications. The project took into account the challenges of operating such a large-scale cluster.

The Kubernetes cluster was extended with dynamic volume provisioning, an ingress controller, and Calico-based networking.

A fault-tolerant data processing pipeline with simulated IoT device pools was implemented to demonstrate the effectiveness of the proposed solution. The scenario showcased the ability of the Kubernetes cluster to recover from failures and maintain its availability even in the event of node failures or faulty application updates. Recommendations were also offered on monitoring and scaling the application to ensure continued efficient operation as the cluster grows.

The data processing pipeline was implemented with multiple components written in Deno. RabbitMQ was used as a message broking system, and CockroachDB was used to store the processed data.

The monitoring system was implemented with the help of cAdvisor and kube-state-metrics in combination with Prometheus and Grafana to visualize the data over time.

Overall, the work provides a comprehensive solution for IT students and professionals seeking to implement a robust and scalable Kubernetes-based environment for their containerized applications. The implementation, simulated scenario, monitoring, and scaling solutions, provide valuable insights and guidance for software engineers and architects embarking on similar projects.

Acknowledgments

Some parts of this document have been written with the help of OpenAI's ChatGPT3[38]. The author is aware of the shortcomings of generative pre-trained language models and has never considered the AI assistant a formal source of information. Instead, it was primarily used for preliminary copywriting drafts of individual paragraphs and proofreading. All the information generated by ChatGPT has been curated by the author and is backed by external sources or years of professional experience.

The graphs have been made using Canva and were inspired by official Kubernetes documentation. Code snippets were created with the help of the Carbon App[39].

Kubernetes icons (characterized by hexagonal shape and blue background) were designed by Kubernetes community members[40] and used under the Creative Commons Attribution 4.0 International license.

The "user" and "circuit" icons used in Figure 56 were made by Freepik's team from www.flaticon.com.

[38] https://chat.openai.com/
[39] https://carbon.now.sh/
[40] https://github.com/kubernetes/community/tree/master/icons

Glossary

Term	Definition
Node	A single instance of a physical or virtual machine that provides computing resources
Host	In the context of containerization, a physical or virtual machine on which containers are run. It provides the underlying operating system, hardware, and resources.
Container	A container is a lightweight and standalone execution environment for an application, including everything it needs to run, that can be easily moved between environments.
Container Image	A container image is a packaged version of an application and its dependencies and configuration that can be run in a container.
Cloud provider	A company that offers cloud computing services, allowing users to access computing resources over the internet on a pay-as-you-go basis.
FQDN	A fully qualified domain name (FQDN) is a complete and unique domain name that specifies the exact location of a network resource. It typically consists of a combination of the hostname, domain name, and top-level domain, such as "server.domain.tld"
Terraform	A tool for building, changing, and versioning infrastructure safely and efficiently. It uses configuration files to describe the desired state of the infrastructure and can manage infrastructure resources across multiple cloud providers and on-premises environments.
Package repository	A Linux package repository is a directory that contains packaged software for Linux systems, which can be downloaded and installed using a package manager such as apt or yum.
Pod	A pod is the smallest deployable unit in the Kubernetes object model. It represents a single instance of a containerized application.
Controller	In Kubernetes, controllers are used to manage the system's desired state. They continuously monitor the current state and apply changes to align it with the desired state.

References

About the Open Container Initiative. (n.d.). Retrieved November 11, 2022, from https://opencontainers.org/about/overview/

Comarch. (n.d.). *Opaska Życia: The Band of Life*. Opaska Życia. Retrieved November 7, 2022, from https://www.comarch.pl/healthcare/produkty/teleopieka/opaska-zycia/

Diego Ongaro & John Ousterhout. (2014). In search of an understandable consensus algorithm. *USENIX Annual Technical Conference*, 305–320.

docker build. (n.d.). Docker Documentation. Retrieved January 30, 2023, from https://docs.docker.com/engine/reference/commandline/build/

Docker Engine release notes. (2022, November 10). Docker Documentation. https://docs.docker.com/engine/release-notes/prior-releases/

Dua, R., Raja, A. R., & Kakadia, D. (2014). Virtualization vs Containerization to Support PaaS. *2014 IEEE International Conference on Cloud Engineering*. https://doi.org/10.1109/ic2e.2014.41

Erl, T. (2005). *Service-oriented Architecture: Concepts, Technology, and Design*. Prentice Hall.

Gillis, A. S. (2022, March 4). *What is the internet of things (IoT)?* IoT Agenda. https://www.techtarget.com/iotagenda/definition/Internet-of-Things-IoT

M, A., Dinkar, A., Mouli, S. C., B, S., & Deshpande, A. A. (2021). Comparison of Containerization and Virtualization in Cloud Architectures. *2021 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*. https://doi.org/10.1109/conecct52877.2021.9622668

Mehndiratta, H. (2021, April 2). Comparing Kubernetes Container Network Interface (CNI) providers. *Kubevious*. https://kubevious.io/blog/post/comparing-kubernetes-container-network-interface-cni-providers

Schmeling, B., & Dargatz, M. (2022). The Impact of Kubernetes on Development. *Kubernetes Native Development*, 1–57. https://doi.org/10.1007/978-1-4842-7942-7_1

Wang, Z., Guo, C., Fu, Z., & Yang, S. (2020). Identifying the Development Trend of ARM-based Server Ecosystem Using Linux Kernels. *2020 IEEE International Conference on Progress in Informatics and Computing (PIC)*. https://doi.org/10.1109/pic50277.2020.9350743

What is the Principle of Least Privilege (POLP). (n.d.). OneLogin. Retrieved November 29, 2022, from https://www.onelogin.com/learn/least-privilege-polp

Table of figures

Table of tables

Table of snippets

Appendix 1 - Software used

- Docker Desktop (20.10.21):
  - Available from: https://docs.docker.com/desktop/install/windows-install/
- Docker Compose (v2.12.2):
  - Included with Docker Desktop
- Deno (1.28.0):
  - Available from: https://deno.land/manual@v1.28.0/getting_started/installation
- Kubernetes (1.25.5):
  - Setup discussed in chapter 3.2
- Python (1.28.0):
  - Available from: https://www.python.org/downloads/
  - No additional libraries were used
- ha-demo-app (version adequate to chapter number):
  - Available from: https://gitlab.com/ha-thesis/ha-demo-app
  - multiple versions available as git branches
- elasticsearch (8.5.0):
  - Available from: https://www.elastic.co/guide/en/elasticsearch/reference/current/docker.html
- kibana (8.5.0):
  - Available from: https://www.elastic.co/guide/en/kibana/current/docker.html
- Data Generator (latest):
  - Available from: https://gitlab.com/ha-thesis/data-generator
- Data Ingestion API (latest):
  - Available from: https://gitlab.com/ha-thesis/data-ingestion-api
- Data Analyzer (latest):
  - Available from: https://gitlab.com/ha-thesis/data-analyzer
- Alerts API (latest):
  - Available from: https://gitlab.com/ha-thesis/alerts-api
- Alerts GUI (latest):
  - Available from: https://gitlab.com/ha-thesis/alerts
- CPU Stresser (latest):
  - Available from: https://gitlab.com/ha-thesis/cpu-stresser
- RAM Hogger (latest):
  - Available from: https://gitlab.com/ha-thesis/ram-hogger

- Prometheus (v2.40.7):
  - Available from: https://prometheus.io/download/
  - Deployment manifest available from: https://gitlab.com/ha-thesis/metrics-components
- Grafana (9.3.2):
  - Available from: https://grafana.com/grafana/download
  - Deployment manifest available from: https://gitlab.com/ha-thesis/metrics-components
- kube-state-metrics (v2.7.0):
  - Available from: https://github.com/kubernetes/kube-state-metrics
  - Deployment manifest available from: https://gitlab.com/ha-thesis/metrics-components
- RabbitMQ Cluster Operator (2.1.0;):
  - Available from: https://www.rabbitmq.com/kubernetes/operator/operator-overview.html
  - Deployment manifest available from: https://gitlab.com/ha-thesis/rabbitmq
- CockroachDB Cluster Operator (2.1.0;):
  - Available from: https://www.cockroachlabs.com/docs/stable/deploy-cockroachdb-with-kubernetes.html
  - Deployment manifest available from: https://gitlab.com/ha-thesis/rabbitmq