

How to handle Docker data?

Introduction

For our project we need to run multiple dockers that must import data as well as export data. Because the amount of data can become big quickly, we're going to explore multiple ways of doing this in this research and decide the way to go for our project.

Research questions

What is the most efficient and scalable approach for importing and exporting large amounts of data in and out of a Docker container?

1. What are the available approaches for importing and exporting data within and outside of a Docker container?
2. How can we measure the efficiency and scalability of the identified approaches?
3. What are the benefits and drawbacks of each approach in terms of efficiency and scalability?
4. What are the security implications of each approach?

What are the available approaches for importing and exporting data within and outside of a Docker container?

Dot methods

- Literature study
- Expert interview

1. **Docker volumes** are a mechanism that enables storing and managing data in a separate volume outside of a container. Volumes provide a persistent data storage solution for containers, which can be shared among multiple containers, allowing data to persist even when the container is deleted or recreated. To use Docker volumes, you can create a new volume and specify it when creating or running a container, which will mount the volume to the container's file system. You can also manage Docker volumes using the Docker CLI, allowing you to list, inspect, create, and remove volumes. Additionally, Docker volumes offer several benefits, such as improved performance, better data management, and more manageable backups and restores. To import data into a container, you can create a new volume and copy the data into it. To export data from a container, you can copy the data to a volume and then access it from the host machine. Overall, Docker volumes are a powerful feature that provide a flexible and efficient solution for data management in Docker containers.
2. **Docker bind mounts** are a mechanism that enables mounting a file or directory on the host machine into a container. Bind mounts provide a way to share files between the host and the container, allowing for easy access to files or directories that reside on the host machine. To use a bind mount, you can specify the source directory or file and the destination directory in the container when creating or running a container. Once the bind mount is set up, any changes made to the source directory or file on the host machine will also be reflected in the container, and vice versa. Docker bind mounts also offer several benefits, such as easy access to files, improved performance, and the ability to modify files outside of the container. However, bind mounts are not ideal for sharing data between containers or for persistent storage, as any changes made to the host machine could potentially break the container. To import data into a container, you can mount the directory containing the data into the container. To export data from a container, you can mount a directory from the container onto the host machine. Overall, Docker bind mounts are a useful feature for sharing files between the host and the container, but should be used with caution in production environments.
3. **The Docker copy command** allows you to copy files between a container and the host machine. To import data into a container, you can use the Docker copy command to copy the data from the host machine to the container. To export data from a container, you can use the Docker copy command to copy the data from the container to the host machine.

4. **The Docker export command** is a Docker CLI command that enables exporting a container's file system as a tar archive. This command can be used to save the state of a container's file system, including any changes made to the container's file system since it was created, as a single file that can be transferred and imported to other machines or environments. To use the Docker export command, you can specify the name or ID of the container and the path and filename of the output tar file. Once the tar file is created, it can be transferred to another machine or environment and imported using the Docker import command. However, it's important to note that the Docker export command does not include any metadata about the container, such as its networking or storage configuration, so it may not be suitable for all use cases. Additionally, exporting and importing containers can result in large tar files, which can be time-consuming to transfer and may require significant storage space. Overall, the Docker export command is a useful feature for exporting a container's file system as a tar archive, but should be used with caution and with a clear understanding of its limitations. To export data from a container, you can use the Docker export command to create a tar file containing the data. To import data into a container, you can use the Docker import command to create a new container from the tar file.

Expert interview

For the expert interview I talked to Henners from the Docker discord. We talked to him about the research we were doing for and let him read the sub question. He said that the 4 approaches right now are a good start but there are so many more possibilities. To start with here is the list of 4 more approaches/possibilities.

1. Containers are able to use HTTPS
2. Containers can use databases and object store
3. NFS
4. Multi stage building

I think these are great options to look into and we will for sure do this within the group project. For now a big thank you to Henners for helping us out.

How can we measure the efficiency and scalability of the identified approaches?

To measure the efficiency and scalability of the identified approaches we need to determine what efficient and scalable means, given our context? We sat down with our Product Owner (PO) to discuss the requirements for the platform they can be found here: <https://github.com/S-A-RB05/.github/blob/main/User%20stories.pdf>

Dot methods

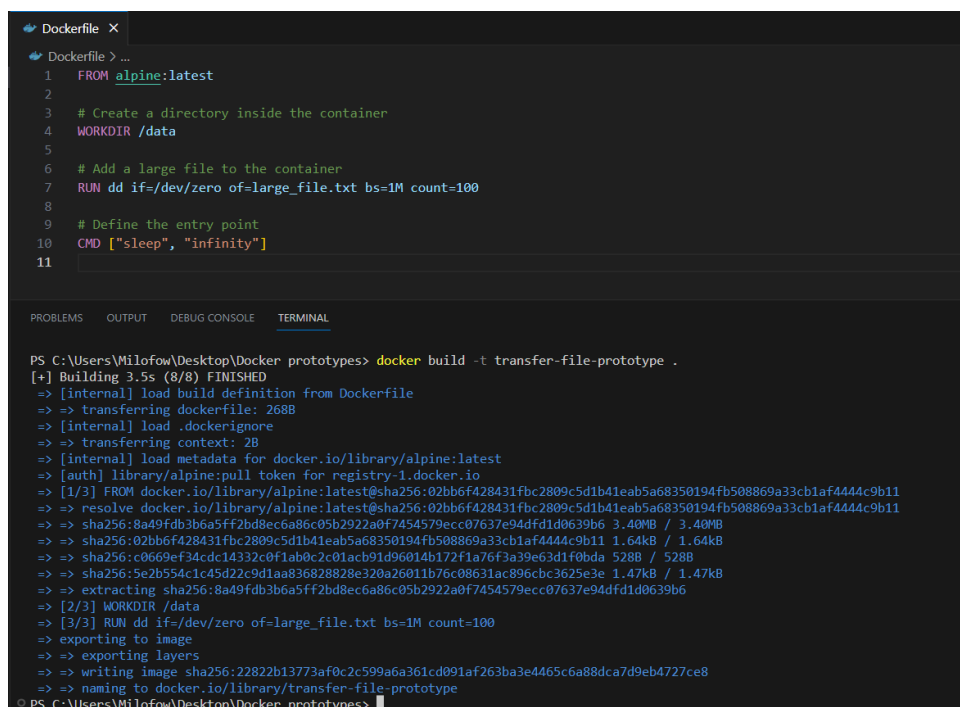
- Explore user requirements
- Prototyping (<https://github.com/S-A-RB05/Research-docker-prototypes>)
- Benchmark test

Efficiency

For our project the speed of importing and exporting is important, because this will be one of the main tasks when testing trading strategies on the platform and if the speed is low the latency will build up.

Besides being fast it should also be reliable meaning it should have a good percentage of succeeding and therefore should be a good practice rather than a special exceptional way of doing it.

To measure these two I will be building a test environment and will be timing how long a file transfer takes, I created a prototype docker image to work with. To test the reliability I tested it multiple times to ensure a constant result.

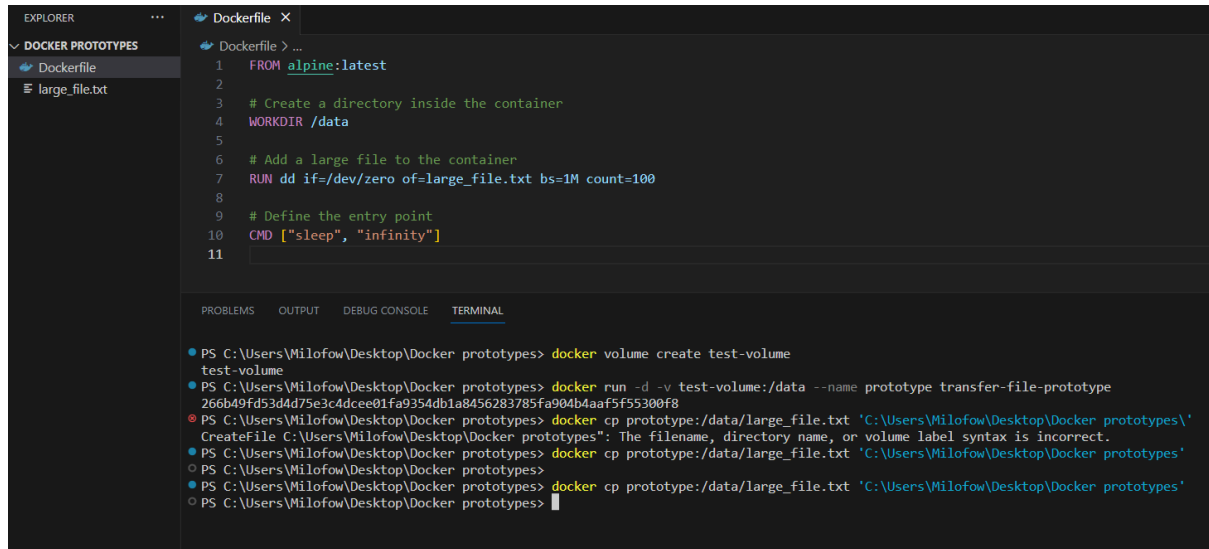


```
Dockerfile X
Dockerfile > ...
1 FROM alpine:latest
2
3 # Create a directory inside the container
4 WORKDIR /data
5
6 # Add a large file to the container
7 RUN dd if=/dev/zero of=large_file.txt bs=1M count=100
8
9 # Define the entry point
10 CMD ["sleep", "infinity"]
11

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

PS C:\Users\Milofow\Desktop\Docker prototypes> docker build -t transfer-file-prototype .
[+] Building 3.5s (8/8) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 268B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/alpine:latest
=> [auth] library/alpine:pull token for registry-1.docker.io
=> [1/3] FROM docker.io/library/alpine:latest@sha256:02bb6f428431fbc2809c5d1b41eab5a68350194fb508869a33cb1af4444c9b11
=> => resolve docker.io/library/alpine:latest@sha256:02bb6f428431fbc2809c5d1b41eab5a68350194fb508869a33cb1af4444c9b11
=> => sha256:8a49fdb3b6a5ff2bd8ec6a86c05b2922a0f7454579ecc07637e94df1d0639b6 3.40MB / 3.40MB
=> => sha256:02bb6f428431fbc2809c5d1b41eab5a68350194fb508869a33cb1af4444c9b11 1.64kB / 1.64kB
=> => sha256:c0669ef34cdc14332c0f1ab0c2c01acb91d96014b172f1a76f3a39e63d1f0bda 528B / 528B
=> => sha256:5e2b554c1c4d5d22c9d1aa836828828e320a26011b76c08631ac896cbc3625e3e 1.47kB / 1.47kB
=> => extracting sha256:8a49fdb3b6a5ff2bd8ec6a86c05b2922a0f7454579ecc07637e94df1d0639b6
=> [2/3] WORKDIR /data
=> [3/3] RUN dd if=/dev/zero of=large_file.txt bs=1M count=100
=> exporting to image
=> => exporting layers
=> => writing image sha256:22822b13773af0c2c599a6a361cd091af263ba3e4465c6a88dca7d9eb4727ce8
=> => naming to docker.io/library/transfer-file-prototype
PS C:\Users\Milofow\Desktop\Docker prototypes>
```

Volume



The screenshot shows a VS Code editor with a Dockerfile and a terminal window. The Dockerfile contains the following content:

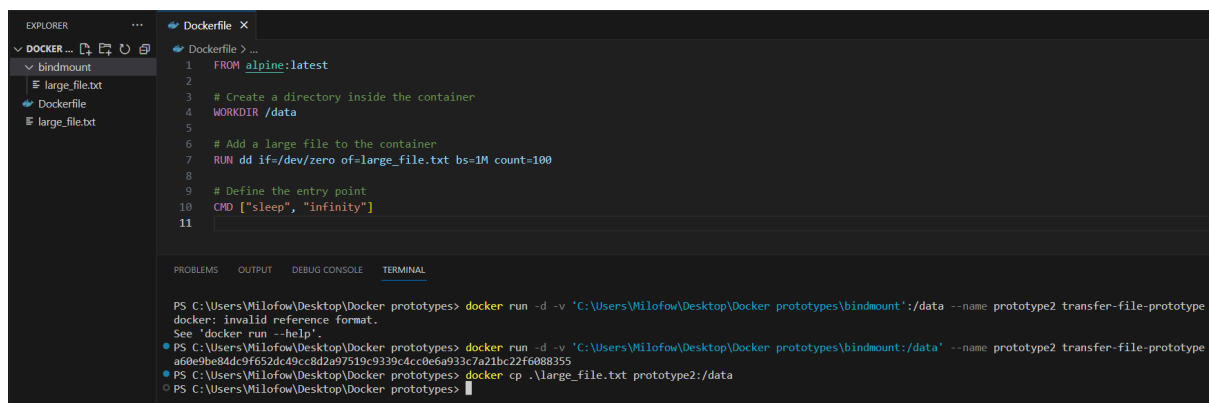
```
Dockerfile > ...
1 FROM alpine:latest
2
3 # Create a directory inside the container
4 WORKDIR /data
5
6 # Add a large file to the container
7 RUN dd if=/dev/zero of=large_file.txt bs=1M count=100
8
9 # Define the entry point
10 CMD ["sleep", "infinity"]
11
```

The terminal window shows the following commands and output:

```
PS C:\Users\Milofow\Desktop\Docker prototypes> docker volume create test-volume
test-volume
PS C:\Users\Milofow\Desktop\Docker prototypes> docker run -d -v test-volume:/data --name prototype transfer-file-prototype
266b49fd53d4d75e3c4dcee01fa9354db1a8456283785fa904b4aaf5f55300f8
PS C:\Users\Milofow\Desktop\Docker prototypes> docker cp prototype:/data/large_file.txt 'C:\Users\Milofow\Desktop\Docker prototypes\'
CreateFile C:\Users\Milofow\Desktop\Docker prototypes\: The filename, directory name, or volume label syntax is incorrect.
PS C:\Users\Milofow\Desktop\Docker prototypes> docker cp prototype:/data/large_file.txt 'C:\Users\Milofow\Desktop\Docker prototypes\'
PS C:\Users\Milofow\Desktop\Docker prototypes> docker cp prototype:/data/large_file.txt 'C:\Users\Milofow\Desktop\Docker prototypes\'
PS C:\Users\Milofow\Desktop\Docker prototypes>
```

This took 0,96 seconds and didn't fail on multiple tries.

Bind mounts



The screenshot shows a VS Code editor with a Dockerfile and a terminal window. The Dockerfile contains the following content:

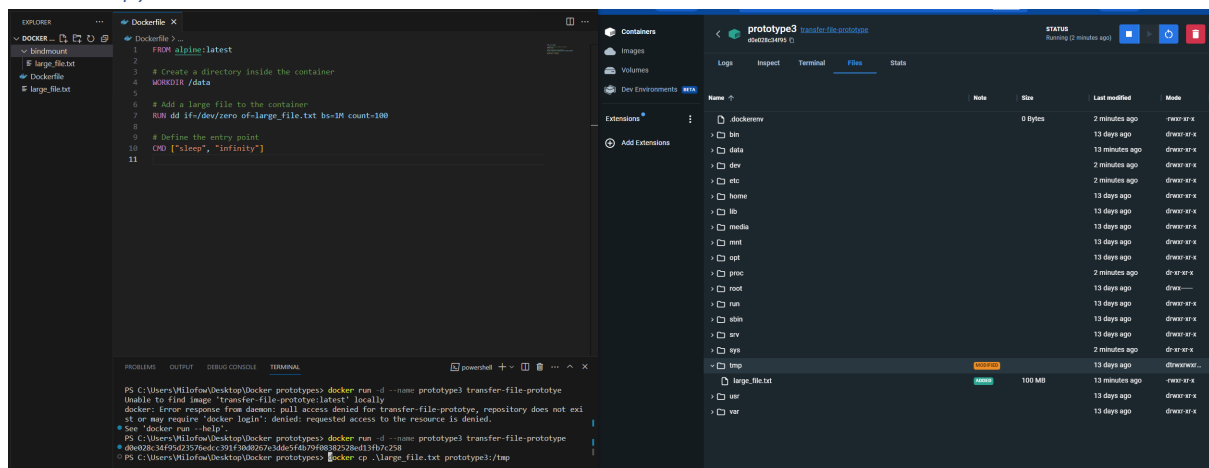
```
Dockerfile > ...
1 FROM alpine:latest
2
3 # Create a directory inside the container
4 WORKDIR /data
5
6 # Add a large file to the container
7 RUN dd if=/dev/zero of=large_file.txt bs=1M count=100
8
9 # Define the entry point
10 CMD ["sleep", "infinity"]
11
```

The terminal window shows the following commands and output:

```
PS C:\Users\Milofow\Desktop\Docker prototypes> docker run -d -v 'C:\Users\Milofow\Desktop\Docker prototypes\bindmount':/data --name prototype2 transfer-file-prototype
docker: invalid reference format.
See 'docker run --help'.
PS C:\Users\Milofow\Desktop\Docker prototypes> docker run -d -v 'C:\Users\Milofow\Desktop\Docker prototypes\bindmount:/data' --name prototype2 transfer-file-prototype
a60e9be84dc9f652dc49cc8d2a97519c9339c4cc06a933c7a21bc22f6088355
PS C:\Users\Milofow\Desktop\Docker prototypes> docker cp .\large_file.txt prototype2:/data
PS C:\Users\Milofow\Desktop\Docker prototypes>
```

This took 1,01 seconds and didn't fail on multiple tries.

Docker copy



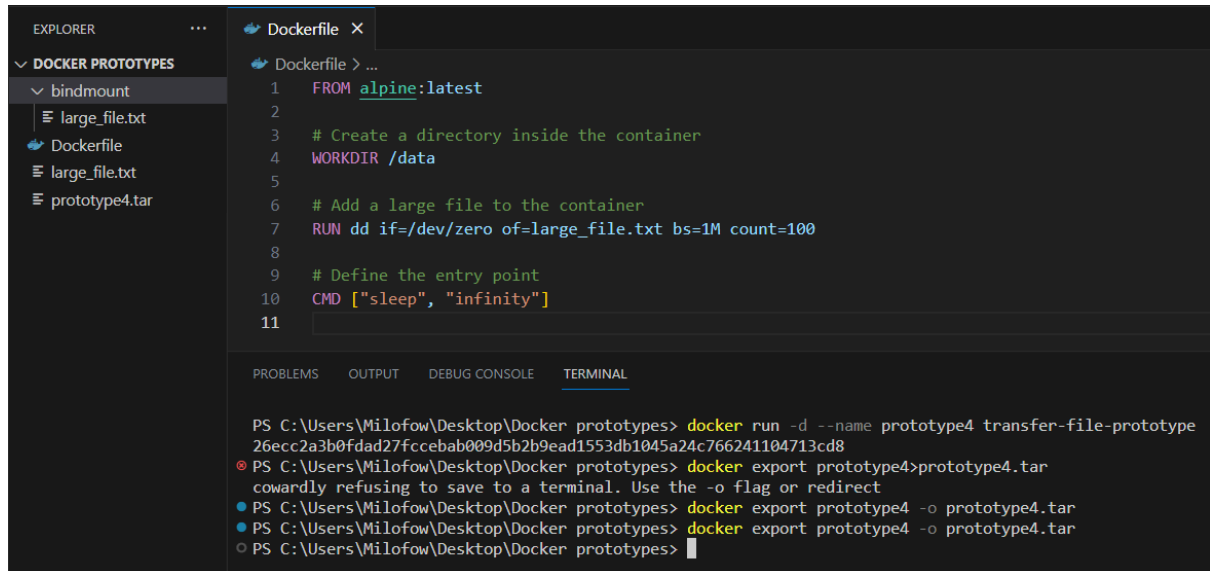
The screenshot shows a VS Code editor with a Dockerfile and a Docker Desktop interface. The Dockerfile contains the following content:

```
Dockerfile > ...
1 FROM alpine:latest
2
3 # Create a directory inside the container
4 WORKDIR /data
5
6 # Add a large file to the container
7 RUN dd if=/dev/zero of=large_file.txt bs=1M count=100
8
9 # Define the entry point
10 CMD ["sleep", "infinity"]
11
```

The Docker Desktop interface shows the 'prototype3' container. The 'Files' tab is selected, showing a list of files and directories. The 'large_file.txt' file is highlighted, showing its size as 100 MB and its last modified time as 13 days ago.

This took 1,08 seconds and didn't fail on multiple tries.

Docker export

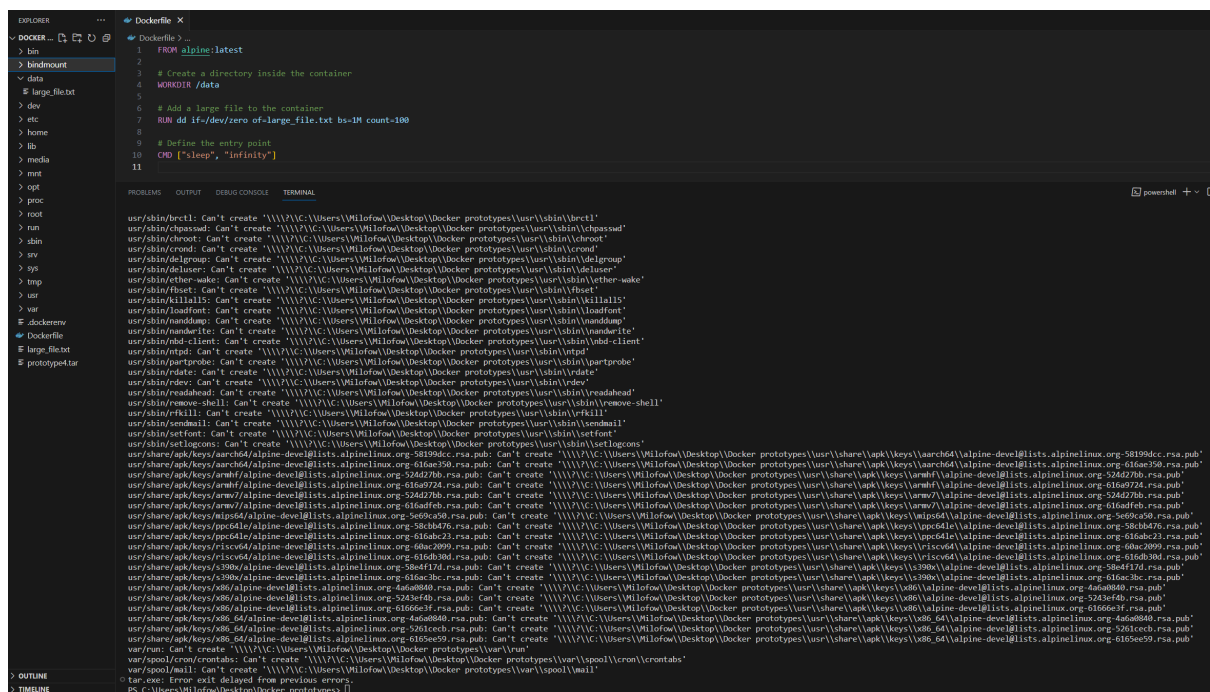


The screenshot shows a Visual Studio Code editor with a Dockerfile open. The Dockerfile contains the following content:

```
1 FROM alpine:latest
2
3 # Create a directory inside the container
4 WORKDIR /data
5
6 # Add a large file to the container
7 RUN dd if=/dev/zero of=large_file.txt bs=1M count=100
8
9 # Define the entry point
10 CMD ["sleep", "infinity"]
11
```

The terminal output shows the command `docker run -d --name prototype4 transfer-file-prototype 26ecc2a3b0fdad27f9cebab009d5b2b9ead1553db1045a24c766241104713cd8` being executed. The output indicates that the container was created successfully and is running. The command `docker export prototype4>prototype4.tar` is also shown, which exports the container to a tar file.

This took 1,23 seconds and didn't fail on multiple tries, but you will also need to extract the .tar file which takes another 2,71 seconds.

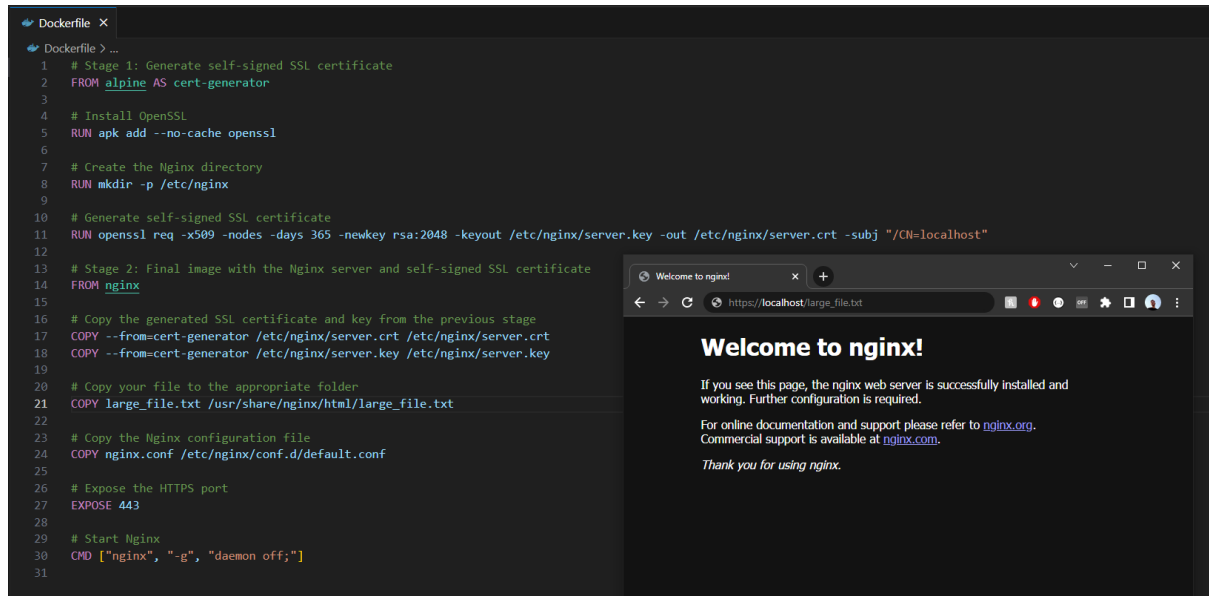


The screenshot shows a Visual Studio Code editor with a Dockerfile open. The Dockerfile contains the following content:

```
1 FROM alpine:latest
2
3 # Create a directory inside the container
4 WORKDIR /data
5
6 # Add a large file to the container
7 RUN dd if=/dev/zero of=large_file.txt bs=1M count=100
8
9 # Define the entry point
10 CMD ["sleep", "infinity"]
11
```

The terminal output shows the command `docker run -d --name prototype4 transfer-file-prototype 26ecc2a3b0fdad27f9cebab009d5b2b9ead1553db1045a24c766241104713cd8` being executed. The output indicates that the container was created successfully and is running. The command `docker export prototype4>prototype4.tar` is also shown, which exports the container to a tar file.

HTTPS



The image shows a Dockerfile on the left and a web browser on the right. The Dockerfile is a multi-stage build for an nginx web server with HTTPS. It starts with an Alpine-based stage to generate a self-signed SSL certificate and install OpenSSL. The second stage uses the nginx image and copies the certificate, key, and configuration files. The final image exposes port 443 and starts nginx. The web browser on the right shows the 'Welcome to nginx!' page, indicating the server is running successfully.

```
1 # Stage 1: Generate self-signed SSL certificate
2 FROM alpine AS cert-generator
3
4 # Install OpenSSL
5 RUN apk add --no-cache openssl
6
7 # Create the Nginx directory
8 RUN mkdir -p /etc/nginx
9
10 # Generate self-signed SSL certificate
11 RUN openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout /etc/nginx/server.key -out /etc/nginx/server.crt -subj "/CN=localhost"
12
13 # Stage 2: Final image with the Nginx server and self-signed SSL certificate
14 FROM nginx
15
16 # Copy the generated SSL certificate and key from the previous stage
17 COPY --from=cert-generator /etc/nginx/server.crt /etc/nginx/server.crt
18 COPY --from=cert-generator /etc/nginx/server.key /etc/nginx/server.key
19
20 # Copy your file to the appropriate folder
21 COPY large_file.txt /usr/share/nginx/html/large_file.txt
22
23 # Copy the Nginx configuration file
24 COPY nginx.conf /etc/nginx/conf.d/default.conf
25
26 # Expose the HTTPS port
27 EXPOSE 443
28
29 # Start Nginx
30 CMD ["nginx", "-g", "daemon off;"]
31
```

Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

This took 1,45 seconds and didn't fail on multiple tries.

Databases/NFS

This will not be possible because directly accessing the database doesn't fit into our architecture and this will also cause multiple points of failure if something goes wrong while connecting in multiple amounts of dockers

Multi stage building

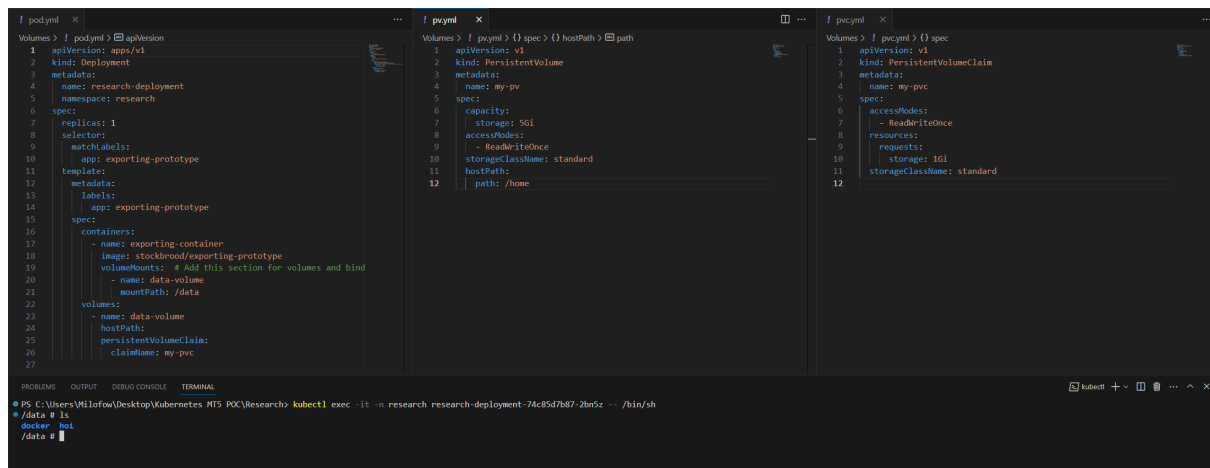
The multi stage building is not applicable in our situation because we don't build every docker to reduce time, but just grab it from docker hub.

Scalability

While working on the group project we started working with Kubernetes and have suspicions that some of the identified ways of im-/exporting data will not work inside a cluster. Therefore the criteria is that it should work in a scalable environment. To test this I will try how easily I can deploy my prototypes to a Kubernetes cluster.

Docker volumes

To achieve this I used a Persistent volume claim (PVC) inside the Kubernetes cluster. Just like volumes are managed by Docker PVC are managed by Kubernetes. I created a new directory to test the PVC with called “hoi”.



```
! pod.yml x
Volumes > / pod.yml > {} spec
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: research-deployment
5   namespace: research
6 spec:
7   replicas: 1
8   selector:
9     matchLabels:
10      app: exporting-prototype
11 template:
12   metadata:
13     labels:
14       app: exporting-prototype
15   spec:
16     containers:
17       - name: exporting-container
18         image: stockbrood/exporting-prototype
19         volumeMounts: # Add this section for volumes and bind
20           - name: data-volume
21             mountPath: /data
22     volumes:
23       - name: data-volume
24         hostPath:
25           persistentVolumeClaim:
26             claimName: my-pvc
27

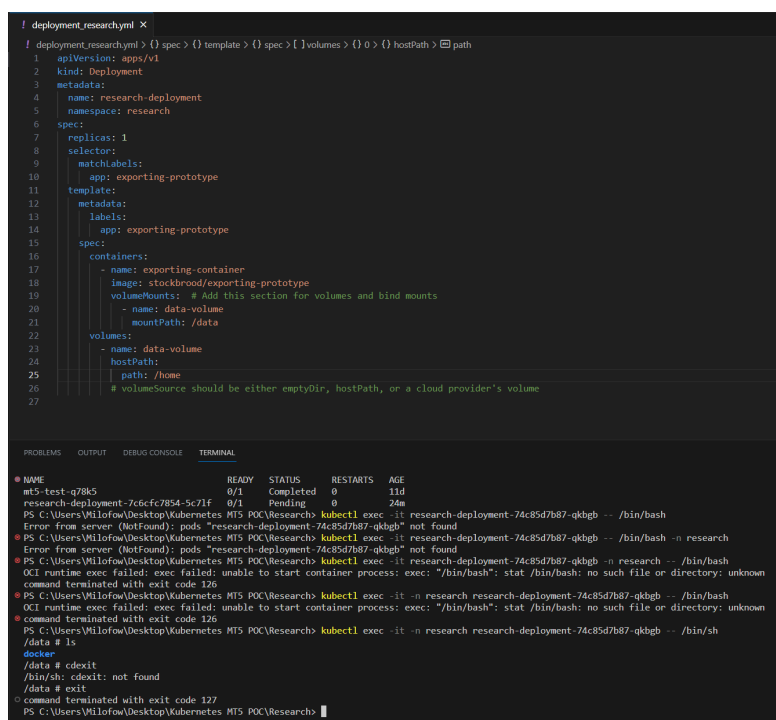
! pvc.yml x
Volumes > / pvc.yml > {} spec > {} hostPath > {} path
1 apiVersion: v1
2 kind: PersistentVolume
3 metadata:
4   name: my-pv
5 spec:
6   capacity:
7     storage: 5Gi
8   accessModes:
9     - ReadWriteOnce
10  storageClassName: standard
11  hostPath:
12    path: /home

! pvc.yml x
Volumes > / pvc.yml > {} spec
1 apiVersion: v1
2 kind: PersistentVolumeClaim
3 metadata:
4   name: my-pvc
5 spec:
6   accessModes:
7     - ReadWriteOnce
8   resources:
9     requests:
10      storage: 1Gi
11   storageClassName: standard

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\Milofow\Desktop\Kubernetes MTS POC\Research> kubectl exec -it -n research research-deployment-74c85d7b87-zbn5z -- /bin/sh
/data # ls
docker hoi
/data #
```

Bind mounts

Just like mounting a folder on the host to the docker you can do the same with this deployment in Kubernetes. In the terminal you can see that the home folder mount is visible inside the data folder inside the container.

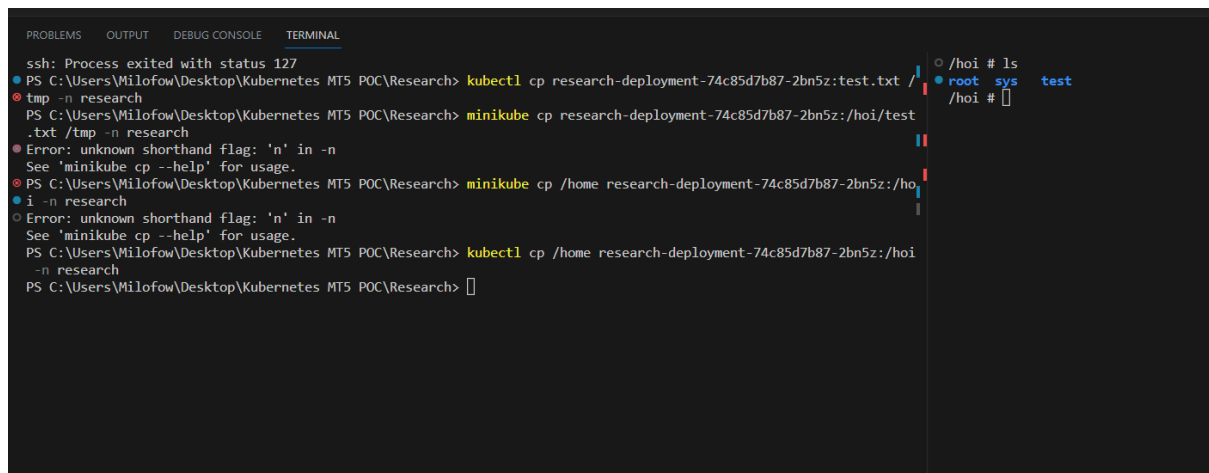


```
! deployment_research.yml x
! deployment_research.yml > {} spec > {} template > {} spec > {} volumes > {} o > {} hostPath > {} path
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: research-deployment
5   namespace: research
6 spec:
7   replicas: 1
8   selector:
9     matchLabels:
10      app: exporting-prototype
11 template:
12   metadata:
13     labels:
14       app: exporting-prototype
15   spec:
16     containers:
17       - name: exporting-container
18         image: stockbrood/exporting-prototype
19         volumeMounts: # Add this section for volumes and bind mounts
20           - name: data-volume
21             mountPath: /data
22     volumes:
23       - name: data-volume
24         hostPath:
25           path: /home
26         # volumeSource should be either emptyDir, hostPath, or a cloud provider's volume
27

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
NAME          READY   STATUS    RESTARTS   AGE
mt5-test-q78k5 0/1     Completed 0           11d
research-deployment-7c6cfc7854-5c71f 0/1     Pending   0           24m
PS C:\Users\Milofow\Desktop\Kubernetes MTS POC\Research> kubectl exec -it research-deployment-74c85d7b87-qkqgb -- /bin/bash
Error from server (NotFound): pods "research-deployment-74c85d7b87-qkqgb" not found
PS C:\Users\Milofow\Desktop\Kubernetes MTS POC\Research> kubectl exec -it research-deployment-74c85d7b87-qkqgb -- /bin/bash -n research
Error from server (NotFound): pods "research-deployment-74c85d7b87-qkqgb" not found
PS C:\Users\Milofow\Desktop\Kubernetes MTS POC\Research> kubectl exec -it research-deployment-74c85d7b87-qkqgb -- /bin/bash
OCI runtime exec failed: exec failed: unable to start container process: exec: "/bin/bash": stat /bin/bash: no such file or directory: unknown
command terminated with exit code 126
PS C:\Users\Milofow\Desktop\Kubernetes MTS POC\Research> kubectl exec -it -n research research-deployment-74c85d7b87-qkqgb -- /bin/bash
OCI runtime exec failed: exec failed: unable to start container process: exec: "/bin/bash": stat /bin/bash: no such file or directory: unknown
command terminated with exit code 126
PS C:\Users\Milofow\Desktop\Kubernetes MTS POC\Research> kubectl exec -it -n research research-deployment-74c85d7b87-qkqgb -- /bin/sh
/data # ls
docker
/data # cdexit
/bin/sh: cdexit: not found
/data # exit
command terminated with exit code 127
PS C:\Users\Milofow\Desktop\Kubernetes MTS POC\Research>
```


Docker copy

This works almost the same as the docker variant instead of docker copy you have the kubectl copy command. On the left you have the command and it copied a folder from the host to the pod.

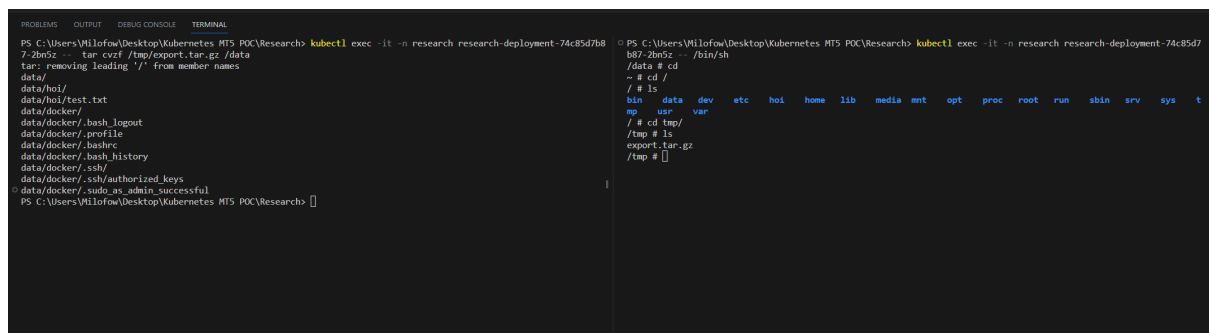


```
ssh: Process exited with status 127
PS C:\Users\Milofow\Desktop\Kubernetes MT5 POC\Research> kubectl cp research-deployment-74c85d7b87-2bn5z:test.txt /
tmp -n research
PS C:\Users\Milofow\Desktop\Kubernetes MT5 POC\Research> minikube cp research-deployment-74c85d7b87-2bn5z:/hoi/test
.txt /tmp -n research
Error: unknown shorthand flag: 'n' in -n
See 'minikube cp --help' for usage.
PS C:\Users\Milofow\Desktop\Kubernetes MT5 POC\Research> minikube cp /home research-deployment-74c85d7b87-2bn5z:/hoi
i -n research
Error: unknown shorthand flag: 'n' in -n
See 'minikube cp --help' for usage.
PS C:\Users\Milofow\Desktop\Kubernetes MT5 POC\Research> kubectl cp /home research-deployment-74c85d7b87-2bn5z:/hoi
-n research
PS C:\Users\Milofow\Desktop\Kubernetes MT5 POC\Research>

/hoi # ls
root sys test
/hoi #
```

Docker export

You can also execute the command inside the pod to export it to a .tar file. You can see here that it exports to the directory.



```
PS C:\Users\Milofow\Desktop\Kubernetes MT5 POC\Research> kubectl exec -it -n research research-deployment-74c85d7b87-2bn5z -- tar cvzf /tmp/export.tar.gz /data
tar: removing leading '/' from member names
data/
data/hoi/
data/hoi/test.txt
data/docker/
data/docker/.bash_logout
data/docker/.profile
data/docker/.bashrc
data/docker/.bash_history
data/docker/.ssh/
data/docker/.ssh/authorized_keys
data/docker/.sudo_as_admin_successful
PS C:\Users\Milofow\Desktop\Kubernetes MT5 POC\Research>

/hoi # ls
bin data dev etc hoi home lib media mnt opt proc root run/sbin srv sys t
mp usr var
/hoi # cd /tmp/
/hoi # ls
export.tar.gz
/hoi #
```

We can see from the speed results that exporting takes the most time and is also not efficient because it copies everything from that container and because our container is pretty big this is not useful.

HTTPS

I exposed the pod running the https web server with the file on it and can access It from my local machine through https.

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
PS C:\Users\Milofow\Desktop\Kubernetes MT5 POC\Research> minikube stop
  Stopping node "minikube" ...
  Powering off "minikube" via SSH ...
  1 node stopped.
PS C:\Users\Milofow\Desktop\Kubernetes MT5 POC\Research> minikube start
  minikube v1.30.1 on Microsoft Windows 10 Pro 10.0.19045.2965 Build 19045.2965
  Using the docker driver based on existing profile
  Starting control plane node minikube in cluster minikube
  Pulling base image ...
  Restarting existing docker container for "minikube" ...
  Preparing Kubernetes v1.26.3 on Docker 23.0.2 ...
  Configuring bridge CNI (Container Networking Interface) ...
  Verifying Kubernetes components...
    Using image gcr.io/k8s-minikube/storage-provisioner:v5
    Using image docker.io/kubernetes/dashboard:v2.7.0
    Using image docker.io/kubernetes/metrics-scraper:v1.0.8
  Some dashboard features require the metrics-server addon. To enable all features please run:

    minikube addons enable metrics-server

  Enabled addons: storage-provisioner, default-storageclass, dashboard
  Done! Kubectl is now configured to use "minikube" cluster and "default" namespace by default
PS C:\Users\Milofow\Desktop\Kubernetes MT5 POC\Research> kubectl get services -n research
NAME          TYPE          CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
test-service  NodePort      10.106.6.206    <none>            443:30147/TCP    4m55s
PS C:\Users\Milofow\Desktop\Kubernetes MT5 POC\Research> minikube service test-service --url -n research
http://127.0.0.1:56830
! Because you are using a Docker driver on windows, the terminal needs to be open to run it.
```

What are the benefits and drawbacks of each approach in terms of efficiency and scalability?

Dot methods

- Literature study

1. **Docker volumes**

Benefits:

1. **Improved efficiency:** Docker volumes can improve the efficiency of containerized applications by reducing the overhead of copying and storing data in the container's file system. With volumes, data can be stored externally and accessed directly by the container, which can lead to faster read and write speeds.
2. **Scalability:** Docker volumes can improve scalability by allowing multiple containers to share the same data volume, which can reduce the amount of storage required and improve overall system performance.
3. **Data persistence:** Docker volumes provide a way to store data that persists beyond the life of a container. This means that containers can be easily recreated or moved between hosts without losing data.

Drawbacks:

1. **Additional complexity:** Using Docker volumes can add complexity to the management of a containerized application, especially when dealing with large or distributed data volumes.
2. **Security risks:** If not properly secured, Docker volumes can be a security risk, since data can be accessed by other containers or even outside the container environment.
3. **Storage limitations:** Docker volumes are limited by the storage capacity of the host machine, so you may need to manage multiple volumes or use external storage solutions for large datasets.
4. **Compatibility issues:** Some Docker images may not be compatible with Docker volumes, which can make it difficult to use them in some environments.

2. Docker bind mounts

Benefits:

1. Improved efficiency: Docker bind mounts can improve the efficiency of containerized applications by reducing the overhead of copying and storing data in the container's file system. With bind mounts, data can be stored externally and accessed directly by the container, which can lead to faster read and write speeds.
2. Flexibility: Docker bind mounts provide a flexible way to share data between the host machine and container, since the data can be updated outside of the container and the changes will be immediately visible inside the container.
3. No storage limitations: Docker bind mounts do not have storage limitations since they use the file system of the host machine. This can be advantageous for large data sets.

Drawbacks:

1. Security risks: If not properly secured, Docker bind mounts can be a security risk, since data can be accessed by other containers or even outside the container environment.
2. Compatibility issues: Bind mounts can be platform-specific and may not work across different operating systems or filesystems. This can lead to compatibility issues and make it difficult to use them in some environments.
3. Additional complexity: Using Docker bind mounts can add complexity to the management of a containerized application, especially when dealing with large or distributed data volumes.
4. Performance degradation: If the host file system is slow or the data being accessed is large, performance may be negatively impacted by using bind mounts.

3. The Docker copy command

Benefits:

1. Ease of use: The Docker copy command is easy to use and allows developers to quickly copy files and directories from the host machine to a Docker container.
2. Flexibility: Docker copy allows developers to copy specific files or directories to specific locations in the container, giving them flexibility in how they manage the files within the container.
3. No additional complexity: The Docker copy command does not add any additional complexity to the management of a containerized application since it simply copies files from the host machine to the container.

Drawbacks:

1. Performance degradation: The Docker copy command can be slow and inefficient, especially when copying large files or directories. This can result in longer build times and slower container startup times.
2. Limited scalability: The Docker copy command does not scale well when dealing with large or distributed data sets. In these cases, it may be more efficient to use Docker volumes or bind mounts to manage the data.
3. No data persistence: The Docker copy command does not provide any data persistence since the copied files exist only within the container.

4. The Docker export command

Benefits:

1. **Portability:** The Docker export command allows developers to easily move a container's file system to another machine or environment, making it highly portable.
2. **Security:** Exporting a container's file system as a tar archive can be useful for auditing or security purposes, as it allows developers to inspect the file system for any potential security issues.
3. **No additional complexity:** The Docker export command does not add any additional complexity to the management of a containerized application since it simply exports the container's file system as a tar archive.

Drawbacks:

1. **Limited functionality:** The Docker export command only exports the container's file system and does not include any metadata or information about the container itself. This can make it difficult to recreate the container exactly as it was on another machine.
2. **No data persistence:** The Docker export command does not provide any data persistence since the exported files exist only as a tar archive and must be re-imported into another container to be used.
3. **Performance degradation:** The Docker export command can be slow and inefficient, especially when exporting large file systems. This can result in longer export times and slower container startup times.

Conclusion

Dot-methods

- Multi-criteria decision making

While reading about the pros and cons of different approaches and working with it in our specific scenario. We came to the conclusion that all of the researched examples are possible and to keep into account that speed is a big criteria for us we want to say that using volumes or exporting could be an issue. HTTPS involves implementing extra services and another pod to run constantly, so that's not where our preference is going to. Because the implementation of the PVC was pretty straightforward and simple we want to go that way. Therefore we have a central point to collect results and from there we can use the copy command to get it to the back end for example. Maybe it's also possible to send it using RabbitMQ to ensure more reliability, but for now We know how we can import and export to and from a container In a scalable environment.