MASTER'S THESIS 2021

Sharing local files with Kubernetes clusters

Martin Jakobsson, Johan Henriksson

ISSN 1650-2884
LU-CS-EX 2021-37

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2021-37

Sharing local files with Kubernetes clusters

Martin Jakobsson, Johan Henriksson

Sharing local files with Kubernetes clusters

Bridging the gap between developers and cloud notebooks

Martin Jakobsson
ma5210ja-s@student.lu.se

Johan Henriksson
johan@backtick.se

July 1, 2021

Master's thesis work carried out at Backtick Technologies.

Supervisor: Markus Borg, markus.borg@cs.lth.se

Examiner: Niklas Fors, niklas.fors@cs.lth.se

Abstract

Data science development largely revolves around working in notebooks, and these usually run in a cluster environment. These cloud notebooks do however come with limitations. Support for tools like version control, testing, refactoring and linting is limited, which negatively impacts developer experience and software quality.

We propose a networked file system for sharing local files and directories with a remote cloud server. This allows the developer to work in a cloud notebook without having to upload files to an external storage provider, and enables the use of traditional development tools since all files are stored on the local machine. The networked file system is implemented as a FUSE file system and is made available to Kubernetes as a custom volume driver.

We show that it is possible to build a widely compatible file sharing solution for any Kubernetes workload. It is easy to use and has reasonable performance.

Keywords: file system, fuse, kubernetes, docker, notebook, container, cowait

Acknowledgements

We would like to thank our supervisor at the Department of Computer Science, Markus Borg, for his support throughout the project. Your helpful advice and unwavering guidance have enabled us to reach new heights.

The completion of this thesis would not have been possible without the people at Backtick Technologies. Your unparalleled support and insightful suggestions have been invaluable.

We would also like to thank to everyone who participated in the interviews. Your positivity and feedback have inspired us to continue working on the project in the future.

Finally, we would like to thank our families and friends for supporting us throughout our studies at Lund University.

Lund, June 2021.

Contents

1 Introduction **7**
1.1 Research Questions 8
1.2 Related Work 8
1.3 Contributions 9

2 Background **11**
2.1 File Systems 11
2.2 Containers 13
2.3 Kubernetes 13
2.4 Computational Notebooks 14
2.5 Cowait 15
2.6 Remote Procedure Calls 18

3 Approach **19**
3.1 Cloud Notebook Storage 19
3.2 Requirements 21
3.3 Networked File Systems 21
3.4 Client Application 22
3.5 File System 22
3.6 Networking 22
3.7 Kubernetes CSI Driver 23
3.8 Solution Overview 24
3.9 Cowait Notebook Integration 24

4 Implementation **25**
4.1 Starting Point 25
4.2 Network Architecture 26
4.3 Client Application 28
4.4 Kubernetes Integration 29
4.5 Cowait Notebook Integration 30

5 Evaluation 33
5.1 Performance Evaluation 33
5.2 Usability Evaluation 40

6 Discussion 47
6.1 Implementation (RQ1) 47
6.2 Performance (RQ2) 48
6.3 Interview Feedback (RQ3) 50
6.4 Threats to Validity 51
6.5 Future work 52

7 Conclusions 53

References 55

Appendix A Evaluation Exercise 59

Chapter 1
Introduction

Contemporary data science revolves around working in computational notebooks, an environment based on interactive code interpreters in web browsers that allow quick, iterative development. The notebooks themselves are usually running on a remote machine or cluster, allowing the user to leverage additional compute resources available in data centers. Popular notebook software is slowly evolving towards full-fledged development environments, but still lag far behind more typical integrated development environments (IDE) available for local use. Since most development tools developed in the past decades expect to work on files located in the user's local file system, they would need to be reinvented to work in a cloud setting. The result is that support for version control software, static analysis, linting and other widely used tools is currently very limited.[1]

The idea for this thesis project was born out of the realization that, perhaps, some of these problems could be solved or at least partially mitigated if the user's local files were accessible though a networked filesystem that was shared with the cloud notebook. That way, the user could still work in their favorite editor and use all the tools available for local development, while still being able to use the cloud notebook for the things it excels at - data exploration with the advantage of distributed cloud computing.

Such a filesystem could potentially have other use cases related to working with cloud servers, and ideally it wouldn't be tied to a specific application. The goal of this thesis is to design a user-friendly, reliable and widely compatible solution to sharing files residing on a local machine, such as a developer laptop, to a cloud server. The user's files and folders will be available to applications running on the cloud server through a custom file system implemented using FUSE[1].

In order to achieve compatibility with as many different applications as possible, the network file system will be implemented as a custom storage driver for Kubernetes. Kubernetes is a popular open source cluster orchestration software that is widely used to deploy and manage modern applications. It is available as a managed service from all major cloud

[1] File System in Userspace, `https://github.com/libfuse/libfuse`

providers, but it can also be used for on-premise servers. By building on Kubernetes, the file system becomes a generic component without any ties to any specific application.

To evaluate whether the implementation is useful in a real-world setting, it will be integrated and evaluated as part of Cowait[2] Notebooks, an experimental cloud notebook solution developed at Backtick Technologies.

1.1 Research Questions

In this thesis, we seek to answer the following research questions:

1. **RQ1**. Could a custom Kubernetes volume driver facilitate file sharing between cluster nodes and developer machines?
2. **RQ2**. Would the performance of such a system be acceptable? Which factors impact the performance of such a solution?
3. **RQ3**. Does a shared filesystem improve developer experience when working in cloud notebooks?

1.2 Related Work

In this section we present some of the related work. First a brief history of computational notebooks is given, followed by problems related to working in notebooks. The suitability of FUSE-based file systems is then covered, with focus on network file systems. Finally, an example is given of how notebook persistence has been solved with a method similar to the one presented in this report.

1.2.1 Computational Notebooks

The history of computational notebooks began with an article written by Knuth in 1984 [8] in which Knuth introduced the term *literate programming*. This term referred to the combination of computer code and documentation into a single document, which could be seen as a work of literature. A more general version of literate programming is *literate computing*, where the computer code and documentation is accompanied by content like tables, graphs and images. This computing paradigm is covered in an article written by Fog et al. [3]. The *computational notebook* can be seen as a special case of literate computing.

Throughout the years countless notebook solutions have been developed, and in an article by Lau et al. [11] 60 different notebook solutions were categorized using 10 dimensions of analysis. Two of the dimensions most relevant to this report were *execution environment* and *data sources*. The solution presented in this report enables Cowait notebooks to execute code in a *remote multi-process* execution environment using *local files* as data sources. This can be compared to Jupyter notebooks where both code execution and data is local, and Databricks where code is executed in a remote multi-process execution environment but the data sources may not be local.

[2] Cowait: distributed python framework, `https://cowait.io`

Computational notebooks have evolved far since the article by Knuth, but there are still problems associated with working in notebooks. These problems were investigated in an article by Chattopadhyay et al. [1] where nine pain points were identified. The most important yet difficult notebook activities were identified as: refactoring code, deploying to production, exploring notebook history and managing long-running tasks. To mitigate these issues several tools [6, 7, 4] and notebook environments [11] have been developed, but more work is still needed. Similar problems were discovered by Singer [14].

1.2.2 File Systems

In an article by Tarasov et al. [15] the authors study the performance impact of user-space file systems to answer the question of whether or not these types of file systems are practical for modern use cases. Findings show that while kernel-space designs are indeed much faster, the real-world performance of a user-space file system is indeed enough for many applications, including networked file systems similar to the one presented in this thesis.

The original paper outlining the design of the widely used *Ceph* file system [18] describes the use of a FUSE-based file system for their client application. The paper also discusses some of the performance issues that commonly occur in networked file systems, some of which are highly relevant to possible future improvements of the implementation presented in this report.

1.2.3 Cloud Notebook Persistence

While the solution presented in this report uses the user machine for persistence, there are also solutions where cloud storage is used. One such solution was presented in an article by Teslyuk et al. [16]. They deployed JupyterHub on a supercomputer and used CephFS to manage 75 TB of storage distributed on 64 hard drives. The Ceph file system was connected to Jupyter using a Kubernetes CSI driver. Users could then connect to the supercomputer and use Jupyter Notebooks with all files stored on the supercomputer. They concluded that it was not straightforward to set up, but worked well once it was running. The solution presented in this report is more light weight which makes it easier to get started.

1.3 Contributions

Implementation

The design and implementation project was performed collaboratively by both authors in a pair-programming fashion. The primary responsibility for the evaluation work was divided, with Martin responsible for the performance evaluation and Johan responsible for the usability evaluation.

Thesis

Similarly to the implementation of the system, both authors have been involved in writing the Introduction, Background, Approach, Implementation and Conclusion chapters. Johan had

primary responsibility for writing the evaluation and discussion parts regarding the usability study, and Martin was primarily responsible for the performance study. Both authors were present during all of the usability interviews.

Chapter 2
Background

This chapter provides a summary of the core concepts and technologies used in this thesis.

2.1 File Systems

In most modern operating systems, the data is presented to the user as a tree-like hierarchy of files and directories. When an application interacts with a file or directory, the operating system kernel forwards the operation to the appropriate file system driver, which, in turn, is responsible for controlling how the data is stored and retrieved.

On UNIX-like systems such as Linux, the file and directory hierarchy can be composed of multiple file systems. File systems can be *mounted* to any empty directory in the hierarchy. This path is referred to as the *mount point*. The mounted file system's file and directory hierarchy then appears as a sub-tree in the global directory structure with its root attached at the mount point. [5, 13] This can be seen in figure 2.1.

It is important to note that a file doesn't necessarily represent a block of data on a physical hard drive. There are file systems designed for storing data on other computers over a network, in volatile memory and in many other ways. The kernel can also use the file hierarchy to expose information about the system itself, such as files representing devices commonly found in `/dev`. These files do not exist on any physical storage medium, they are *virtual* files whose contents are dynamically generated when reading.

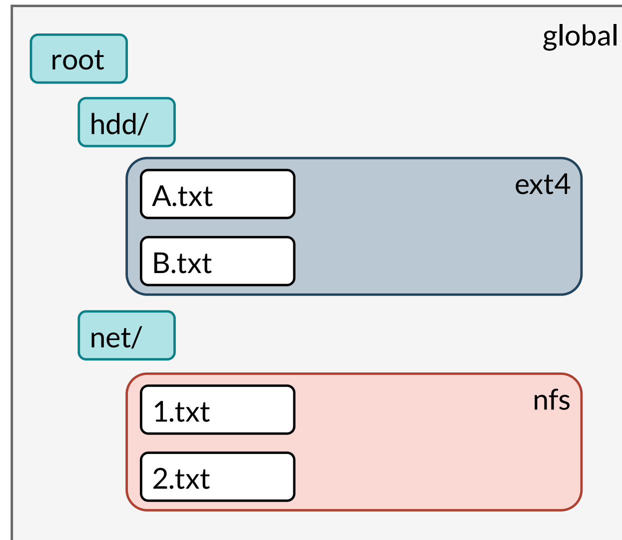


Figure 2.1: Example filesystem hierarchy. Two different file systems are mounted at *hdd* and *net*. *hdd* is a physical hard disk formatted as ext4, and *net* is a network share provided by the Network File System (NFS). Each file system contains two files.

2.1.1 FUSE

FUSE[1] is an acronym for "File system in User space" and enables file systems to be implemented as regular userspace programs. [5] This means that a file system can be developed independently from the Linux kernel, and be distributed like any other program. All FUSE based file systems share a single kernel module which forwards all system calls related to file and directory operations to the correct userspace file system implementation. This can be seen in figure 2.2.

FUSE greatly lowers the barrier to entry for creating new file systems by allowing the developer to avoid all the complexity of kernel development and providing a simplified API for the userspace implementation. However, forwarding all file operations to user space incurs additional overhead, which results in lower performance compared to kernel space implementations. For some types of file systems this overhead would be of great concern, but for network-based file systems the overhead would be negligible. [15]

[1] FUSE website: `https://github.com/libfuse/libfuse`

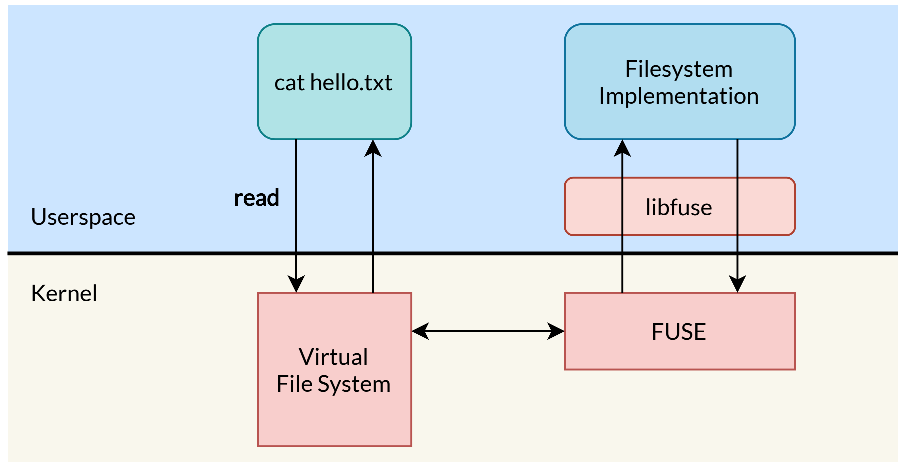


Figure 2.2: FUSE overview. System calls are forwarded to the file system implementation by the FUSE kernel module.

2.2 Containers

When developing software it is common for the application to depend on external packages. These packages need to be installed for the application to run. A developer may install these packages on their computer to get the application to run, but if the application is to be deployed to a server the developer needs to ensure that the same packages are installed on the server machine. This is a cumbersome and error prone process, and a popular way to alleviate these issues is to use containers.

A container is a unit of software that packages code and all its dependencies, so that the application can run reliably from one computing environment to another. Docker is a widely used system for creating, managing and running such containers. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings [2].

Containers provide similar isolation as to virtual machines, with some important differences. Virtual machines allow multiple operating systems to share the same hardware, containers allow multiple applications to share the operating system kernel in a secure fashion. Sharing a single kernel reduces the overhead cost of process isolation.

2.3 Kubernetes

It is not uncommon to deploy an application to a group of servers, a *cluster* as opposed to a single machine. There are many benefits to a clustered setup, such as redundancy and the ability to scale to larger workloads, but maintaining such a setup manually can be difficult and error prone. It is therefore common to use a cluster orchestrator, the most popular being Kubernetes.

As stated on their website, Kubernetes is an *open-source system for automating deployment, scaling, and management of containerized applications.* [9] Kubernetes divides the cluster into different types of resources, most importantly Pods, Deployments, Services and Volumes.

- *Pods* are groups of one or more containers that form "deployment units". Pods can be thought of as a group of processes that must always run alongside each other on the same cluster node. For example, a web server running in a single container, serving files from some underlying storage solution, and exposed to the internet on port 80.
- A *Deployment* manages Pods, ensuring that the desired number of pod instances run at all times. Deployments are also responsible for rolling updates of Pods, allowing updates without downtime.
- *Services* are used to load balance network requests to a group of pods. They provide a known hostname and divide incoming requests among the worker pods.
- *Volumes* allows us to request persistent storage from the cluster. Pods can attach and use these volumes to store data across restarts. Many different storage drivers are supported, but their differences are invisible to the pods.

2.3.1 Container Storage Interface

Previously, storage drivers for Kubernetes where merged directly into the Kubernetes source code. However, this quickly became cumbersome as Kubernetes exploded in popularity and many organizations wanted to integrate new storage solutions.

This prompted the creation of the Container Storage Interface (CSI)[2]. In a way, CSI is to Kubernetes what FUSE is to the Linux kernel - a low level component that provides a means for external, higher level applications to integrate new storage options. CSI is itself a volume driver that delegates file system operations to a CSI driver implementation.

CSI drivers are implemented as two separate components, a *Controller* and a *Node*. The CSI node application runs on each cluster node, and is responsible for managing volume mounts and connecting them to Kubernetes pods on that particular host machine. CSI Controllers are responsible for managing the volumes themselves. In a typical case, it might involve provisioning and attaching disks to a virtual machine, and then mounting that disk into a pod.

2.4 Computational Notebooks

Data science development largely revolves around working in notebooks, an environment based on interactive code interpreters in web browsers that allow quick, iterative development. The idea behind a notebook is to have an interpreter running in the background, preserving state for the duration of the programming session. Code is organized into *cells*, which can contain one or more statements. The cells can then be executed independently of each other, in any order, but they all mutate the shared interpreter state. This gives the programmer the ability to re-run portions of the program while keeping some of the previously generated state, which can be very useful when iterating on a problem. On the other hand, it can quickly become difficult to trace the path of execution that led to the result, and subsequent executions of the notebook may yield different results or even errors.

[2]CSI specification: `https://github.com/container-storage-interface/spec`

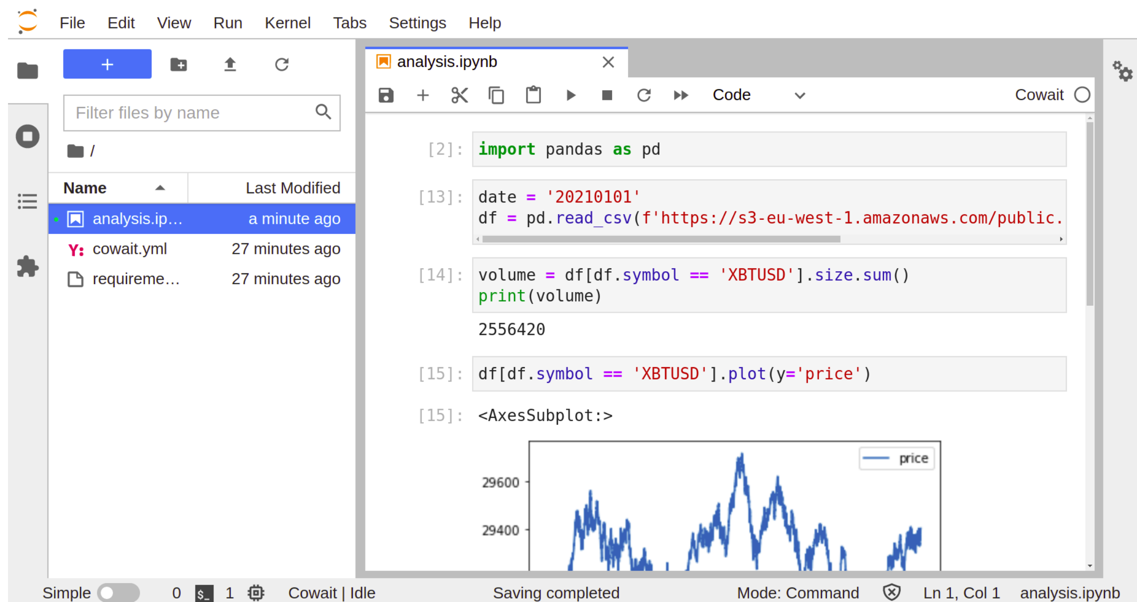


Figure 2.3: A Jupyter notebook running in JupyterLab

Notebooks usually have support for displaying various types of interactive output, such as graphs or tables. This makes them very suitable for working with and exploring datasets. An example of a Jupyter notebook can be seen in figure 2.3.

2.5 Cowait

Cowait[3] is a framework that aims to make it easier to execute Python code on Kubernetes. It consists of two core parts: a novel workflow engine built directly on top of Docker and Kubernetes, and a build system to easily package your code into containers. Together, they form an abstraction of containers and container hosts that allows developers to leverage the power of both containerization through Docker and cluster deployment using Kubernetes without having to know their ins and outs. This is achieved by hiding the complexity behind simple concepts that should be familiar to developers of varying experience.

Cowait provides a user-friendly way to:

- Develop and test distributed workflows on your local machine, with minimal setup.
- Manage dependencies for Python projects.
- Unit test your workflow tasks.
- Easily deploy your code on a Kubernetes cluster without any changes.

Like most workflow engines, Cowait organizes code into Tasks. A Task is essentially nothing more than a function, and just like a typical function, it can accept input arguments and return values. Similarly they may also invoke other tasks, with one key difference: a call

[3] Cowait website: https://cowait.io

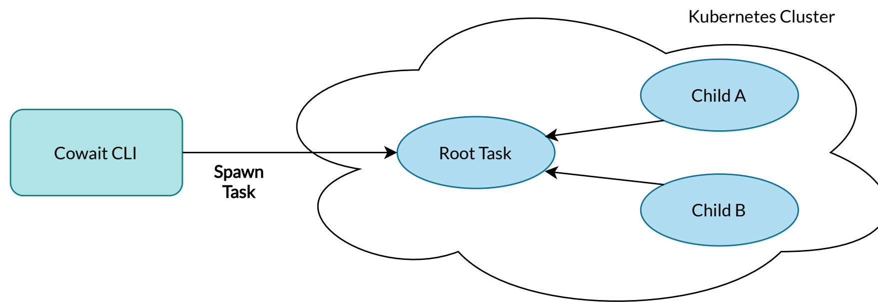


Figure 2.4: Cowait Tasks example. The user creates the root task using the Cowait command line tool. The root task in turn creates two more sub-tasks, Child A and Child B.

to invoke another task will be intercepted by the Cowait runtime and executed in a separate container - potentially on a different machine. The idea that makes Cowait different is that the tasks are allowed to interface directly with the cluster orchestrator. This means that tasks can start other tasks without having to go through a central scheduler service. Instead, tasks create other tasks on demand, and they communicate with their parents using web sockets, as illustrated by figure 2.4.

Tasks are automatically bundled into container images along with all their dependencies by the build system. They can then be distributed using container registries.

2.5.1 Cowait Notebooks

Cowait's task management system is based on running containers, and lends itself well to running arbitrary software. Additionally, it has support for automatically generating subdomains and routing web traffic to a specific container. These features form the basis for a notebook-on-demand hosting system.

The idea is to run a JupyterLab instance within a Cowait task. JupyterLab is one of the most popular open source notebook applications used widely in the industry. It is written in Python, which makes it easy to integrate into Cowait. Once the Jupyter task is started in a cluster, it is automatically assigned a public URL that the user can connect to. This allows on-demand hosting of notebooks in any Kubernetes cluster, without any additional setup required.

Running within a Cowait task allows the notebook to access the underlying task scheduler and allow sub-tasks to be launched directly from the notebook cells. This allows the user to easily execute background tasks on the cluster.

Since each JupyterLab instance can run multiple notebooks simultaneously, the custom kernel acts as if it were a separate task running in its own container. Cowait has a concept of *virtual* tasks for exactly this purpose. By introducing a virtual task representing each running kernel, the virtual task can act as the parent for all sub tasks launched by that kernel, which helps to direct the flow of information to the correct place. This can be seen in figure 2.5.

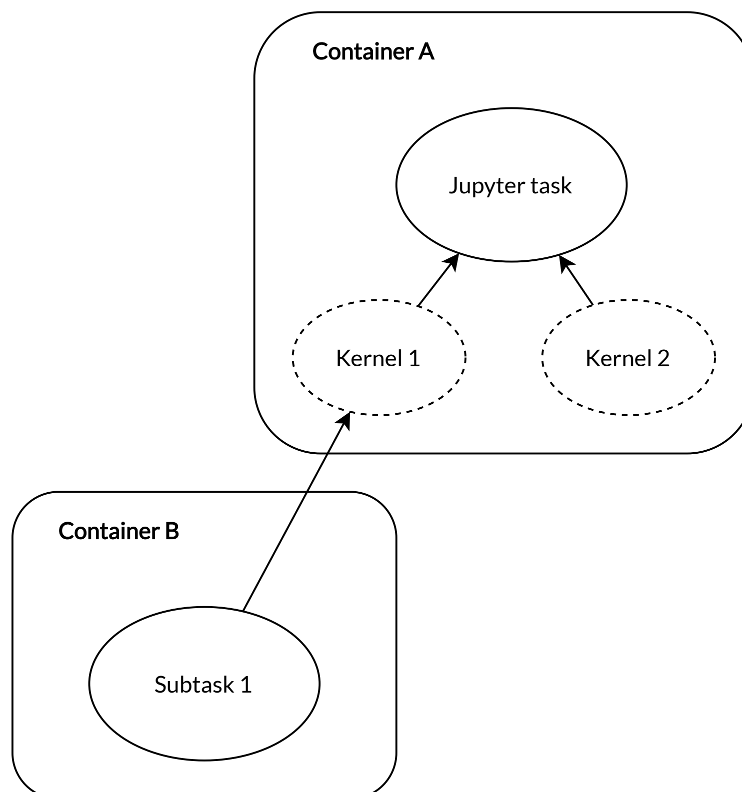


Figure 2.5: An example task hierarchy for a JupyterLab instance running two kernels, one of which has an executing Cowait subtask. Both the Kernel tasks connect to the Jupyter host task, identifying themselves as virtual tasks. Subtask 1 is started by the user from within the notebook, and it connects to the web socket server provided by Kernel 1. Note that the Jupyter Task, Kernel 1 and Kernel 2 execute in the same container, while Subtask 1 resides in a separate container, potentially on a different machine.

Notebook Execution

In addition, the build system can be used to package these workflow-integrated notebooks into task container images. This provides a path for migrating existing notebook software and make it easier to get started with Cowait, reaping most of the benefits such as dependency management, reproducible executions and ease of deployment. Cowait has a custom notebook executor module, that can parse and execute saved Jupyter notebook in an environment compatible with the custom Cowait kernel. The executor module communicates with the workflow engine to provide input parameterization, return output values and allow child task scheduling. The executor module can run tasks in headless mode and has no dependence on Jupyter.

2.6 Remote Procedure Calls

A Remote Procedure Call (RPC) is a request from a client to a server to perform a procedure, typically transmitted over a socket. The procedure can accept parameters and return a value that is passed back to the calling client. It works in a similar way to a function call, except that the caller and the function reside in different processes, and possibly on different physical machines.

One of the most widely used RPC frameworks is called *gRPC*[4]. It is open source and was initially developed by Google in 2015. The gRPC framework is commonly used for communication between microservices in backend systems, and to connect clients to backend services. It is also used extensively throughout the Kubernetes ecosystem.

The gRPC system is a client-server system, meaning that there is a client and a server and the client initiates requests to the server. It is not possible for the server to initiate requests to the client. This is mainly a design decision of the gRPC system, and not a limitation by the underlying HTTP/2 connection, which will be mentioned again later in this report.

[4]gRPC website: `https://grpc.io`

Chapter 3

Approach

This chapter describes the overall approach used to design a shared file system between a local machine and a Kubernetes cluster pod. The first section presents an overview of how file storage typicall works in a cloud notebook. In the second section, a list of requirements that the file system needs to uphold in order to solve the problem in a satisfying manner. The third section explains how a networked file system enables the local machine to act as a storage provider for a remote system. The fourth section the FUSE file system is introduced as the backbone of the solution, and the fifth section explains how the system is integrated into Kubernetes. Finally, an overview of how the solution is fulfills the requirements is presented.

Additional implementation details can be found in chapter 4.

3.1 Cloud Notebook Storage

In traditional notebook development the notebook software is initially run on the local machine, as seen in figure 3.1. When the need for compute power increases, a natural solution is to move the notebook to a more powerful machine rented from a cloud provider, and to use a storage provider for file storage. This setup can be seen in figure 3.2.

A common problem with this setup is the need to transfer files to and from the cloud storage provider. This task would be less of a problem if the files could instead be streamed directly from the user's machine, which is exactly what the file system presented in this report enables, as described in figure 3.3.

With the file system presented in this report, the files are stored on the user machine but the cloud notebook can access them as if they were stored on the machine serving the notebook. The user's machine acts like a temporary storage provider.

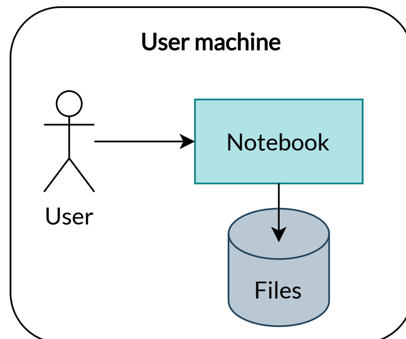


Figure 3.1: Notebook usage locally. All files are located on the same machine as the running notebook.

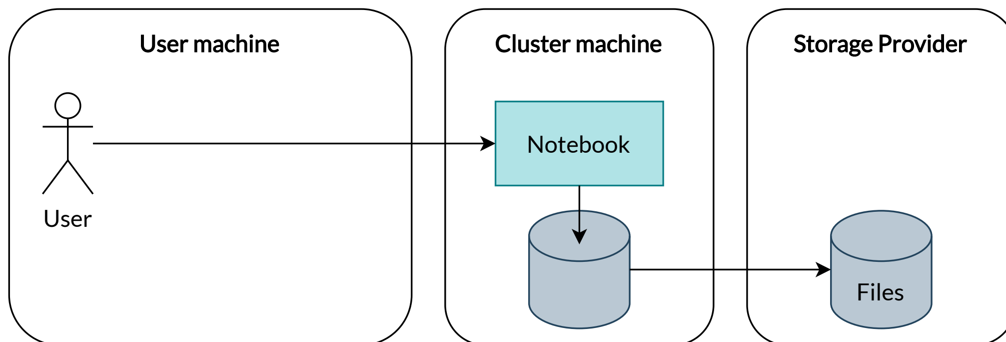


Figure 3.2: Notebook usage on a cluster using a storage provider for file storage. The file system on the cluster machine asks the storage provider for files when needed.

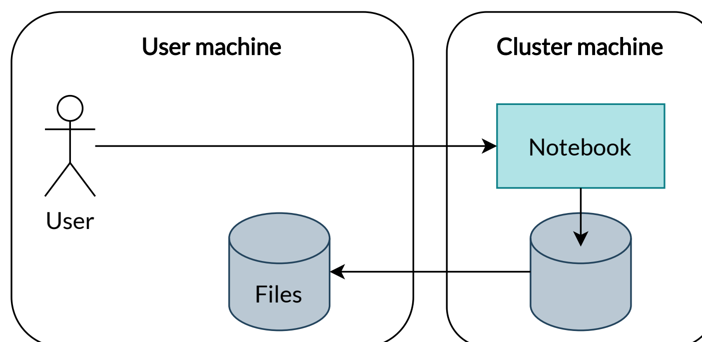


Figure 3.3: Notebook usage on a cluster using the file system presented in this report. The user machine acts as a storage provider.

3.2 Requirements

In order to allow the developer to use standard tools designed for use on a local file system, we must devise a way to bridge the gap between the file systems one the cloud notebook server and the developer's local machine. Currently, most cloud-hosted notebooks expect the user to upload all files related to their project to cloud storage which is then accessible through the notebook web interface. When the files leave the local machine, the ability to use tools like real editors and version control systems is lost.

It follows that the optimal solution would be one where the files simply never leave the user's machine. Since code files are small, content and changes could be streamed to the cloud server directly from the user as needed. A setup like this resembles a traditional networked file system, but where the files are stored on the users machine rather than a server. Having the master copy of each file on the user's machine limits the effect of possible data loss due to lost connections or synchronization problems. This way, there is also no need to transfer files until they are actually needed.

A potential file sharing solution for this purpose would need to fulfill the following requirements:

- The shared files are stored on the user's machine. A client application connects to the remote server, and provides access to files for use by applications running on the target server. This is the opposite of most networked file systems, where the user normally accesses files stored on the server machine.
- File access should be provided to the target machine using regular POSIX filesystem APIs. This ensures compatibility with all existing software.
- Must be compatible with Kubernetes, ideally without any dependencies required in the target Pod.
- End users must not be required to have open ports reachable from the public internet.
- User-side setup must be straight forward. The client application should be a single binary without external dependencies that can be shipped along with other software.

3.3 Networked File Systems

Using a remote machine as a storage provider is not a novel concept, in fact, operating systems have had support for network storage in various forms for decades. However, they are typically designed in a way that makes them unsuitable for the task presented - where the user wants to share files *to* a remote server machine. In a traditional networked file system, the user's machine connects to a server machine which stores the files and makes them available to the user.

In order to solve the problem of sharing files located on the user's machine to a remote server, the system would need to work in reverse. To make matters worse, the user's machine is almost never open to connections from the public internet, which means that the user can not easily act as a server in a traditional sense. Rather, the target server must listen for

incoming connections from users offering to serve files. Requests to the file system on the target machine can then be forwarded to the user.

3.4 Client Application

The solution needs a component to carry out requests to perform file and directory operations on the user's computer. This piece of software will be referred to as the *client application*, and its responsibility is to accept network requests from the target machine, and translate them into file operations on the user's end. The client application should be implemented as a simple command-line utility that can be executed in the background by other processes. Ideally, it should support all major operating systems: Linux, Mac OS, and Windows (through the Windows Subsystem for Linux).

The Go programming language was chosen for the implementation of the client application for several reasons. Go is a modern systems programming language with a focus on networking, which makes it a good candidate for our purposes. It offers a large ecosystem of libraries, and it compiles to native binaries for all of the major platforms. All dependencies are statically linked, which means that the resulting binary does not have any external library dependencies.

3.5 File System

In order for the file sharing solution to be successful, it must be compatible with any existing software. The best way to achieve this is by implementing it as a custom file system driver, since that immediately makes it compatible with any software that is using the traditional file and directory API offered by the operating system.

A file system is typically implemented as a core part of the operating system, which makes it difficult to create, distribute and install new file systems. On Linux such a file system implementation would have to be implemented as a kernel module, comes with all sorts of extra complications related to writing and debugging kernel software, as well as the risk of compromising the overall stability of the system.

The easiest way to implement a custom file system is to leverage FUSE. While userspace filesystems have some disadvantages in terms of performance, it should be negligible compared to the costs of transferring data over a network connection. The FUSE kernel module is also widely available in most cluster settings, whereas a custom kernel module would have to be pre-installed on the cluster nodes to achieve compatibility with Kubernetes.

3.6 Networking

The idea is to create a simple networked filesystem by intercepting file and directory operations on the target system, and then forwarding them over the network to the user's computer, where a client application carries out the actual file operations and returns the results.

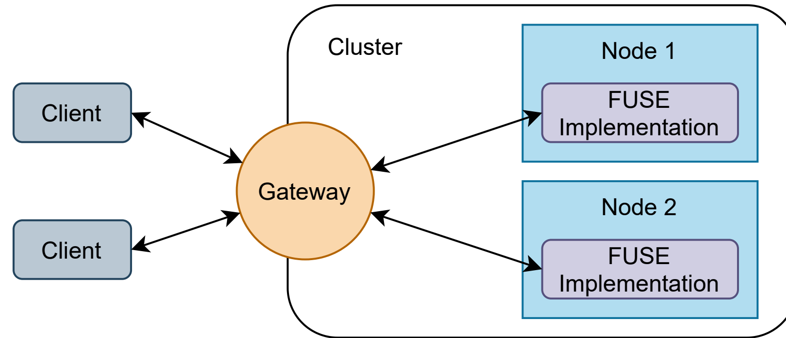


Figure 3.4: Illustration of how clients reach target nodes through the gateway.

3.6.1 Protocol

The network protocol defines the rules of communication between the different components in the system. *gRPC* was selected as the communication framework, which brought several advantages, the most important being a simple way to define the protocol messages and automatically generating code for message serialization. However, gRPC is designed around a client-server model where calls are always originating from client to server, with limited options for server-to-client communication. This design is not perfectly suited for the needs of this project, but the benefit in terms of ease of development were deemed worth the necessary sacrifices for the purposes of the thesis.

Because the primary goal of the protocol is to transfer file system operations between the user and target machine, a majority of the messages are designed as close approximations of the FUSE operations they are meant to represent. A more detailed description of the protocol messages is available in section 4.2.1.

3.6.2 Gateway

For the client to be able to communicate with the target machine, the target must be accessible at a well known address reachable through the internet. Because it is also difficult to know exactly which cluster node will host the workload when setting up the filesystem, and the cluster nodes might not even be reachable from the internet, the solution to all of these problems became to introduce a proxy server, referred to as the *gateway*. The gateway facilitates communication between client applications and the cluster nodes, as shown in figure 3.4.

3.7 Kubernetes CSI Driver

Applications running on Kubernetes access persistent storage through a concept called *Volumes*. Volumes represent storage systems mounted into the application container at a given path. The best way to develop a widely compatible storage solution for Kubernetes is to implement a custom Volume driver. This allows the Kubernetes cluster orchestrator to take on the responsibility of mounting and unmounting the file system to the target container.

The Container Storage Interface is designed to streamline the process of creating custom volume driver by allowing them to be built as external applications instead of having to be merged in to the Kubernetes source. Using CSI to build a proper volume driver for Kubernetes decouples the file system implementation from a particular application. The driver can be used as a storage provider for any deployment running on the cluster.

An additional benefit is that the driver and the file system integration run in separate pods that are granted additional privileges and access to the host systems `/dev/fuse` driver. This greatly improves security since each pod using the file system would otherwise have to run in privileged mode, and the running program could potentially escape its container.

Building a standardized Kubernetes component also streamlines the process of deployment and configuration for future users of the file system.

3.8 Solution Overview

Together, these parts make up all the required components required to create a custom volume type for Kubernetes that allows pods to mount directories from remote machines.

- Creating a proper file system implementation using FUSE ensures compatibility with all existing applications that interact with files and directories.
- Implementing a CSI driver allows native integration with Kubernetes, enabling the solution to act as a storage provider for any type of application.
- A client application written in Go allows us to compile native binaries for all major operating systems, and static linking ensures that there are no external dependencies.
- The gateway service ensures that users can connect and provide files to the cluster without them having to be reachable from the public internet.

If successful, such a system would fulfil all the defined requirements. This would answer RQ1 by showing that it is indeed possible to create a Kubernetes volume driver that solves the problem of sharing files between the user's machine and a remote cluster.

3.9 Cowait Notebook Integration

The original idea for the file system stems from the problem of sharing files with cloud notebook software. To test if the file system mitigates these problems, and if it is useful in a real-world scenario, the filesystem will be integrated into the Cowait notebook software. This will help answering RQ3, i.e. to see if a shared filesystem can improve common workflows when working with cloud notebooks.

A successful integration will need to automatically set up a Kubernetes volume representing the user's local directory, and connect the client application to this volume prior to launching the notebook. When the user closes the notebook, these resources should be cleaned up.

Chapter 4
Implementation

This chapter provides additional details on the implementation of the networked file system and the custom volume driver. Section 4.1 describes the starting point for developing the custom file system and a naive approach to implement the networking. Section 4.2 expands on this, introducing a viable network architecture and providing an overview of the protocol. The third section describes the implementation of the client application, and the rest of the chapter outlines the development of the Kubernetes volume driver.

4.1 Starting Point

An implementation of a networked filesystem that fulfils our requirements outlined in section 3.2 will have to be composed of a number of separate components. At the very least, there has to be a program that interacts with the files available on the users machine, and another program that facilitates access from the target machine.

The easiest way to build custom file systems is to use the *FUSE driver*. FUSE is a kernel module that forwards file-related system calls to a userspace process, which is then responsible for carrying out the actual file operations. In order to make a networked file system, the implementation forwards these system calls to a different machine over the network, and returns the result. Creating a FUSE file system is fairly straight-forward, as there are high-level libraries available for most languages, and examples are widely available.

The naive approach is to run the file system implementation on the target machine and have it act as a server that accepts incoming connections from user clients. Clients can then offer to provide the backing volume (the local folder on the user's machine) that is then mounted to the target. Once a client is accepted, subsequent file and directory operations to the mounted directory is forwarded to the user's filesystem over the network. A write operation is illustrated in figure 4.1.

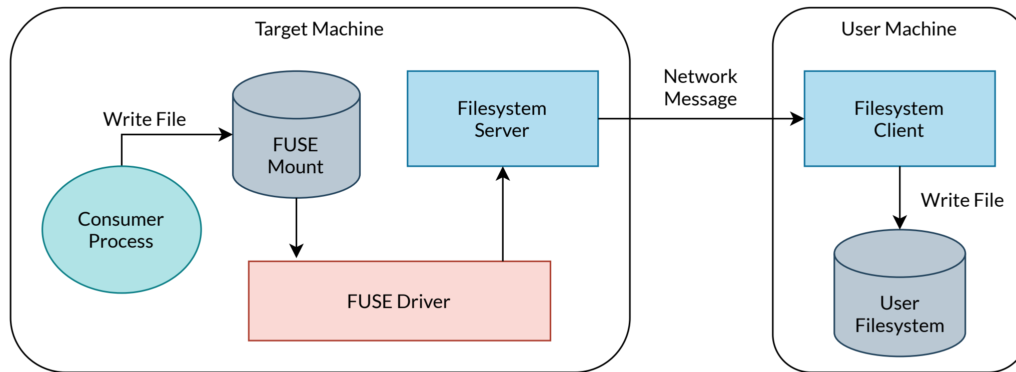


Figure 4.1: Initial design of the networked FUSE filesystem. In this design, the client and server communicate directly. Requests to the file system are forwarded and handled by the client application.

4.2 Network Architecture

Since one of the design requirements is that the user must not be required to expose their computer to the internet, the naive design presented in section 4.1 does not work, since it assumes that the user can act as a server. If one wishes to avoid this, there are only two suitable options for the network architecture. Either the target machine acts as a server (as shown in figure 4.1), which would mean that any target machine must be exposed to the internet, or there has to be a third component acting as an intermediary that both the client program (on the user machine) and the target program can communicate through. This design is illustrated in figure 4.2.

Because the intended target machines are Kubernetes pods, it became apparent that the intermediary solution was the most practical. It would be unreasonable to require all cluster nodes to have publicly accessible IP addresses, and even if it were a requirement, additional complexity would have to be introduced to discover the IP address of the target pod once it is created. By introducing an intermediary service, henceforth known as the *gateway*, we can set up a well known public endpoint for the entire cluster that clients can connect to.

As a side benefit, the intermediary gateway service can keep track of which volumes have been requested by cluster nodes. By giving each volume a unique identifier that is hard to guess, we can add limited security to the system by forcing connecting clients to provide the identifier for the volume they wish to provide. In Kubernetes, volumes have unique identifiers that are hard to guess and only accessible by someone with access to the cluster itself. Using these volume identifiers as shared secrets should provide adequate protection, since an attacker with access to them most likely have direct access to the target pods anyway.

4.2.1 Protocol

The networking protocol largely mimics the FUSE API, since its purpose is to forward FUSE operations to the user machine. Communication is based on gRPC, and the messages are defined using Protocol Buffers. This simplifies the process of writing new client applications in different languages.

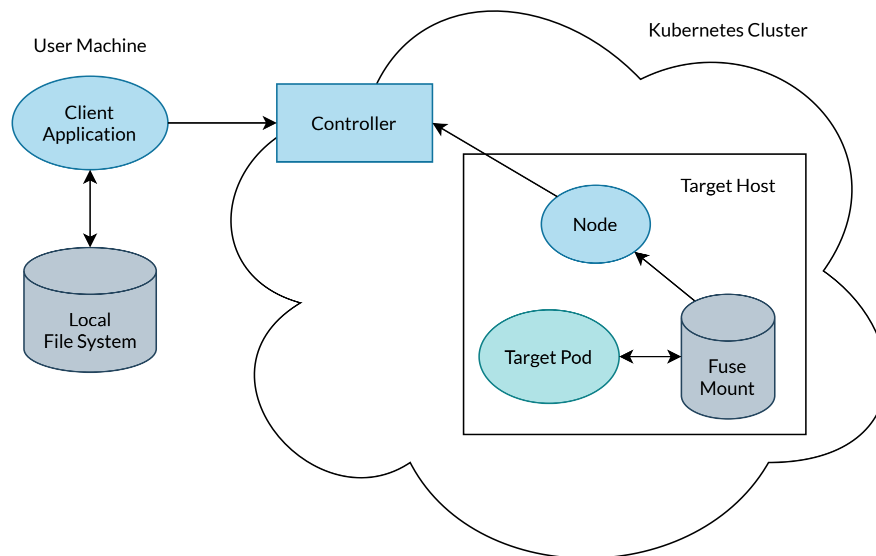


Figure 4.2: An overview of the file system components.

The most important file system messages:

- `LOOKUP`: Resolves a path to an inode. Returns information similar to what is returned by the `fstat` syscall, such as its inode number, file size, permissions, owner, time of creation and time of last modification.
- `CREATE`: Create a new file with the given file modes (such as `O_APPEND` or `O_TRUNCATE`). Returns a handle to the created file.
- `OPEN`: Open a file or directory for reading. Returns a file handle.
- `RELEASE`: Close an open file handle.
- `UNLINK`: Delete a file by name.
- `GETATTR`: Get file attributes. Returns information similar to an `fstat` syscall.
- `SETATTR`: Set file attributes such as permissions, owner or the time of last modification.
- `RENAME`: Rename a file or directory.
- `READ`: Read data from an open file handle. Returns an array of bytes containing the data.
- `WRITE`: Writes an array of bytes to a previously opened file handle at a given offset. Returns the number of bytes written.
- `READDIR`: List all files in a directory. Returns names and inode numbers.
- `MKDIR`: Create a new directory at a given path.

- `RMDIR`: Remove a directory by name.

In addition to the file system messages, there are also a number of messages related to keeping track of the clients:

- `HANDSHAKE`: Sent by the client upon connection. Provides the unique identifier of the volume the client wishes to serve.
- `PING/PONG`: Periodic health check messages to monitor the connection.

4.3 Client Application

A client application is required to actually carry out filesystem operations on the user's computer. The client application connects to the gateway and signals that it is ready to provide a certain volume by providing the unique volume identifier. If the identifier is recognized by the gateway, the client is accepted and it begins to serve requests to read and write files and directories. If the identifier is invalid, the connection is dropped.

4.3.1 Permissions

A common problem in networked filesystems is how to reconcile users between the client and server machines, since the set of users on both machines are not necessarily the same. The design presented in this thesis solves this problem by offering configuration settings that allow the user to configure an owner user ID and group ID (root by default) at startup. These values will be presented to the target machine when requesting file metadata through `GETATTR` or `READDIR`. Files and directories created on the target machine will be assigned to the owner of the filesystem client process running on the users machine. Operations that modify file permissions will be ignored by the client. File permissions always appear as 775 to the target machine.

Hard- or symbolic links are not supported by the filesystem to avoid any risk of accidentally allowing access to files residing outside the directory exposed by the user.

4.3.2 Cross-platform Compatibility

In order for the solution to be truly useful, it has to work on the most common platforms used by developers. Initially, the authors hoped that filesystem related system calls would be consistent across POSIX-compliant systems like Linux and Mac OS (Darwin). In the end, this turned out to be almost true, and with some small fixes, Mac OS compatibility was achieved. Windows is supported through the Windows Subsystem for Linux (WSL).

Several filesystem operations include a set of mode flags. While the system calls appear similar between Linux and Darwin, there are some inconsistencies among the flags, and thus they need to be translated if the client and server run different kernels.

In addition to the flags, file information structures returned by the `stat` family of system calls also have slight differences in the way they store timestamps, so they are transferred over the network in a platform independent representation. `stat` is used to implement several of the FUSE operations such as `LOOKUP`, `GETATTR`, `SETATTR` and `READDIR`.

The design assumes that the filesystem target will be running Linux, so the compatibility layer has been implemented in the client application. File mode flags are translated upon receiving related RPC calls, and a cross-platform abstraction of `stat` is used to return a Linux-compatible structure from any system.

4.4 Kubernetes Integration

Another advantage of the intermediary design is that it's similar to what is expected from a custom Kubernetes volume driver that implements the Container Storage Interface (CSI). CSI is what will allow us to integrate our file system into Kubernetes, and have the cluster orchestrator do much of the heavy lifting for us. By basing our file sharing solution on CSI, it becomes a stand-alone Kubernetes component that can be used by any Kubernetes pod. The driver can be used in any cluster and as a part of any kind of deployment, potentially providing utility in other situations where it could be useful to access local files on a remote machine.

CSI implementations, usually referred to as *drivers*, consist of two main parts - a single controller service, and a node service that runs on each machine in the cluster. It is the responsibility of the node service to handle volume mounts on the cluster nodes, and attach or detach them to target pods.

In our case, the node component of the CSI driver also connects directly to the controller, and when the node is asked by the cluster orchestrator to mount a certain volume, it requests this volume from the controller. The client program also connect directly to the controller and advertises itself as the provider of a certain volume. If a node has previously requested a volume with a matching identifier, a connection is established between the two. Otherwise, the client connection is dropped.

4.4.1 CSI Node

The node software runs on each cluster node. Its purpose is to manage mount points on the host, and expose them to the correct pods. When a pod is created with a CSI-based volume attached, the cluster orchestrator asks the corresponding CSI driver node on the pod's host machine to mount it into the target container. At this point, our node implementation mounts our custom FUSE filesystem at the requested mount point. The file system application connects to the controller, notifying it that a volume has been requested.

4.4.2 CSI Controller

The controller acts as the central coordinating service. According to the CSI specification, a controller is responsible for managing the volumes themselves. CSI is intended to allow Kubernetes to be extended with custom volume drivers, so in a more typical driver, the controller might be responsible for provisioning and attaching disks to the virtual machines which make up the cluster. In our case, there are no physical disks to manage, so the controller serves a slightly different purpose. Instead, it runs a server that accepts incoming connections from both file system clients and cluster nodes.

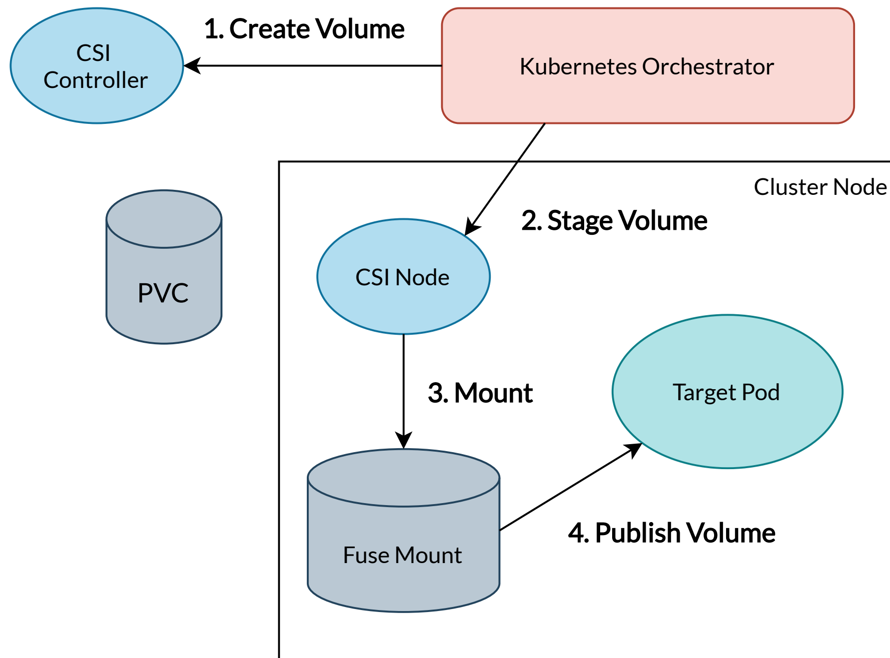


Figure 4.3: Illustration of the volume mount process. When a new *PersistentVolumeClaim* is created, the orchestrator executes the following steps. 1) *CreateVolume* is called on the CSI controller. 2) *StageVolume* is called on the CSI node on the target host. 3) Node creates a volume mount on the host. 4) *PublishVolume* is called, the volume is bind mounted into the target pod.

Clients must have a single endpoint to connect to that is accessible from the public internet. Since we can not assume that each cluster node has a reachable public address, we need an additional service to provide the single endpoint and then proxy requests to the correct cluster node.

4.5 Cowait Notebook Integration

For the purposes of evaluation, the finished file system was integrated into Cowait Notebooks. Several steps were added to the notebook launcher:

1. A Kubernetes volume is defined, with our custom filesystem as the storage driver.
2. Once the volume is created, its unique identifier is kept.
3. The file system client application is launched in the background, passing the volume identifier. It is set up to share the current project directory with the notebook.
4. Cowait creates the notebook pod. When the volume becomes available, the notebook starts up.

Then, the user can interact with the notebook as usual, but the working directory of the notebook is now linked with the working directory on the user's local machine.

Once the work is done, the user exists the Cowait notebook by pressing *Ctrl+C* in the terminal. This triggers the tear down process:

1. Notebook pod is deleted.
2. Client application process is killed.
3. The Kubernetes Volume is deleted.

The necessary changes were implemented by the authors and contributed to the Cowait open source project.

Chapter 5
Evaluation

The evaluation seeks to answer the research questions below:

- RQ1. Could a custom Kubernetes volume driver facilitate file sharing between cluster nodes and developer machines?
- RQ2. Would the performance of such a system be acceptable? Which factors impact the performance of such a solution?
- RQ3. Does a shared filesystem improve developer experience when working in cloud notebooks?

While RQ1 has been largely answered by the outcome of the implementation of the filesystem, it remains to be seen if the resulting system works well enough for real world use. RQ2 and RQ3 are intended to evaluate whether the solution is useful in practice. The evaluation has been divided into two parts: one quantitative part which seeks to answer RQ2 using various performance metrics, and one qualitative part which seeks to answer RQ3 using a usability study in which a sample of relevant developers interact with the system and present their feedback.

5.1 Performance Evaluation

To evaluate the performance of the file system an experimental methodology was developed. The methodology was inspired by the process described by Wohlin et al. [19] and can be seen in figure 5.1. In the following sections the scoping, planning and operation stages are covered, followed by the results of the experiment. The discussion can be found in chapter 6.

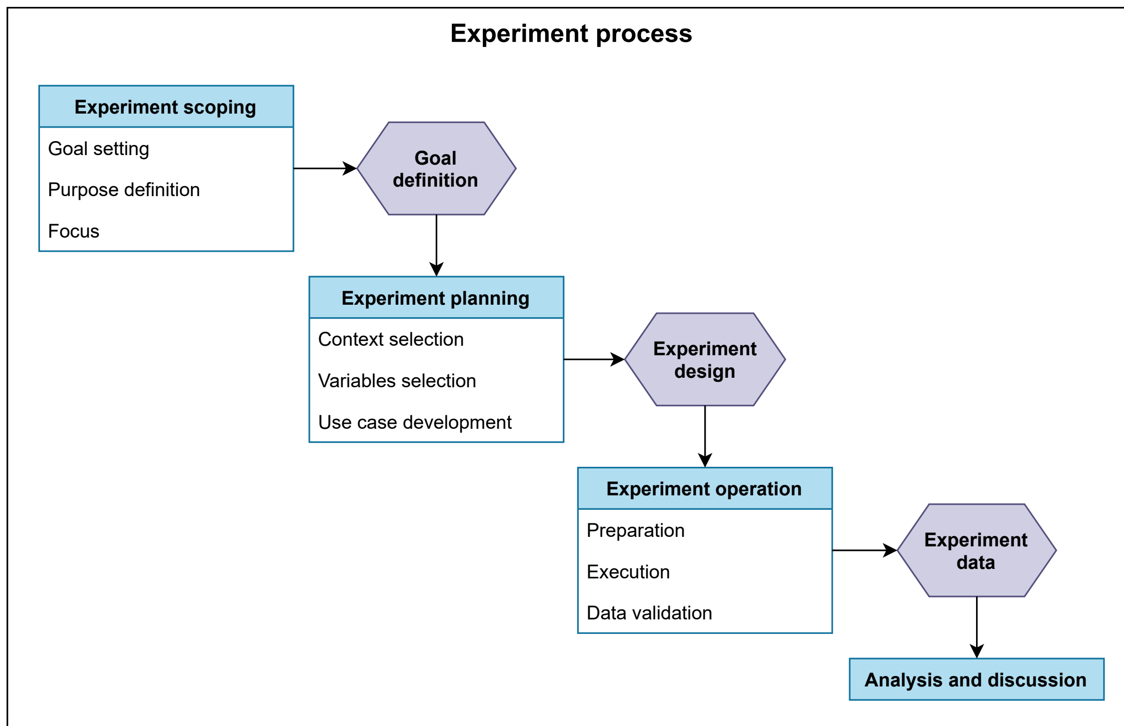


Figure 5.1: Experiment process for the performance evaluation.

5.1.1 Experiment scoping

The goal of the experiment was to verify that the shared file system presented in this report has acceptable performance in a set of realistic cases. This was done by measuring the response time[1] when performing common file system operations. These operations were performed by a program running on a Kubernetes cluster.

5.1.2 Experiment planning

Context selection

The experiment was conducted on three different Kubernetes clusters, which are listed in table 5.1. Each cluster was configured to support the shared file system, as well as Cowait notebooks. All measurements were conducted from a Cowait notebook executing a Python script, where time was measured using the `time.time()` function and file system operations were performed using the `os.system()` function. Since the Cowait notebook is designed to utilize a shared file system, the script could use that file system as the measurement target.

The latency for each cluster can be seen in table 5.1, and was measured by running the shell command `ping` ten times on the same day for each of the clusters.

[1] *response time* refers to the time elapsed between the start and finish of a file system operation

Cluster location	Network latency (ms)
Lund (LAN)	3.1 ± 2.6
Amsterdam	23.8 ± 2.4
Singapore	296 ± 35

Table 5.1: Clusters used for the performance evaluation, along with the approximate round-trip network latency. For the latency the mean and standard deviation are given.

Variable selection

When the experiment was performed there were three independent variables in focus: the network latency[2], the file size and the number of files in a directory. Network latency was chosen since the shared file system operates over the network. File size was chosen since reading and writing files takes more time for larger files. Finally, the number of files in a directory was chosen since it takes more time to list the files in a directory if it contains more files. The independent variables are listed again below for brevity:

1. File size
2. Number of files in a directory
3. Network latency to the cluster

The experiment used a single directory for the measurements. This directory contained one or more files and the files could have different sizes. The measurements were conducted by performing certain file system operations on these files.

Defining use cases

To investigate the performance of the system four different use cases were defined. They were chosen to give a clear view of the overall file system performance without having to measure every possible file system operation. The use cases were based on the following file system operations:

1. Transferring files:
   (a) Reading from a file
   (b) Writing to a file
2. Listing the files in a directory:
   (a) Regular listing, only returning file names
   (b) Long listing, returning file names and metadata

[2] *network latency* refers to the round-trip time to the cluster, commonly known as the *ping* time

These file system operations formed the basis of the four use cases seen in table 5.2. In the table there are two columns describing the directory used for the measurements, namely the number of files in the directory and the size of these files. The *variations* column shows how many points within the range were used for the measurements. In this case the points were chosen as the powers of two.

Case	Operation	Number of files	File size (kB)	Variations
1	Reading from a file	1	1 → 65536	17
2	Writing to a file	1	1 → 65536	17
3	Regular listing	1 → 256	0.1	9
4	Long listing	1 → 256	0.1	9

Table 5.2: Use case specifications.

All four use cases were performed by measuring the time needed to execute a shell command, and these commands can be seen in table 5.3. In the table the `dir` directory is located in the shared file system, and is the one used for the measurements. The `/tmp` directory is located outside the shared file system, and `file` is the name of the file being transferred.

Case	Operation	Command
1	Reading from a file	`cp dir/file /tmp`
2	Writing to a file	`cp /tmp/file dir`
3	Regular listing	`ls dir`
4	Long listing	`ls -l dir`

Table 5.3: The command associated with each use case.

5.1.3 Experiment operation

In order to perform the experiment some preparations had to be made. The clusters (listed in table 5.1) had to be configured to support the shared file system. This was achieved by installing the CSI volume driver from chapter 4. Since the experiment was conducted through a Cowait notebook, the clusters were also equipped with a Traefik reverse proxy [17].

After the preparations the experiment could be executed on the three clusters. Measurements were made for the different directory setups, as described in table 5.2, and for each directory setup the experiment was performed 16 times. The order of these runs was randomized in order to reduce the risk for biases. All measurements were orchestrated from the same computer on the same network within two consecutive days.

Once the experiments were completed the data was analyzed and validated. The data validation was performed to ensure that the data was correct, and that the measurements were consistent over time. The results of the analysis are presented in the sections below.

5.1.4 Experiment results

Transferring files

The time needed to transfer a file can be seen in figure 5.2, where both read and write times are shown for all three clusters. It is clear that file transfer takes the most time for the Singapore cluster and the least time for the LAN cluster. This is not surprising, considering the network latencies listed in table 5.1. The figure also shows that writing files takes more time than reading a file.

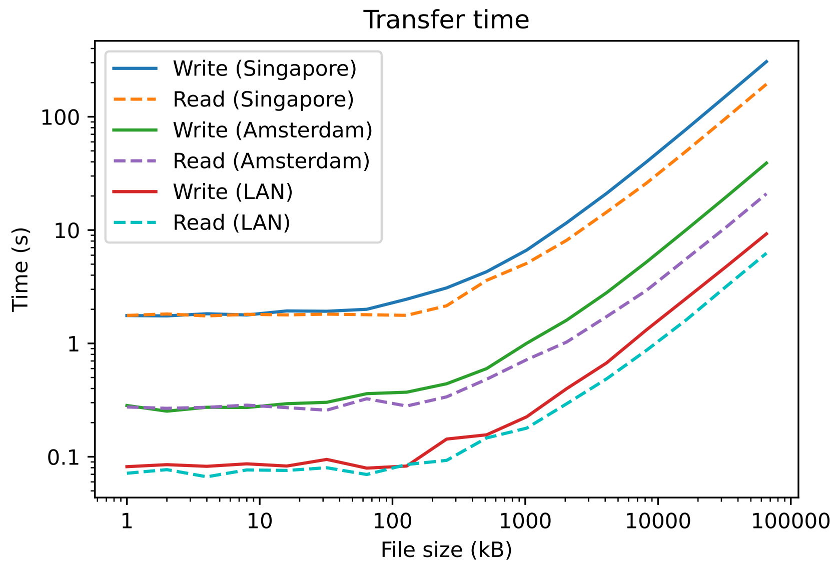


Figure 5.2: Time needed to transfer files of different sizes through the file system. Read and write times are shown for the different clusters.

From the transfer times it is possible to calculate the transfer speed, which can be seen in figure 5.3. The transfer speed is lowest for the Singapore cluster and highest for the LAN cluster. For files larger than a megabyte the speed is close to constant, and the read speeds are higher than the write speeds.

Listing files

The time needed to list the files in a directory can be seen in figure 5.4. Similarly to the results above, the Singapore cluster takes the most time and the LAN cluster is the fastest. There is also a difference between the regular and long listing times, with the long listings taking slightly more time than the regular listings. For the Amsterdam cluster there is a sudden division between the listing types. The origin of this phenomenon has not been further investigated, but it could be related to the small increase the LAN cluster is showing for 256 files.

From the listing times it is possible to calculate the listing speeds. These can be seen in figure 5.5, where it is clear that the listing speed is the lowest for the Singapore cluster and the highest for the LAN cluster.

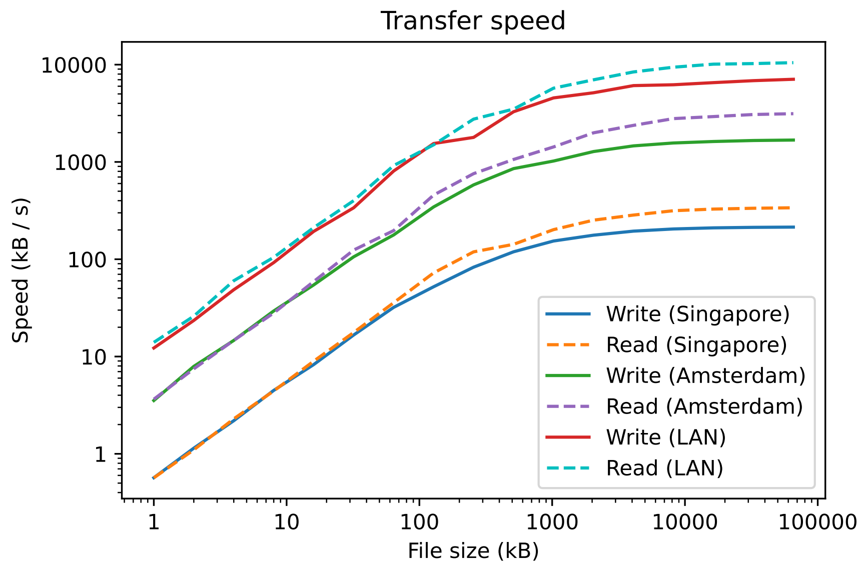


Figure 5.3: Transfer speed when transferring files of different sizes through the file system. Read and write speeds are shown for the different clusters.

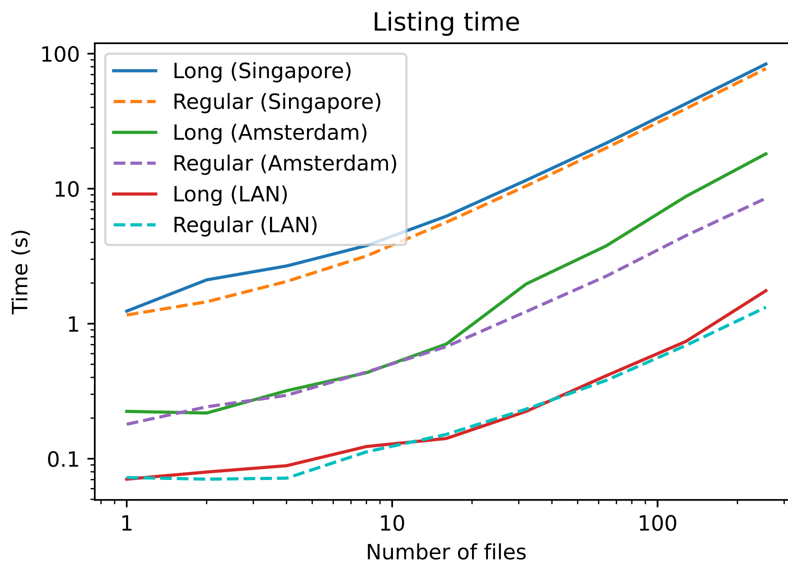


Figure 5.4: Time needed to list the files in a directory as a function of the number of files in the directory. Both regular and long listing times are displayed for the different clusters.

5.1.5 Analysis

The performance evaluation was performed to find the situations where the performance of the shared file system was acceptable. Thus, there needs to be a definition of what counts as *acceptable* performance. We define the shared file system to have acceptable performance if

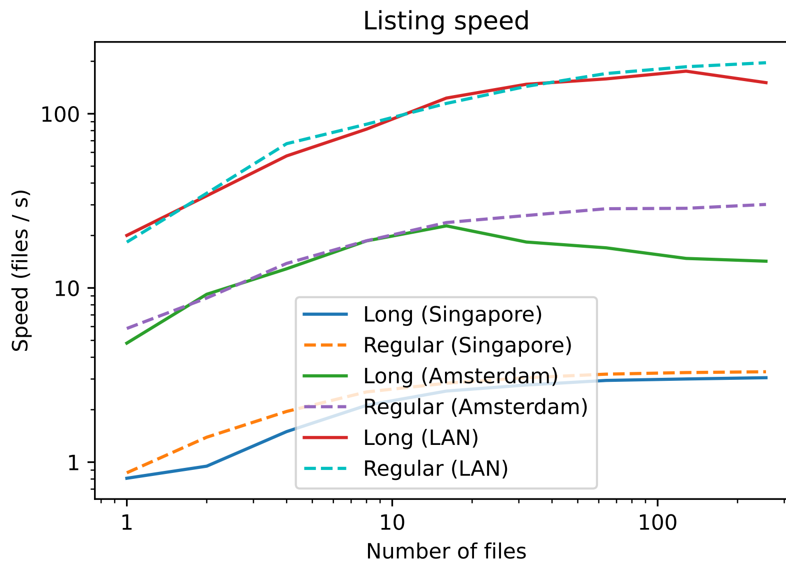


Figure 5.5: Speed achieved when listing the files in a directory as a function of the number of files in the directory. Both regular and long listing speeds are displayed for the different clusters.

it takes less than one second to list the files in a small directory and less than one second to transfer a small file. A small directory is a directory containing only a few files.

The choice of one second as the time limit was based on a book by Nielsen [12, p. 135] where three limits were identified for the response time of a user interface. These limits were 0.1 seconds, 1 second and 10 seconds and they were based on human perceptual abilities. The 1 second limit is the most relevant in this case since it marks the limit of the user's flow of thought. If an action is expected to be fast but takes more than one second to complete, the system will be perceived as slow and the user experience will be reduced.

To find the situations where the shared file system has acceptable performance, two tables have been created. In table 5.4 the time limit has been applied to file transfers, with the table showing the size of the largest file possible to transfer within the time limit. Similarly, table 5.5 shows the largest directory that can be listed within the time limit. The numbers in these tables were found by looking in figure 5.2 and 5.4.

Time (s)	**File size (MB)**					
	Singapore		Amsterdam		LAN	
	Write	Read	Write	Read	Write	Read
1	-	-	1	2	6	10
3	0.25	0.4	4	8	20	30
10	1.8	2.5	15	30	70	*100*

Table 5.4: Maximum size of a file transferred in a certain amount of time. Both reading and writing times are shown for the different clusters. Numbers in *italics* are extrapolations.

Time (s)	Number of files					
	Singapore		Amsterdam		LAN	
	Long	Regular	Long	Regular	Long	Regular
1	-	-	20	25	140	180
3	5	7	50	80	*400*	*700*

Table 5.5: Number of files listed in a certain amount of time. Both long and regular listing times are shown for the different clusters. Numbers in *italics* are extrapolations.

In table 5.4 it is clear that the Singapore cluster does not provide acceptable performance, since no file transfers complete within one second. A similar situation can be seen in table 5.5. Even if the user waits three seconds the performance is very low. The Amsterdam cluster and the LAN cluster do however appear to provide acceptable performance.

It should be noted that if the user is using e.g. a notebook to interact with the server the user will experience an extra delay equal to the round-trip latency to the cluster. This is because the requested operation needs to be sent to the cluster, and the result needs to be sent back after the completion of the operation. This discrepancy is however negligible in most cases, especially when the operation itself takes over a second to perform. For this reason the numbers in table 5.4 and 5.5 do not include the extra round-trip time. If they did, the numbers would be slightly lower for the Singapore cluster.

In conclusion, the performance was acceptable for both the LAN cluster and the cluster in Amsterdam for small files in small quantities, but due to the high latency the performance was not acceptable on the cluster in Singapore.

5.2 Usability Evaluation

To evaluate the usability of the system a study was performed according to the process in figure 5.6. In the following sections the three parts of the process are covered, followed by a section presenting the results of the study.

5.2.1 Study Design

The usability evaluation was primarily designed to answer RQ3 (*Can a shared filesystem improve the developer experience when working in cloud notebooks?*) and to ensure that the performance of the system is indeed adequate for real-world use by actual developers.

A group of developers with relevant work experience was recruited to carry out an evaluation exercise and an accompanying interview. The exercise was designed to demonstrate the capabilities of the filesystem implementation in a real world setting.

In the following sections the interview guide and sampling strategy are covered, followed by a few sections about the exercise.

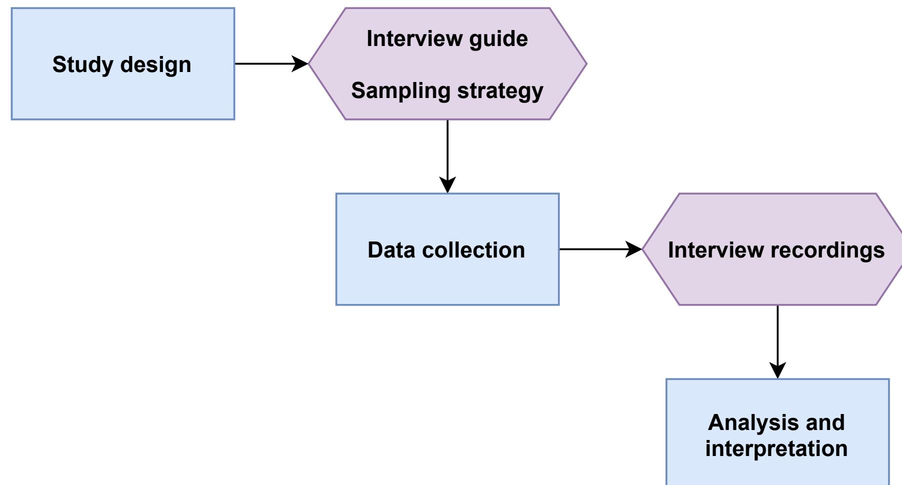


Figure 5.6: Usability evaluation process

Interview Guide

The interviews were semi-structured using the following organisation:

1. The interviewer briefly introduced the subject to Cowait notebooks and the shared file system
2. The subject performed the exercise, and could ask questions and submit feedback to the interviewer in a think-aloud manner [10]
3. The interviewer asked the subject three questions:
    (a) What is your overall impression of working with Cowait notebooks?
    (b) Do you think cloud notebooks with a shared file system could help improve the data science workflow in your organization?
    (c) Do you see any other advantages/disadvantages in having access to your local file system when working in a cloud notebook?

Sampling Strategy

A total of ten subjects from seven different companies were chosen to perform the evaluation exercise and the accompanying interview. They were sampled using *convenience sampling*, which in this case means that they were known by the authors or their colleagues. Demographic information about these subjects can be seen in table 5.6 including approximate age, experience using cloud notebooks, role in the company and company size.

The opinions and feedback given by the subjects are their own and does not necessarily reflect the official stance of the company they work for.

Evaluation Exercise

The exercise involved working with a Kubernetes cluster. For this purpose the interview subjects received access to a cluster located in Amsterdam, The Netherlands. The location

ID	Experience	Role	Age	Sex	Company	Domain
A	High	Data Scientist	20 - 29	M	Large	Hardware
B	High	Data Scientist	20 - 29	M	Large	Retail
C	High	Data Scientist	20 - 29	M	Small	Consultant
D	Mid	Software Engineer	20 - 29	M	Small	Consultant
E	Mid	Software Engineer	20 - 29	M	Small	Consultant
F	Mid	Data Scientist	20 - 29	M	Medium	IT Security
G	Mid	Data Scientist	30 - 39	M	Small	Consultant
H	Mid	Data Scientist	20 - 29	M	Large	Retail
I	Mid	Software Engineer	40 - 49	M	Medium	IT Security
J	Low	Data Scientist	20 - 29	M	Small	Consultant
K	Low	Software Engineer	20 - 29	M	Large	Hardware

Table 5.6: Interview subject demographics. Sorted by self-reported notebook experience.

was chosen to be reasonably far away so that the network latency would be realistic. The cluster was prepared with an installation of the custom CSI driver, as well as a Traefik reverse proxy [17] necessary for Cowait Notebooks.

Before starting out, subjects were given a brief introduction to the thesis project and an outline of the steps and goal of the exercise. The expected time to finish the exercise and interview was around one hour.

The subjects were asked to set up a Cowait notebook with a shared filesystem client on their computer, and then follow a series of steps that guided them through a toy problem set up to simulate a typical problem that a user might solve using a cloud notebook. The exercise was divided into three parts, each serving a different demonstration purpose.

The full exercise is available in Appendix A.

Assumptions

The following assumptions were made when designing the exercise:

- The filesystem is primarily intended for sharing source code, not datasets. Source code is assumed to be distributed in relatively small files of less than 100 kB. The evaluation does not involve transfers of any large files. Studying the behaviour of the system when dealing with large files has already been covered by the performance evaluation.

- Performance evaluations have shown that the filesystem in its current state is not suitable for working with folders with large amounts of files. It is assumed that this problem would be largely solved by a simple metadata caching scheme (elaborated on in section 6.2), and thus the evaluation does not involve any such folders.

The Exercise Parts

In the first part, the subject was asked to start up a notebook and write code to download data from the internet and perform a simple computation. Then, the subject had to write a

test for the notebook code and run it on the local machine instead of the cloud notebook. After the test passed, the subject used their local git client to commit the work completed so far. The last two steps were intended to showcase the fact that the shared filesystem did indeed enable the use of local tooling even when working in a cloud notebook.

The goal of the second part was to create a new notebook that runs code from the first part in parallel. Running multiple notebooks concurrently allows us to gather a larger dataset, while demonstrating some of the other benefits of Cowait notebooks. The section ended with the user writing the result of the parallel computations to disk. Because of the shared filesystem, the results file was actually stored on the subject's computer. Part two demonstrated that the shared filesystem made it possible to use a cluster to perform computations on large datasets while simultaneously being able to read and write smaller files on the developer machine.

The third and final part revolved around executing the notebook code from the command line as a Cowait job instead. When leaving the notebook behind, there was no longer a shared filesystem with the cluster. Thus, a Docker image had to be rebuilt with the latest version of the code, and in the end, the written result file was actually lost when the container exited. This part served to remind the subject of what it was like to work without real-time file sharing.

5.2.2 Data Collection

During the interviews both authors of this report were present, with one acting as the driver and the other ensuring that the interview guide was followed. Some interview subjects performed the interview individually. In total there were 11 interview sessions.

The interviews were performed in Swedish using the Zoom video conference software, and the interviews were recorded with the consent of the interview subjects. These recordings were collected for the interviewers to know what was said during the interview without having to take extensive notes at a high speed. There were however some notes taken during the exercise to capture any insights acquired from the think-aloud protocol.

5.2.3 Analysis and interpretation

The data analysis consisted of listening to the interview recordings, collecting the answers to each question, and then combining the answers from all subjects. These answers could then be used to perform a synthesis, where both common answers and alternative view points could be identified. These interpretations are presented in the next section, along with insights acquired from the think-aloud protocol used during the exercise.

5.2.4 Interview Feedback

During the exercise, the subjects could ask questions and were encouraged to provide feedback in a think-aloud manner. Once the exercise was completed, a short interview was conducted. The gathered feedback has been divided into 4 categories.

Overall Impression

Overall, the interviewees found the Cowait notebook easy to use. The filesystem connection was automatically set up in the background without any problems in each session.

Once the subjects realized that they could really access files on their own computers from the cloud notebook, most subjects were curious about how the shared filesystem worked and how it was set up. While the reason for the question was often curiosity about the technical details, the question itself indicates that the process of setting up the shared filesystem was transparent to the end user.

While most interviewees found the experience of using Cowait Notebooks with a shared filesystem enjoyable, the interest in actually using it was somewhat disappointing. The data scientists who were the most used to working in cloud notebooks were already very used to their workflows, and hesitant to introduce such a radical change. Setting up a completely new working environment to integrate the shared file system was perceived to be more trouble than it's worth.

Almost half of the interviewees showed different degrees of worry over the stability of the network connection, and brought up concerns over what would happen to their work in case of a lost connection. Respondents agreed that this could have been largely mitigated by receiving a more thorough explanation of the behaviour of the file system prior to performing the exercise.

In four of the interviews, the interviewees noted that the system feels like too much "magic" and will need to be very well documented in order to give the user a solid enough understanding that they would feel confident using it.

Advantages

In general, most of the interview subjects found it advantageous to be able to seamlessly share files with the cluster. The primary reason given was that it would be great to avoid having to manually upload and download files to the cloud notebook, but not everyone agreed. Some of the interviewees stated that any advantage in this regard would be outweighed by the advantages of having a completely managed cloud notebook system.

The ability to use local software for remote development was commonly brought up as an advantage, however, not everyone agreed that this is a problem that they currently experience. It was also noted that this benefit may be less relevant to users without prior development experience who are not used to these tools.

One interviewee was excited about the idea of avoiding the complicated SSH environments they were currently using in order to work on shared compute resources. Employees would access powerful machines on the local network by a combination of Kubernetes port forwarding and SSH access. Problems included each user having to set up an entire development environment, including credentials for tools like git.

Several subjects mentioned the possibility that avoiding file upload to cloud machines could potentially be a benefit when dealing with highly confidential data, since the information in question never actually resides on a physical disk outside of the user's control.

Disadvantages

A common concern that was raised in multiple interviews is that of security, both concerning the security of file transfers but most importantly the feeling of having their computer "exposed" to the outside world. While running unknown software is common, the nature of a situation where a remote machine has such obvious access to the users local hard drive does raise concerns. It is perhaps worth noting that this feeling would most likely be mitigated by the user putting themselves in the situation out of their own needs, rather than being unwillingly put in it by the evaluation exercise.

Another disadvantage that was mentioned by multiple subjects who primarily do a lot of work in managed cloud notebooks in larger organizations, was the difficulty of sharing or collaborating with other people who are not necessarily developers. Managed cloud notebooks allow users to easily share links to hosted notebooks, something that is considerably more difficult in the evaluation setup. While the notebook URL can be shared, the shared filesystem notebook requires the author to be online and actively serving the files.

Regarding the issue of handling confidential data, not everyone agreed that local storage was preferable. Some of the participants instead preferred storing the confidential data on cloud servers. The reasoning being that an employee computer is inherently less secure than a server located in a guarded data center. Some employers even prohibit data from being stored outside of their cloud environments.

Performance

None of the interview subjects reported any discomfort related to the performance of the file system. When asked, users reported that the performance seemed to be adequate for the demonstrated use case. A few subjects did, however, raise concerns that it might be easy to accidentally use the system in other ways than intended, which could negatively affect the performance. An example would be using the filesystem to share large data files with the cloud server, which would be slower than if the notebook was running locally.

Chapter 6

Discussion

This chapter attempts to answer the research questions based on the implementation outcome and the results of the two evaluation studies. The implementation of the proof-of-concept system is discussed in section 6.1. Results of the performance evaluation is analyzed in section 6.2. Interpretation of the interview results are presented in section 6.3.

6.1 Implementation (RQ1)

The proposed solution design was turned into a working proof-of-concept implementation that fulfils all of the initial requirements. Users can easily share a sub-tree of their local file system with an application running on a remote cluster. The user's files appear to the remote application as regular files and directories, further ensuring compatibility with any software that operates on a standard file system. It is implemented as a custom volume driver, which decouples the implementation from a specific application and provides compatibility with any Kubernetes workload. Building on Kubernetes, the most popular cluster orchestrator and quite possibly the standard way of deploying software going forward, ensures that the file sharing solution can be used with virtually any application.

By introducing a central gateway service, cluster nodes and user machines have a well known endpoint to communicate through. This allows users to connect and provide volumes without exposing any ports to the public internet, greatly simplifying the set up on the user's end. However, in the current design, the gateway service is a single point of failure. If it crashes, all active file sharing sessions will be lost, with no option but to restart the applications that were using them. The gateway service is also a potential performance bottleneck, since all file system traffic flows through it. While this is a problem in the proof-of-concept system, there are multiple ways to go about solving it. A better design would be to separate the gateway from the CSI controller, which would open up the possibility to independently replicate and balance load over multiple gateway services.

Overall, the design of the system has led to a fairly simple implementation, and while it

could use improvements in several areas, it works surprisingly well in its current form. With some additional work and testing, it could become a useful way to interact with cloud servers in the future.

Modularity

While the file sharing solution is not tied to any specific application, the implementation of the FUSE filesystem is currently intertwined with the CSI volume driver, specifically the CSI node. Besides limited modularity, this design has a few drawbacks.

Firstly, the FUSE file system runs in the same process as the CSI Node. Because of this, the entire CSI node (which could be serving multiple pods simultaneously) could be brought down by an unrecoverable error in any of the running file system mounts. If FUSE ran in a separate process for each mount, the CSI node would be isolated from any crashes.

Secondly, there is no way to run the file system outside of Kubernetes, which makes it much harder to properly test the individual components. Thus, separating the FUSE file system to a separate process would bring multiple benefits.

Thirdly, the CSI Controller service is sharing a process with the gateway service, causing similar interdependence problems. A crash in the CSI controller will bring down all active file system sessions, and vice versa.

All of these problems can be quite easily mitigated by separating the FUSE file system implementation into its own separate process.

Communication Security

One of the most serious problems with the file system is that the network traffic to and from the gateway is currently not encrypted. The system is therefore vulnerable to eavesdropping and man-in-the-middle attacks. Since the gRPC implementation used for network communication has built-in support for Transport Layer Security (TLS), this is mostly a matter of configuration. However, the additional complexity of handling certificates was considered outside the scope of this thesis.

Another issue is that anyone with access to the unique volume identifier can connect to the target pod and provide the volume. If the filesystem were to be deployed as part of a real software-as-a-service solution, it would need an improved authentication system. The gateway service would be a natural place to implement user authentication, considering that it already acts as a central coordinator for the system.

6.2 Performance (RQ2)

The goal of the performance evaluation was to verify that the performance of the shared file system is acceptable. Three clusters were used for the evaluation, and the clusters were located in different parts of the world in order to cover a wide range of network latencies. During the evaluation both file transfers and directory listings were tested.

The performance of the file system can be analyzed by considering the following setups:

1. Small number of small files

2. Small number of large files
3. Large number of small files

In the first case the file system only contains a small number of files and the files themselves are small in size. This case was analyzed in section 5.1.5 where a file system was deemed to have acceptable performance if both file transfers and directory listings could be finished in less than one second. The result was that the cluster in Singapore (with a latency of around 300 ms) was too slow but the other clusters were fast enough. This shows that as long as the cluster is not too far from the user the performance can be acceptable.

In the second case the files are large but few. Here the file size is in focus, and especially the transfer speed. Figure 5.3 shows that the transfer speed when reading or writing large files levelled out for files larger than 1 MB, but the speed was different between the clusters. The transfer speeds ranged between approximately 0.1 and 10 MB/s, but these differences could likely be reduced with some future work on the file system implementation.

In the third case the files are small but numerous, which means that there is a directory which contains a large number of files. Here the directory listings are the most relevant, and in figure 5.4 the time needed to list the files in a directory is shown. If a directory contains for instance 200 files, the time needed to list the directory contents is over a minute for the cluster in Singapore. This is clearly not acceptable performance, but luckily there are several possible improvements that can be made. These include *caching* and *prefetching* of metadata, which are explained in the sections below.

Caching

The empirical evaluation of the filesystem demonstrated that a lot of time was spent on listing directories, fetching file attributes, opening and closing handles and other operations which are not directly related to reading and writing data.

These operations take very little time to execute on the client side, but are typically run sequentially, and each operation incurs the cost of a network round trip. For large directories, these round trips add up quickly, especially in high latency environments. [18]

However, since the file system is designed for two machines operated by a single user, conflicting operations are unlikely to occur simultaneously on both sides. This means that it should be possible to cache most of the results of the metadata operations on the server side. The client application could subscribe to filesystem notifications and send messages to invalidate the cache as necessary.

By replacing network calls with memory look ups one can expect a performance improvement of several orders of magnitude for the most common operations, such as listing the contents of a directory.

Prefetching

Caching can only ever hope to increase the performance of repeated requests, but the user will still experience long delays initially. According to the performance evaluation, large directories with many files have the most negative impact on user experience.

The only way to reduce this initial latency is to pre-fetch data into the cache before the user needs it. The client could start streaming file metadata in the background immediately upon an established connection.

Another approach is to predict future file system operations and prefetch the data needed for those operations. For instance, if the user lists the files in a directory, the operating system might perform a `READDIR` operation followed by a `LOOKUP` operation for each file in the directory. When the `READDIR` operation has been received the `LOOKUP` information can be prefetched into the cache. This would greatly reduce the time needed for listing directories. This approach has been used before, for instance by the Ceph filesystem [18].

6.3 Interview Feedback (RQ3)

The main purpose of the interviews was to evaluate whether a shared file system could improve the overall experience of working with cloud notebooks. Participants were asked to consider the benefits of being able to use software installed on their local machines (such as version control systems) in combination with the notebook, and to think of possible drawbacks.

Respondents can be roughly divided into three groups, ordered from most common to least common:

1. **Data Scientist, primarily using cloud notebooks.** They tended to prefer the managed solutions, either due to not experiencing any difficulties that could be mitigated by the file system, or citing the importance of other advantages such as ease of collaboration with non-technical colleagues.

2. **Data Scientist, primarily using local notebooks.** These participants were usually positive towards the idea, probably because it closely resembles the environment they are used to working in. However, this group had little need for any extra resources offered by cloud notebooks.

3. **Software Engineer, sporadic use of notebooks** Although a small group, they were the most positive towards the shared file system. This was likely due to them being the most experienced in working with more traditional development tools.

Out of all the participants, the group that was the most positive to the presented solution was the ones who were primarily software engineers but who occasionally work in notebooks (group 3). Perhaps this is an unsurprising result, considering that the idea for the thesis itself originates from people similar to this group, and that the solution aims to bring the data science workflow closer to that of traditional software development. Unfortunately, this is also the smallest group, which indicates that the solution might not be well received by its intended audience.

Group 2, data scientists primarily working in local notebooks, were generally positive to the idea of having a shared file system with a cloud server. The main reason given was that it closely resembles their current workflow, which leads them to believe that a transition to a cloud solution based on a shared file system would be easy. This group currently has little need for the extra compute resources provided by cloud servers, but they make up an important target demographic since they might need access to more powerful machines in the future.

The largest group, consisting of data scientists who primarily work in managed cloud notebook environments in larger organizations (group 1), were the most hesitant to the idea.

Participants in this group were generally happy in their current environment and the majority did not currently experience problems that could be easily mitigated by access to local tooling. It is worth noting that part of this group had spent most of their working careers in cloud-based tools and because of this, could have a more difficult time seeing the benefits from the simple demonstration presented in the evaluation exercise.

Issues related to collaboration options were brought up frequently. For example, a notebook based on a local file system is not very well suited for simultaneous use by multiple developers. Such a notebook is by definition only reachable by others while the software is running on the author's machine, which could be a big disadvantage compared to a hosted solution where any notebook can be reached via a permanent URL at any time. However, this criticism is tied to the particular notebook implementation. It is possible to envision ways in which a snapshot of the file system at a particular time could be published to a managed service where it would be available to others.

None of the participants reported any discomfort related to the performance of the file system, indicating that the system is fast enough to provide a good user experience for small scale projects such as the one demonstrated during the interview. This result is in line with the performance evaluation.

It is difficult to conclude whether or not the overall experience of working in cloud notebooks was improved based on the results of the interviews. Because of the way the evaluation exercise was set up, any feedback was heavily biased by the particular notebook implementation used, and this likely affected the results. All of the interviewees could see at least *some* advantage to having a shared file system with the notebook, so it is probably fair to conclude that offering it as a storage option could improve the notebook experience.

6.4 Threats to Validity

When performing a study there will always be threats to validity. The validity of a study can be divided into three types: internal validity, external validity and reliability.

Internal validity concerns the relation between cause and effect, and it is threatened if there are confounding factors affecting the result. In the performance evaluation the performance of the system could be affected by the environment in several ways. Firstly the network latency and bandwidth could vary over time, as well as the packet size used for the TCP connection. Secondly both the user machine and the cluster machine could be running other programs, which could affect the performance. Lastly the performance could have been affected by the measurement setup itself. The authors believe that none of these factors have enough influence to change the conclusions presented in this report, but they could certainly alter the measurements slightly.

External validity concerns the generalizability of the results, both for the performance evaluation and the usability evaluation. Since the performance evaluation only considered four of the most basic file system operations, there could be other operations with a different performance profile. Regarding the usability evaluation, the exercise was designed to be realistic, but there is still a risk that the interview subjects would act differently when working with their own data, which could change their need for a shared file system. There is also a risk that the chosen participants are not representative of the target audience, in this case data scientists. This risk was reduced by sampling participants with a wide demographic

range, but the range could certainly be wider.

The reliability of a study concerns the similarity in interpretation between the authors and other scientists. For the usability evaluation this would mean that another scientist could listen to the interviews and arrive at the same conclusions as the authors. To ensure reliability of the study both authors were present during the interviews, with the intention of reducing the risk for single researcher bias.

6.5 Future work

While the implementation presented in the thesis is a fully working proof-of-concept, there are many ways in which it could be improved. The most important enhancements have been discussed extensively in sections 6.1 and 6.2.

- Improve directory listing performance by caching file metadata.
- Improve read/write performance by caching files on the remote system.
- Implement proper transport layer security to ensure data confidentiality and integrity.
- Decouple the gateway service from the CSI controller to improve scalability.
- Decouple the FUSE filesystem implementation from the CSI node to improve testability and modularity.
- By introducing a proper way to persist the state of currently active volumes, it might be possible to recover from CSI controller crashes.

Chapter 7
Conclusions

This thesis presents a working proof-of-concept implementation of a general solution to sharing files between a user's computer and software running on a remote cloud machine. The complete solution consists of a custom networked file system based on FUSE, a client application and a gateway service. Compatibility with any Kubernetes workload was achieved by implementing the solution as a custom volume driver leveraging the Container Storage Interface, an open standard for developing new storage options for Kubernetes.

To evaluate whether the solution provides any value for real-world use cases, two separate studies were conducted. A quantitative performance study was carried out to verify that the system is fast enough to meet minimum requirements set by prior user experience research, as well as to study how the performance varies with different file sizes, number of files and network conditions. Engineers with relevant working experience were recruited and asked to perform an exercise designed to highlight the benefits of a shared file system, and their feedback was gathered to see if the file system improves the overall experience of working with cloud notebooks.

The performance evaluation showed that even a naive implementation of a networked file system, such as the one presented in the thesis, can perform well enough for streaming code between developer machines and cloud notebooks. Assuming reasonable latency, small files and small folders, response times stayed within the acceptable limit of 1 second.

Interviews showed varying interest in a cloud notebook with a shared file system. Data scientists who were comfortable using managed cloud solutions were hesitant to use such a system, citing convenience or collaboration concerns. The group that was the most positive were developers with a software engineering background, who were excited to be able to use familiar tooling for local files.

While the system performs reasonably well for small scale projects, much work remains to be done in order to achieve acceptable performance for larger code bases. On the bright side, there is a clear path towards reducing response times, greatly improving the user experience.

References

[1] Souti Chattopadhyay, Ishita Prasad, Austin Z. Henley, Anita Sarma, and Titus Barik. What's wrong with computational notebooks? pain points, needs, and design opportunities. In *Human Factors in Computing Systems*, pages 1–12, 2020.

[2] Docker. The industry-leading container runtime. https://docker.com.

[3] B. Fog and C. N. Klokmose. Mapping the landscape of literate computing. In *Proceedings of the 30th Annual Workshop of the Psychology of Programming Interest Group*, 2019.

[4] Andrew Head, Fred Hohman, Titus Barik, Steven M. Drucker, and Robert DeLine. Managing messes in computational notebooks. In *Human Factors in Computing Systems*, pages 1–12, 2019.

[5] Dr. Brijender Kahanwal. File system design approaches. *International Journal of Advances in Engineering Sciences*, 2014.

[6] Mary Beth Kery, Bonnie E. John, Patrick O'Flaherty, Amber Horvath, and Brad A. Myers. Towards effective foraging by data scientists to find past analysis choices. In *Human Factors in Computing Systems*, pages 1–13, 2019.

[7] Mary Beth Kery and Brad A. Myers. Interactions for untangling messy history in a computational notebook. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 147–155, 2018.

[8] D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[9] Kubernetes. Production-grade container orchestration. https://kubernetes.io.

[10] Hannu Kuusela and Pallab Paul. A comparison of concurrent and retrospective verbal protocol analysis. *The American Journal of Psychology*, 113(3):387–404, 2000.

[11] Sam Lau, Ian Drosos, Julia M. Markel, and Philip J. Guo. The design space of computational notebooks: An analysis of 60 systems in academia and industry. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 1–11, 2020.

[12] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers, 1993.

[13] D. M. Ritchie and K. Thompson. The unix time-sharing system. *The Bell System Technical Journal*, 57(6), 1978.

[14] Jeremy Singer. Notes on notebooks: is Jupyter the bringer of jollity? *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, 2020.

[15] Vasily Tarasov, Abhishek Gupta, K. Sourav, S. Trehan, and E. Zadok. Terra incognita: On the practicality of user-space file systems. In *HotStorage*, 2015.

[16] Anton Teslyuk, Sergey Bobkov, Alexander Belyaev, Alexander Filippov, Kirill Izotov, Ilya Lyalin, Andrey Shitov, Leonid Yasnopolsky, and Vasily Velikhov. Architecture and deployment details of scalable jupyter environment at kurchatov institute supercomputing centre. In *2020 Ivannikov Memorial Workshop*, 2020.

[17] Traefik. The cloud native application proxy. https://traefik.io/traefik.

[18] Sage Weil, Scott A Brandt, Ethan L Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In *Proceedings of the 7th Conference on Operating Systems Design and Implementation*, 2006.

[19] Claes Wohlin, Magnus C. Ohlsson, Martin Höst, Per Runeson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer Berlin Heidelberg, 2012.

Appendices

Appendix A

Evaluation Exercise

Introduction

Cowait is a system for packaging a project with its dependencies into a Docker image, which can then be run as a container either on the local machine or on a Kubernetes cluster. It alleviates several problems in data engineering such as dependency management, reproducibility, version control and parallel computation. Cowait runs code as containerized tasks, and a task can start subtasks with parameters and return values. These subtasks run in parallel as separate containers, which enables parallel computation.

A Cowait notebook is essentially a Jupyter notebook running with a Cowait kernel. This enables the notebook to act as if it was as Cowait task, which means it can start new Cowait tasks in the background. The notebook can run either locally or in a Kubernetes cluster, and the notebook works in the same way in both cases.

One of the defining differences between Cowait notebooks and other cloud notebooks is the access to the local file system. When starting a Cowait notebook from the command line it will automatically receive access to the current working directory, using a networked file system set up in the background.

In this lab you will learn how to use Cowait Notebooks by creating a simple, yet realistic, project. The notebooks will run on a Kubernetes cluster, but all the project files will reside on your computer. The lab takes around 20 minutes.

Preparations

Please complete the following steps before proceeding.

Prerequisite Software

- git
- Docker
- Python 3.7

Initial Setup

1. Install Cowait:

   ```
   $ pip3 install cowait
   ```

   If you already have Cowait installed, make sure it is at least version 0.4.23.

2. Clone the demo repository:

   ```
   $ git clone https://github.com/backtick-se/cowait-notebook-eval
   $ cd cowait-notebook-eval
   ```

Docker Registry

You will need an image registry to distribute your code to the cluster. The easiest way is to sign up for a free account on Docker Hub at https://hub.docker.com/signup

After signing up, ensure your Docker client is logged in:

```
$ docker login
```

Cluster Configuration

Participants of the evaluation study should have received a `kubeconfig.yaml` file that can be used to access the evaluation cluster. If you are not participating in the evaluation, you will have to set up your own Cowait cluster. A traefik2 reverse proxy deployment is required.

Put the provided kubeconfig file in the current working directory. Then, set the `KUBECONFIG` environment variable:

```
$ export KUBECONFIG=$(pwd)/kubeconfig.yaml
```

Lab

Part 1: Your first Notebook Task

The goal of part one is to create a notebook that computes a value we are interested in. Then, we turn the notebook into a Cowait task, so that it can be executed as a batch job.

1. Open `cowait.yml` and update the `image` setting to `<your dockerhub username>/cowait-notebook-eval`. This configures the name of the container image that will contain all our code and dependencies.

2. Create a `requirements.txt` file and add `pandas`

3. Build the container image, and push it to your registry:

```
$ cowait build --push
```

4. Launch a Cowait Notebook using your newly created image:

```
$ cowait notebook --cluster demo
```

   It might take a few minutes for the cluster to download the image. Once the task is running, a link will be displayed. Open it to access the notebook.

5. Create a new notebook called `volume.ipynb`. Make sure to select the Cowait kernel.

6. Download some data into a pandas dataframe. The dataset contains every trade executed on the Bitmex cryptocurrency derivatives platform, divided into one file per day.

```
import pandas
date = '20210101'
df = pandas.read_csv(f'https://s3-eu-west-1.amazonaws.com/
   public.bitmex.com/data/trade/{date}.csv.gz')
```

7. We want to compute the total US dollar value of Bitcoin contracts over the course of the day. Bitcoin Perpetual Futures contracts have the ticker symbol `XBTUSD`. To do this, use pandas to find all the rows containing `XBTUSD` transactions, and sum the `size` column.

```
volume = int(df[df.symbol == 'XBTUSD'].size.sum())
print(volume)
```

8. Parameterize the notebook by changing the date variable to an input parameter:

```
date = cowait.input('date', '20210101')
```

   In Cowait, *inputs* allow us to send arguments to tasks. Later, we can substitute the input value to execute the notebook code for any date we like. If no input is set, the default value `20210101` will be used.

9. Return the total volume from the notebook using `cowait.exit()`:

```
cowait.exit(volume)
```

   Similarly to *inputs*, tasks can also return *outputs*. Returning an output allows us to invoke the notebook and use the computed value elsewhere.

10. Write a simple sanity test for the notebook that verifies the computation for a date with a known volume. Create a file called `test_compute_volume.py` with your favorite text editor:

```
# test_compute_volume.py
from cowait.tasks.notebook import NotebookRunner
async def test_compute_volume():
    vol = await NotebookRunner(path='volume.ipynb', date
        ='20210101')
    assert vol == 2556420
```

The **NotebookRunner** task executes a notebook file and returns any value provided to **cowait.exit()**.

11. Open a new terminal in the same folder and run the test. Make sure it passes.

    ```
    $ cowait test
    ```

    Contrary to the notebook, the tests will run in a Docker container on your computer.

12. Now is a good time to save your progress. Since the files are available on your local machine, use your git client to create a commit.

    ```
    $ git add .
    $ git commit -m 'Volume notebook'
    ```

Part 2: Going Parallel

We now have a notebook for calculating the volume for one day. But what if we want to know the volume for several days? While we could create a loop and download each day in sequence, it would be much more efficient to do it all at once, in parallel.

1. Create a new notebook with the Cowait kernel, and call it **batch.ipynb**.

2. First, we will create two input parameters and create a range of dates that we are interested in.

    ```
    from helpers import daterange
    start = cowait.input('start', '20210101')
    end = cowait.input('end', '20210104')
    dates = [ date for date in daterange(start, end) ]
    dates
    ```

3. Then, we can create a **NotebookRunner** for each date in the list. This will start four new tasks, each calculating the volume for one day. While these are running the notebook can perform other calculations.

    ```
    subtasks = [ NotebookRunner(path='volume.ipynb', date=date) for
         date in dates ]
    ```

4. To get the results of the calculations we need to wait for each task to finish:

    ```
    # just for reference, dont try to run this
    result1 = await task1
    result2 = await task2
    ```

    Since we have a list of pending tasks, we can use **cowait.join**. Create a new cell with the following code:

```
results = await cowait.join(subtasks)
```

5. Finally let's print the results:

```
print(results)
```

6. Write the results to a JSON file on your local machine.

```
import json
with open('result.json', 'w') as f:
  json.dump(results, fp=f)
```

7. Use the `Run All Cells` feature in the `Run` menu to try out the notebook. This will run a tasks for each day in the date range, in paralell, on the cluster.

8. Now is a good time to save your progress.

```
$ git add .
$ git commit -m 'Volume batch notebook'
```

Part 3: Production

We now have a runnable notebook, and it is time to put it into production. We can run the `batch` notebook without Jupyter using the command line.

1. Open a terminal in the same folder and make sure the 'KUBECONFIG' environment variable is set:

```
$ export KUBECONFIG=$(pwd)/kubeconfig.yaml
```

2. Before we can run tasks on the cluster we have to push an updated container image to a docker registry. This image will bundle all the code you've written along with any dependencies required to run it. It will continue to work as written, forever.

```
$ cowait build --push
```

3. The notebook can now be executed on the cluster as a batch job for a range of dates.

```
$ cowait notebook run batch.ipynb \
    --cluster demo \
    --input start=20210201 \
    --input end=20210207
```

EXAMENSARBETE Sharing local files with Kubernetes clusters
STUDENTER Martin Jakobsson, Johan Henriksson
HANDLEDARE Markus Borg (LTH)
EXAMINATOR Niklas Fors (LTH)

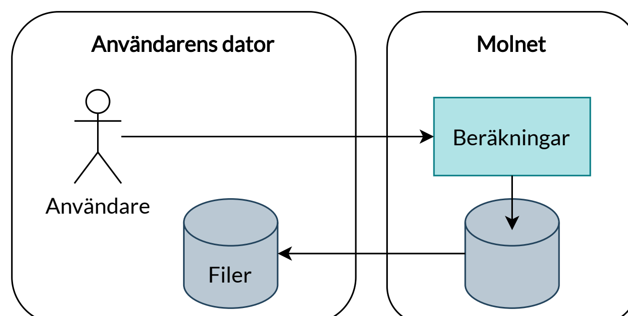
Fildelning genom en portal till molnet

POPULÄRVETENSKAPLIG SAMMANFATTNING **Martin Jakobsson, Johan Henriksson**

I takt med att världen blir mer datadriven flyttas allt fler stora beräkningar till molnet. Då tillkommer ofta problem med filhantering. Detta examensarbete bidrar med ett nytt sätt att dela filer med molnet, där molntjänsten får tillgång till filer som lagras på användarens dator.

En forskare sitter och genomför vetenskapliga beräkningar på sin dator, men i takt med att mängden data som ska beräknas växer tar beräkningarna mer tid att genomföra. För att snabba upp processen bestämmer sig forskaren för att flytta arbetet till molnet. Flera datorer kan då samarbeta för att snabbare genomföra beräkningarna, och forskaren kan då arbeta effektivare. Detta leder dock till ett nytt problem: beräkningarna är definerade i kodfiler som forskaren skapar på sin dator, men nu måste de skickas till molnservern. Om forskaren vill ändra något i dessa filer så måste den manuellt ladda upp filerna igen för att molntjänsten ska ha den senaste versionen. Detta blir snabbt en omständig process, eftersom man ofta går igenom många iterationer av ändringar i sin kod innan man når ett färdigt resultat.

För att underlätta överföringen av filer har vi skapat en portal som sätts upp mellan användarens dator och en molnserver. Användaren kan då ge molnservern tillgång till utvalda mappar och filer på sin dator. Genom att strömma läs- och skrivoperationer över internet kan molntjänsten arbeta direkt mot filer som lagras på användarens hårddisk. På så vis undviks problemet att manuellt behöva flytta filer fram och tillbaka. Dessutom finns den mest aktuella versionen alltid på användarens dator, vilket ger användaren full kontroll över sina egna filer.



Man kan dock undra om den här lösningen verkligen fungerar så bra som det är tänkt. Portalen skickar trots allt information över internet, vilket tenderar att vara långsamt. Efter noggranna mätningar har det dock visat sig att prestandan duger i de allra flesta situationer. Prestandan förväntas dessutom kunna förbättras ytterligare genom att introducera bland annat cachning.

Att dela filer genom en portal till molnet visade sig vara en lösning med stor potential. Portalen gör att steget till molnet blir kortare och användarens filer blir lättare att hantera.