TU WIEN Informatics

# Accelerating Software Integration and Delivery

## A Case Study on a FLOSS Project

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

**Bachelor of Science**

in

**Software & Information Engineering**

by

**Aydan Namdar Ghazani**
Registration Number 11709245

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Lektor Dipl.-Ing. Dr.techn. Markus Raab, BSc

Vienna, July 8, 2023

| Aydan Namdar Ghazani | Markus Raab |
|---|---|

# Erklärung zur Verfassung der Arbeit

Aydan Namdar Ghazani

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 8. Juli 2023

Aydan Namdar Ghazani

# Acknowledgements

I want to thank my family for always supporting me,
Markus and Yvonne for this wonderful opportunity of a project,
and everyone else who stood behind me, after all this time.

# Kurzfassung

PermaplanT ist ein innovatives FLOSS Projekt. Um den Entwicklungsprozess zu beschleunigen, sind gute Continuous Integration and Delivery (CI/CD) Praktiken unerlässlich. Durch die Implementierung einer effizienten deployment pipeline können Entwickler schnell Feedback zu ihren Änderungen erhalten, frühzeitig Fehler erkennen und widerstandsfähigeren Code produzieren. Durch kontinuierliche Tests und Validierung wird sichergestellt, dass das Projekt wie beabsichtigt funktioniert und ein hohes Qualitätsniveau aufrechterhalten wird. Dieser iterative Ansatz erleichtert effiziente Feedback-Schleifen und ermöglicht es dem Projekt, sich schnell an die sich ändernde Benutzeranforderungen anzupassen. Die Erstellung einer nachhaltigen deployment pipeline mit automatisierten Tests hilft PermaplanT, eine benutzerzentrierte, anpassungsfähige Lösung mit möglichen häufigen Softwareupdates zu liefern.

# Abstract

PermaplanT is a new cutting-edge FLOSS project. To accelerate its development process, the establishment of well-structured Continuous Integration and Continuous Delivery practices are essential. By implementing an efficient deployment pipeline, developers can get rapid feedback on their changes, detect bugs early and produce more resilient code. Continuously testing and validating each iteration, ensures that the project functions as intended, and maintains a high level of quality. This iterative approach facilitates efficient feedback loops, enabling the project to quickly adapt to changing user requirements. The creation of a sustainable deployment pipeline with automated tests helps PermaplanT deliver a user-centric, adaptable solution with frequent software updates.

# Contents

CHAPTER 1

# Introduction

*"Begin at the beginning," the King said gravely, "and go on till you come to the end: then stop."*

— Lewis Carroll, *Alice in Wonderland*

In the FLOSS software development realm, collaboration, and coordination among diverse contributors are key aspects in ensuring the delivery of high-quality software. With numerous developers working on different aspects of a project, efficient integration becomes a challenging task.

Continuous Integration (CI) has emerged as a powerful approach to address integration obstacles. It provides a framework for automating software build, testing, and integration processes. By frequently integrating code, teams can detect issues early on, ensuring code stability and reducing integration complexities.

Continuous Delivery (CD) extends these benefits by automating the software deployment process. It builds on top of CI and ensures that software changes are delivered to production or staging environments consistently and reliably. By automating deployment activities, such as configuration management and environment setup, CD minimizes manual errors and automates the release cycle.

In the following chapter, we will delve into the FLOSS initiative PermaplanT, exploring the research question, hypothesis, and significance of this thesis. Later on, we will look into the problems of the Deployment Pipeline in Chapter 3, provide a solution in Chapter 4, see the results in Chapter 5 and finish the thesis with Chapter 6, where we see the Consequences and talk about Future Work.

## 1.1 A Permaculture Web Application

PermaplanT [1] is an innovative FLOSS initiative that emerged at the beginning of 2023. Driven by a team of software developers and permaculture experts, this project assists users in designing and creating their permaculture gardens, by providing a user-friendly web application.

Permaculture is a sustainable system seeking to emulate observed patterns from ecosystems to create productive and resilient human settlements. PermaplanT offers an intuitive tool for users to create, draw, design, and share their permaculture maps.

One of the standout features of PermaplanT is its extensive plant database, offering users a wide selection of over 8,500 plants to choose from. With this vast collection of plant options, users can curate their permaculture landscapes and craft gardens by the standards of permaculture experts. The web application not only motivates users to design their own spaces but also provides valuable insights into plant compatibility.

Users can enjoy instant feedback as they place their chosen plants onto their virtual gardens. The application analyzes the selections and provides real-time recommendations for optimal plant placements, both up to seven years in the past and into the future. This time-traveling capability enables users to make well-informed decisions in their botanical activities.

Beyond the individual user experience, PermaplanT fosters a sense of community among permaculturists, gardeners, and plant enthusiasts. The web application offers a collaborative space where users all over the world connect, share their experiences, and seek inspiration from one another.

At its core, PermaplanT is aiming to revolutionize the world of permaculture design by combining technology with ecological wisdom.

**Note:** Ensuring a smooth CI/CD process in PermaplanT comes with its share of challenges. Since there are new code changes every day and different developers joining and leaving the team, it requires good communication between team members, making sure that everyone knows what's happening. It's like putting together a puzzle where the pieces are changing consistently. Creating a reliable and efficient CI/CD process here means not just solving technical problems but also ensuring transparency and low entry barriers.

## 1.2 Objectives, Research Questions, and Hypothesis

The primary objective of this study is to address the challenges faced by PermaplanT in its deployment pipeline and to devise a strategy for enhancing its efficiency. The research questions guide the investigation: firstly, by examining methods to reduce the lead time of PermaplanT's deployment pipeline; and secondly, by exploring possibilities for restructuring the pipeline to facilitate quicker feedback for developers. These questions are underpinned by the hypothesis that reducing the lead time will not only optimize development efficiency but also result in more frequent software releases. Through comprehensive analysis and exploration of these research questions, this study aims to offer valuable insights and recommendations for accelerating PermaplanT's deployment practices.

**RQ 1.** How can we accelerate the lead time of PermaplanT's deployment pipeline?

**RQ 2.** How can we restructure the deployment pipeline to provide developers with faster feedback?

**H 1. (RQ 1.)** By reducing lead time, PermaplanT can achieve increased development efficiency and more frequent releases.

## 1.3 Significance and Contribution

The significance of this research lies in its potential to address the challenges faced by small software development teams in implementing effective CI/CD processes that ensure fast feedback to developers.

This work aims to contribute to the existing knowledge in the field of CI/CD practices. Ultimately the FLOSS nature of the project ensures transparency and allows a wider community to access and learn from this research, enabling collaborative learning and improvement within the FLOSS software development community.

CHAPTER 2

# Terminology

*Words are the source of misunderstandings.*

— Antoine de Saint-Exupéry

This chapter delves into the essential concepts and definitions surrounding Agile Software Development and the key principles of Continuous Integration, Continuous Delivery, Continuous Testing, and deployment pipelines.

## 2.1 Agile Software Development

Agile workflow practices integrate seamlessly well with CI/CD. Together, they enable teams to build high-quality software efficiently and adapt to changing requirements more frequently. The Agile Manifesto was created back in 2001 by seventeen of the leading thinkers in software development [2]. Their goal was to create a lightweight solution against heavyweight software development processes and "deliver working software frequently". Agile development has increased the productivity of many teams and Agile conferences are the origin of many new ideas like CI/CD [2].

### 2.1.1 Continuous Integration

Continuous Integration is frequently integrating work in small sizes to a Version Control System (VCS), aiming for at least one commit per day [3]. Working in small increments minimizes complexity and helps quickly identify and resolve defects. If a team member introduces a failure and leaves, their change should be reverted to clear the path to production [3].

### 2.1.2 Continuous Delivery

Continuous as in continual [4]. A process that is always running and polling for changes in the VCS. The software stays in a releasable condition at all times, allowing us to be responsive to changes in the real world.

According to Farley [5] the term Continuous Delivery is derived from the first principle in the Agile Manifesto [6].

> *"Our highest priority is to satisfy the customer through early and continuous delivery of valuable software"*

Moreover, he highlights that CD is a state-of-the-art software development practice, which not only increases software quality and development efficiency but also provides fun to everyone participating in it.

At its core, CD aims to reduce problems in combining different software artifacts before a major release by always integrating and testing the software in smaller divided steps. Each commit to a remote VCS is run through a deployment pipeline which will test if the software is releasable.

### 2.1.3 Continuous Testing

**re · li · a · ble** | adjective | Giving the same result in successive trials. [7]

Software testing ensures the robustness, reliability, and functionality of a software product. It profoundly impacts software quality, user experience, and development confidence. A deployment pipeline without tests is like building a bridge without checking its structural integrity. Not running tests in CI means you are not practicing CI [8]. It is Continuous Testing that allows developers to integrate their changes into the VCS and have confidence in doing so.

### 2.1.4 Deployment Pipeline

> *The deployment pipeline is a machine that helps us do that, by organising our software development work, to go from Commit to Releasable Outcome as quickly and efficiently as possible, repeatably and reliably [9].*

The pipeline framework used in this project is Jenkins [10]. So the ideas that are applied here are easily translatable to any pipeline (like Github Actions [11] or Azure Pipelines [12]).

### Jenkins

A Jenkins pipeline is a structured workflow consisting of stages and steps.

Stages are logical segments representing a distinct phase or milestone and serving as building blocks for the overall workflow.

Within each stage, there are steps. Steps are individual actions or operations that are performed sequentially. They represent the specific tasks that need to be accomplished as part of the larger stage. These steps encompass various actions such as building code, running tests, deploying applications, and more.

One crucial aspect of the pipeline's orchestration is that stages are executed sequentially. Each stage waits for the successful completion of the previous stage before proceeding. This dependency ensures that the pipeline progresses in a controlled and organized manner, where subsequent stages can rely on the successful outcomes of earlier stages. Parallel stage execution is also possible. In this case, successful completion of all parallel stages is expected before proceeding to the next stage.

### Lead Time

The *lead time* [13] is the speed of a deployment pipeline, which influences various facets of a development lifecycle and, ultimately, the success of a project. A swift deployment pipeline gives developers timely feedback on their code changes, allowing for rapid iterations and adjustments across the whole team leading to faster feature deliveries and requirement satisfaction. Speed and opportunity costs are tightly coupled, as lacking deliverable software can ultimately lead to losing out to the competition. It's important to note, that stability should never be compromised in favor of lead time. A fast but incorrect pipeline is equally worse than a slow but stable one.

# CHAPTER 3

# Problem

*The obstacle is the path.*

— Zen Proverb

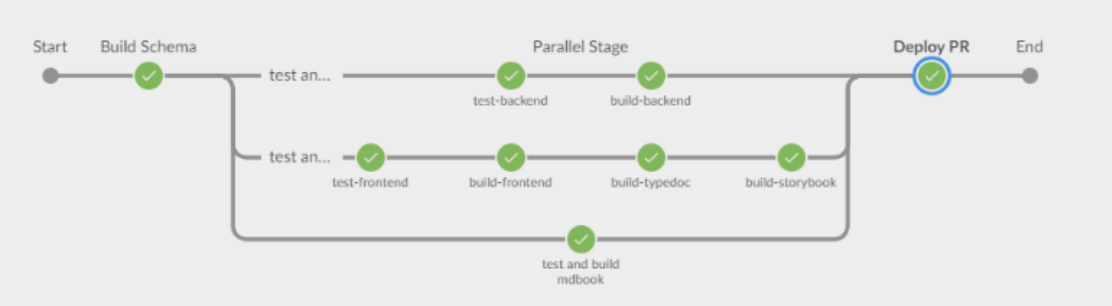


Figure 3.1: The Stages of PermaplanT's Deployment Pipeline [14]

## 3.1 Poor Lead Time

PermaplanT's deployment pipeline, Figure 3.1, encompassed around 30 minutes. Notably, the 'test-backend' stage occupied a substantial portion, spanning around 19 minutes; the 'build-backend' stage was about 8 minutes, see the left side of Table 5.2.

Examining the distribution, approximately 65% of the total lead time was dedicated to testing, and close to 30% was allocated for building. Collectively, these two stages accounted for 95% of the overall lead time.

On the left side of Table 5.2 we can see that the steps 'Package', 'Test', and 'Build' occupy most of the lead time.

## 3.2 Opaque Structure

The deployment pipeline was also lacking a transparent and meaningful structure. Faster steps like ‘Syntax’ and ‘Documentation’ should be at the front of the pipeline to reveal bugs promptly, and if possible, be parallel executed. Moreover, bottlenecks and areas for improvement should be transparent and not hidden away. For example, if a developer committed a documentation error, he would need to wait 8 minutes before realizing that failure.

CHAPTER 4

# Method

*For now, what is important is not finding the answer, but looking for it.*
— Douglas R. Hofstadter *Gödel, Escher, Bach: An Eternal Golden Braid*

## 4.1 Limitations

Since we were not in power of Jenkins scheduling algorithm, it was not possible to always assign the same worker to the same stage.

## 4.2 Parallelization

> *RQ 1. How can we accelerate the lead time of PermaplanT's deployment pipeline?*

By implementing strategies to improve parallelization, the deployment pipeline can effectively utilize more available resources reducing lead time and providing developers with quicker feedback

### 4.2.1 Parallelizable Steps

Before parallelizing, we had to ensure that steps were not depending on each other. After guaranteeing independence, we were able to assign additional free resources to lower the lead time.

The approach involved planning and analyzing different combinations before deciding on one. We will not discuss variants that were discarded for obvious reasons and rather focus on the best three candidates.

In Table 5.1 we can see, that using four parallel nodes did not reduce the lead time further, but could provide developers with faster feedback. We decided against this solution, to not occupy too many workers at once.

We also realized that the step 'Package' was redundant, so we removed it.

### 4.2.2 Implementation

To implement this, we had to extract a general function. By doing so, we could run many parallel stages invoking this new function. The pipelines framework then handled load balancing and execution.

## 4.3 Restructuring

> *RQ 2. How can we restructure the deployment pipeline to provide developers with faster feedback?*

After splitting up the sequential stages into parallel atomic stages, we are naturally solving structural and resource problems, providing more transparency and aborting failed runs quicker.

### 4.3.1 Sanity Stage

A sanity stage encapsulates many tasks, which finish within a couple of seconds. This way we can provide developers with almost instantaneous feedback and motivate future work to append similar sanity tests to this stage.

The final pipeline can be seen in Figure 5.1.

CHAPTER 5

# Results

*Trust, but verify.*

— Russian proverb

## 5.1 Deployment Pipeline

Looking at Table 5.2 a speed-up of almost 50% can be achieved.

| Stages | Minutes |
|---|---|
| Test | 8 |
| Build + Syntax | 12 |
| Package + Documentation | 10 |
| Steps Total | 31 |
| Pipeline Total | 16 |

| Stages | Minutes |
|---|---|
| Test | 6 |
| Build | 15 |
| Package + Syntax | 10 |
| Documentation | 8 |
| Steps Total | 39 |
| Pipeline Total | 16 |

Table 5.1: Three (left) & Four (right) Parallel Stages [15]

| Steps | Minutes |
|---|---|
| Formatting | 0.01 |
| Package | 7 |
| Syntax | 1 |
| Documentation | 4 |
| Test | 8 |
| Build | 8 |
| Steps Total | 28 |
| Pipeline Total | 30 |

| Stages | Minutes |
|---|---|
| Formatting | 0.01 |
| Syntax & Documentation | 3 |
| Test | 12 |
| Build | 11 |
| Steps Total | 26 |
| Pipeline Total | 17 |

Table 5.2: Sequential (left) [14] & Parallel (right) [16]

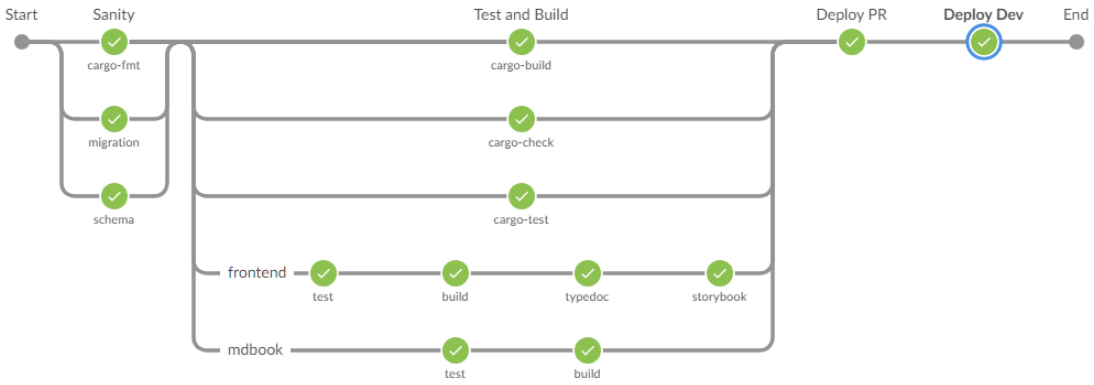


Figure 5.1: Accelerated Pipeline in Jenkins [17]

## 5.2 Productivity

*H1: (RQ 1.) By reducing lead time, PermaplanT can achieve increased development efficiency and more frequent releases.*

Examining Figure 5.2 and 5.3 we can see an increased productivity, therefore accepting our hypothesis.

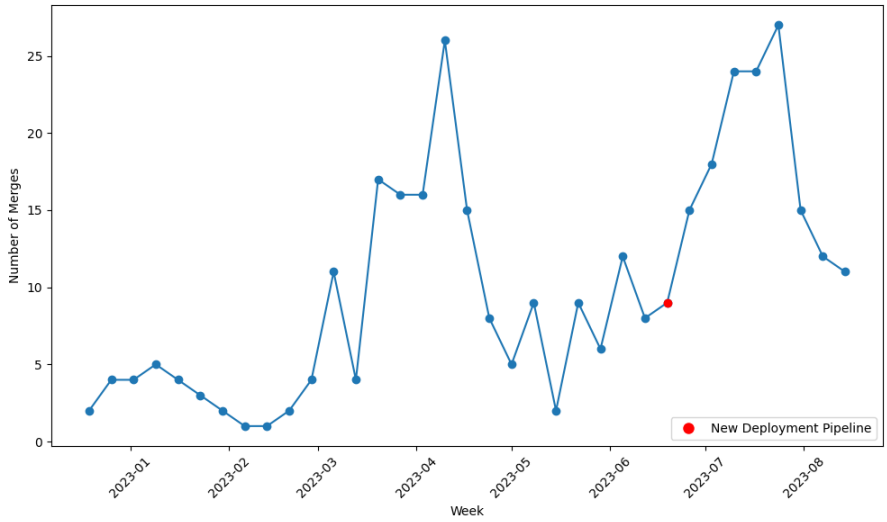


Figure 5.2: Merged Pull Requests per Week

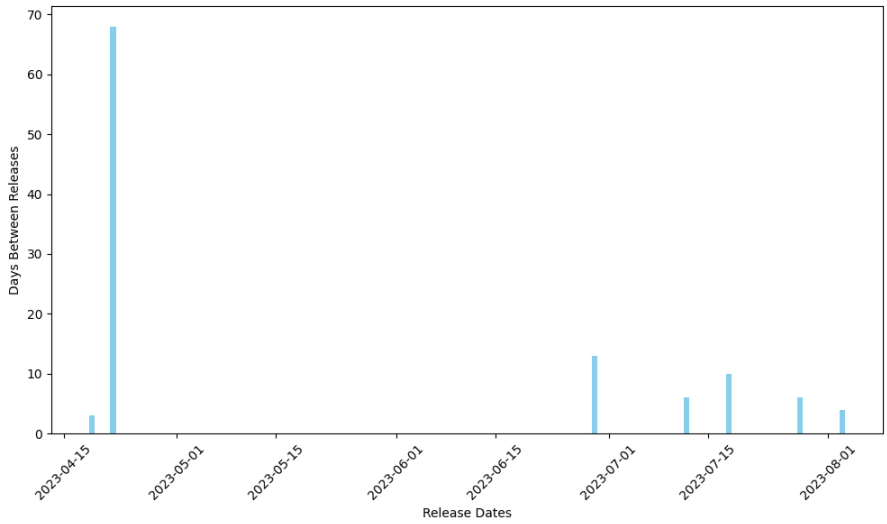


Figure 5.3: Release Frequency

CHAPTER 6

# Conclusion

*Sharing is good, and with digital technology, sharing is easy.*

— Richard Stallman

## 6.1 Consequences

### 6.1.1 Development

The pace of development is strictly intertwined with the lead time and structure of a deployment pipeline. A slow lead time exerts a direct impact on the development speed; at the same time, a poorly structured one will fail to deliver rapid feedback. Not doing so results in hindering developers' efforts rather than supporting them. A slower pace of development directly translates into a reduced frequency of deployments.

### 6.1.2 Deployment

The frequency of software deployment stands as a combined outcome of lead time and development speed, which is again bound to the lead time of a deployment pipeline. When deploying, we are experiencing the lead time of a pipeline twice.

### 6.1.3 Resources

When a failure is hiding in the back of a prolonged pipeline, resources are being wasted, eventually blocking all free workers for a failing deployment.

## 6.2 Future work

Some problems remain unsolved:

1. The frontend stage resembles the old ‘test-backend’ stage in its complexity. Some refactoring could be done, eventually extracting fast steps into the sanity stage (version check, linting, typedoc).

2. Caching of downloads and compilations [18] could be done to prevent repetition.

3. The potential need for refactoring the source code of the deployment pipeline in the event of increased complexity, aiming to modularize it, potentially allowing multiple different pipelines to coexist, without duplicating code. For example, see here [19].

# List of Figures

# List of Tables

# Glossary

**FLOSS** An abbreviation for Free/Libre and Open Source Software, refers to software that is both free in terms of user freedom and open source in terms of its accessible source code [20].. vii, ix, 1–3

**worker** is a computing unit within a distributed environment that performs tasks or computations.. 11, 12, 17

# Acronyms

**CD** Continuous Delivery. 1, 5, 6

**CI** Continuous Integration. 1, 5, 6

**CI/CD** Continuous Integration and Delivery. vii, 2, 3, 5

**VCS** Version Control System. 5, 6

# Bibliography


[1] "Permaplant website." https://www.permaplant.net. Accessed: 2023-08-16.

[2] G. Kim, P. Debois, J. Willis, J. Humble, and J. Allspaw, *The DevOps Handbook: How to Create World-class Agility, Reliability, and Security in Technology Organizations*, p. 4. G - Reference,Information and Interdisciplinary Subjects Series, IT Revolution Press, 2016.

[3] D. Farley, *Continuous Delivery Pipelines: How to Build Better Software Faster*, p. 36. Independently published, 2021.

[4] P. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*, p. 27. Addison-Wesley Signature Series, Pearson Education, 2007.

[5] D. Farley, *Continuous Delivery Pipelines: How to Build Better Software Faster*, p. 3. Independently published, 2021.

[6] "Agile manifesto." https://agilemanifesto.org. Accessed: 2023-08-16.

[7] "Reliable." https://www.merriam-webster.com/dictionary/reliable. Accessed: 2023-08-17.

[8] P. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*, p. 15. Addison-Wesley Signature Series, Pearson Education, 2007.

[9] D. Farley, *Continuous Delivery Pipelines: How to Build Better Software Faster*, p. 8. Independently published, 2021.

[10] "Jenkins." https://www.jenkins.io. Accessed: 2023-08-16.

[11] "Github actions." https://github.com/features/actions. Accessed: 2023-08-17.

[12] "Azure pipelines." https://azure.microsoft.com/en-us/products/devops/pipelines. Accessed: 2023-08-17.

[13] D. Farley, *Continuous Delivery Pipelines: How to Build Better Software Faster*, pp. 77–78. Independently published, 2021.

[14] "Old pipeline." https://build.libelektra.org/blue/organizations/jenkins/PermaplanT/detail/master/270/pipeline/194. Accessed: 2023-08-19.

[15] "Parallelization pull request." https://github.com/ElektraInitiative/PermaplanT/pull/577. Accessed: 2023-08-19.

[16] "New pipeline." https://build.libelektra.org/blue/organizations/jenkins/PermaplanT/detail/master/310/pipeline/65. Accessed: 2023-08-19.

[17] "Accelerated pipeline in jenkins." https://build.libelektra.org/blue/organizations/jenkins/PermaplanT/detail/master/319/pipeline. Accessed: 2023-08-19.

[18] "Caching the cargo home in ci." https://doc.rust-lang.org/cargo/guide/cargo-home.html#caching-the-cargo-home-in-ci. Accessed: 2023-08-19.

[19] "A modular pipeline." https://github.com/apache/mxnet/tree/master/ci/jenkins. Accessed: 2023-08-19.

[20] "Floss/foss." https://www.gnu.org/philosophy/floss-and-foss.en.html. Accessed: 2023-08-18.