BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

BULK OPERATION ORCHESTRATION IN MULTIREPO CI/CD ENVIRONMENTS

HROMADNÁ ORCHESTRÁCIA V MULTIREPO CI/CD PROSTREDIACH

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. JAKUB VÍŠEK**
AUTOR PRÁCE

SUPERVISOR **Ing. MICHAL KOUTENSKÝ**
VEDOUCÍ PRÁCE

BRNO 2021

Brno University of Technology
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2020/2021

Master's Thesis Specification

Student: **Víšek Jakub, Bc.**
Programme: Information Technology and Artificial Intelligence
Specialization: Computer Networks
Title: **Bulk Operation Orchestration in Multirepo CI/CD Environments**
Category: Networking
Assignment:

1. Familiarize yourself with the principle of CI/CD and existing solutions.
2. Familiarize yourself with the multirepo approach to software development.
3. Analyze the shortcomings of existing CI/CD solutions in the context of multirepo development with regard to user comfort. Focus on scheduling and deploying bulk operations on multiple interdependent repositories as part of a single logical branching pipeline.
4. Propose and design a solution to these shortcomings.
5. Implement said solution.
6. Test and evaluate the solution's functionality in a production environment.

Recommended literature:

- Humble, Jez, and David Farley. *Continuous delivery*. Upper Saddle River, NJ: Addison-Wesley, 2011.
- Forsgren, Nicole, Jez Humble, and Gene Kim. *Accelerate : building and scaling high performing technology organizations*. Portland, OR: IT Revolution Press, 2018.
- Nicolas Brousse. *2019. The issue of monorepo and polyrepo in large enterprises.* In Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming (Programming '19). Association for Computing Machinery, New York, NY, USA, Article 2, 1-4.
- Ciera Jaspan et. al. *2018. Advantages and disadvantages of a monolithic repository: a case study at google.* In Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '18). Association for Computing Machinery, New York, NY, USA, 225-234.

Requirements for the semestral defence:

- Items 1 to 4.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Koutenský Michal, Ing.**
Head of Department: Kolář Dušan, doc. Dr. Ing.
Beginning of work: November 1, 2020
Submission deadline: May 19, 2021
Approval date: October 26, 2020

Abstract

The multirepo model, where solution code is managed in many separate source control repositories instead of a single one, is gaining traction in software engineering. Amongst the disadvantages of this strategy is the amount of mundane work prone to human error involved in performing bulk operations, especially when these repositories are diverse in structure and utilized technologies. This thesis aims to design and implement a solution focused on time-saving and convenience of use that will allow for the definition and orchestration of development processes concerning many separate source control repositories. Finally, the completed solution is piloted in the production environment and evaluated.

Abstrakt

Multirepo model přístupu ke správě a verzování zdrojového kódu, jež zahrnuje použití mnoha oddělených repozitářů verzovacích systémů, je poslední dobou často zmiňován v odborné literatuře. Jednou z jeho nevýhod je množství zdlouhavých, nezajímavých a repetitivních úkonů, které je nutno provádět při hromadných operacích tvořících transakce napříč těmito repozitáři. Multirepo repozitáře navíc umožňují využití široké škály technologií, což jen umocňuje riziko lidské chyby, ke které při ručně prováděných hromadných operacích může dojít. V rámci této práce je navrženo, implementováno a otestováno řešení pro automatizaci operací prováděných napříč množstvím repozitářů uspořádaných v multirepo modelu, což s nimi uživatelům zlepšuje zkušenost.

Keywords

multirepo, monorepo, polyrepo, metarepo, CI/CD, Continuous Integration, Continuous Delivery, Continuous Deployment, release orchestration, bulk operations

Klíčová slova

multirepo, monorepo, polyrepo, metarepo, CI/CD, Continuous Integration, Continuous Delivery, Continuous Deployment, orchestrace release procesu, hromadné operace

Reference

VÍŠEK, Jakub. *Bulk Operation Orchestration in Multirepo CI/CD Environments.* Brno, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Michal Koutenský

Rozšířený abstrakt

Tato diplomová práce si klade za cíl usnadnit a zefektivnit každodenní práci softwarových inženýrů vyplývající z přístupu, jakým je spravován zdrojový kód a samotný proces vývoje projektů, na kterých pracují. Oba tyto faktory ovlivňuje celá řada okolností, od způsobu vedení a řízení projektů, přes velikost týmů, dostupnou infrastrukturu až po použité nástroje a technologie.

Úvodní část práce je věnována problematice *kontinuální integrace*. Spolu s dopady, jež jsou se zavedením a provozem popsaných praktik typicky pozorovány v oblasti vývojového procesu a jeho účastníků, jsou zmíněny také s tím spojené prerekvizity. Součástí úvodu je také popis nejznámějšího rysu kontinuální integrace, tzv. *pipeline*, jejích hlavních součástí a dále základní doporučení týkající se jejich realizace. Následně jsou krátce představeny další problematiky často považované za následníky kontinuální integrace, a sice *kontinuální deployment* a *kontinuální doručování*.

Dalším předmětem zájmu této práce jsou verzovací systémy pro správu zdrojových kódů, a to jak z hlediska jejich technických aspektů a jimi využívaných principů fungování, tak způsobů, jakými jsou tyto systémy používány uživatelskou komunitou. V tomto ohledu jsou zejména představeny dva modely přístupu k využívání verzovacích repozitářů zdrojových kódů: modely monorepo a multirepo. K oběma postojům jsou shrnuty argumenty ze zkoumaných literárních i publicistických zdrojů. Nakonec je poskytnuto srovnání výhod a nevýhod podle několika zvolených kritérií.

Na základě spolupráce s týmem vývojářů zapojených do softwarového projektu spravovaného v souladu s multirepo modelem se zbývající část práce věnuje právě tomuto z nich, vůči jehož adopci zastává dostupná literatura převážně negativní postoj. Je identifikováno množství dílčích úkonů, kterým se členové tohoto týmu pravidelně věnují. Prostřednictvím uživatelského průzkumu je zjištěno, které z těchto činností jsou považovány za nejvíce uživatelsky nepřívětivé a dále, které faktory se na tomto hodnocení nejvíce podepsaly.

Analýza existujících nástrojů, kterými by zkoumané činnosti a s nimi spojené faktory nepřívětivosti bylo možné usnadnit či zmírnit, nepřináší uchopitelné výsledky. Dochází proto k definici požadavků na nový systém, který by dokázal adresovat nejpalčivější ze zkoumaných operací: propisování změn v multirepo projektech napříč stanoveným stromem závislostí. Tyto požadavky jsou stanoveny tak, aby jejich naplnění vedlo k bezpečnému, rozšiřitelnému a do jisté míry univerzálního systému, schopného integrace s širokou škálou systémů pro verzování a sestavování zdrojových kódů. Zároveň jsou reflektovány skutečné požadavky účastnící se skupiny vývojářů, aby bylo vytvořený systém možné pilotně nasadit a otestovat v jejich produkčním prostředí a vhodně tak zhodnotit jeho přínos.

Takto získané specifikace jsou rozloženy na komponenty a následně jsou zvoleny nástroje a postupy pro jejich realizaci. V rámci návrhu jsou pro každou z komponent pečlivě uváženy její odpovědnosti a rozhraní pro integraci se zbytkem systému. Návrh zohledňuje požadavky na rozšiřitelnost vhodným rozdělením systému na komponenty jádra a zásuvné moduly. Podobně je přistoupeno k zadané sledovatelnosti, ovladatelnosti a opakovatelnosti dílčích kroků prováděných v rámci jednotlivých realizovaných orchestračních úkolů.

Implementace systému je provedena s důrazem na stabilitu a verifikaci vyvíjených komponent. Hotové komponenty jsou poté integrovány v souladu s popsanými praktikami kontinuální integrace a kontinuálního deploymentu. Řešení je opatřeno dvojicí vysokoúrovňových zkušebních sad, které v rámci vytvořené CI/CD pipeline umožňují automatizovaným způsobem validovat schopnost systému dostát zadaným nárokům.

Bulk Operation Orchestration in Multirepo CI/CD Environments

Declaration

I hereby declare that this master's thesis was prepared as an original work by the author under the supervision of Ing. Michal Koutenský. The supplementary information was provided by members of the Sky ecosystem team. I have listed all literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Jakub Víšek
May 19, 2021

Acknowledgements

I'd like to thank Michal, my supervisor, for his time spent consulting and reviewing this text. I am also very grateful to the Sky team members for their patience during our consultations. A special dose of thanks is reserved explicitly for S.N. and B.P. for establishing the topic and helping me jump through the hoops of its submission.

On a more personal note, I'd like to thank Zbyněk for his support and patience while I was working on this project and to my academic colleagues Honza & Honza for all the good memories we've had so far.

Contents

Chapter 1

Introduction

Software engineering, its management, software quality assurance or operations engineering — each of these fields constitute fully-fledged paths to which one could surely devote their entire research or professional career. Each comes with its own set of goals that, while at heart sharing a common purpose, often contradict, at least partially. Nevertheless, continuous integration is an effective tool with positive impacts on all of the aforementioned disciplines and more. The first chapter discusses the practice in detail and describes the currently available solutions that make adopting it easier.

The second chapter provides a brief introduction to source code versioning systems before moving on to a bigger picture that concerns the patterns of these systems' usage: two distinct, mostly opposite repository models, each coming with its own set of features and characteristics. The qualities of both approaches are compared; their inherent benefits and drawbacks are discussed as well.

Furthermore, the less-discussed model of the two, the multirepo model, is put into the context of the day-to-day activities of a software engineer. The third chapter aims to point out the differences in regular development workflows when working in such an environment and identify their flaws regarding user comfort through a user study.

The next chapter follows up on the results of that study by specifying a set of requirements to address the discovered problems and designing a solution capable of addressing them. The solution is then implemented with the goal of a pilot deployment in mind, which would confirm the solution's ability to fulfill the defined requirements and help the engineers' mentioned team with their regular duties.

The final chapter discusses the available options of deployment and some of the related specifics. The pilot deployment to the production environment is evaluated and described, including the used methods of evaluation.

Chapter 2

Continuous Integration, Continuous Delivery

When working on a small project intended for personal or limited use only, a software engineer is often on his own during all phases of the software development cycle. The deployment that theoretically follows successful implementation and testing may not even happen in practice as the application in question could as well already be present in its intended production environment – the engineer's computer. Moreover, the intensity of completed testing may vary in regards to the developer's time pressure, experience, or choice of development methodology, from simple black-box testing utilizing two pairs of sample inputs and outputs to a comprehensive test suite that spans various levels of the application's architecture.

Collaborating in a team brings forth the need to synchronize changes from many authors with possibly different development practices. Even if each of these authors did their due diligence to the furthest extent imaginable and verified that their contributions match all specified requirements, conflicts are bound to happen. These generally arise when two development branches are merged or *integrated*, meaning the software worked on can no longer be built or no longer functions as expected. While some approaches to software development are more prone to painful integration periods than others, merge conflicts and undesirable side effects of seemingly successful merges always manifest.

Faced with a failed integration, a development team backed by a company whose product or service the application provides has to spend additional time resolving the encountered conflicts and/or patching the introduced discrepancies in behavior. Even worse, this assumes that these discrepancies are discovered at the time of the merge, which may not always be the case. However, not all software development has this kind of financial backing to provide an incentive to help correct unsuccessful integration. Where does a maintainer of an open-source software find time to read through, build and test a feature contribution, then make an acceptance decision and repeat the whole ordeal? What about the contributor, who now cannot get the whole application to build on his machine after all the time spent creating a particular feature? Is there perhaps a way the whole process could have been made simpler or easier?

2.1 Continuous Integration

The practice of continuous integration, or CI for short, aims to ensure that any time source code is retrieved from the storage system used by developers, it shall be possible to build it into working software [15]. In a traditional environment that does not embrace and suitably implement the CI practice, one cannot trust software built from a retrieved version of the source code to function up to the requirements until it is subjected to the testing process. In a CI environment, multiple measures are implemented, each of which strives to raise confidence in the source code and its resultant artifacts.

The paramount principle that essentially sets the direction for the remaining ones is that integration shall happen every time a developer submits a change to the code base. Should it fail for any reason, no additional changes shall be submitted, except for those whose goal is to resolve the failure. Only after the original changes are properly incorporated is it possible to submit further changes [15].

Shift of Responsibilities

The recipe for fast and effortless integration lies in the limitation of their scope, as emphasised by the desire for their *continuous* character. Submitting small chunks of work, no less than daily, and integrating them as part of the submission process, makes for numerous benefits [15]:

- Lesser amount of changes implies less space for potential conflicts with changes of other collaborators of the project.
- Concerning the purpose of integration, the delivered changes must constitute a unit of work, i.e., an internal refactoring with no effect on functionality or a change in behavior that honors some specification. This way, these behavioral changes are easier to track than when integrating with larger chunks or only once at completion.
- Incremental changes integrated regularly can help discover conflicts early on and potentially trigger changes in solution design or problem specification.

Despite these benefits, the developers themselves must, after all, adhere to the methods and techniques required for the efficiency and rewards of CI. It may take a serious amount of effort to adjust their way of working to be CI-compatible. Developers reluctant to adapt, especially those who contribute with low frequencies and high volumes of changes, may find themselves in unpleasant situations. Their completed changes could fall behind the rapidly changing codebase, and they may have to spend even more time merging these back.

However, circumstances like these may occur even when changes are otherwise dutifully integrated continuously. Just imagine a large scale refactoring or a forced sudden migration to a new platform. At times like these, it is necessary for every member of the team to accept shared responsibility and to prioritize returning the application to a verifiable working state until moving on to other tasks. During usual operation, the author of the changeset is held responsible for the success of the resulting integration step and should follow the same policy of waiting for its result before switching context.

Process Automation

Due to the frequency at which team members are expected to undergo the integration process, they must ensure that this procedure is fast and reliable. Specifically, the time it

takes for a set of submitted changes to pass or fail shall not exceed 10 minutes [15]. The need for reliability stems from the faith placed in the implemented measures, ultimately summed up into the changes' acceptance verdict. Unsound and slow processes will ultimately lead people to circumvent or take shortcuts across the system instead of utilizing its potential. Moreover, it is hard to picture a system where the mechanism for building source code and testing it within the appropriate scope would be manual, partially manual, or simply not fully automated. The mentioned time limit and reliability requirements would likely run afoul in a scenario giving space to the expression of human error.

Aside from making the CI process tolerable, having it automated and performing well means there is at least a reference build environment. This may be useful when investigating issues building elsewhere. Depending on the kind of tests and technology in use, the environment may even serve as a starting point for establishing the parameters of the deployment environment where the software will eventually run.

In addition, the automated process' definition can serve as an authoritative guide to building and testing the application. As opposed to documentation, this definition never turns obsolete. As developers learn to rely on it, it will be expected to *work*, regardless of the system load, network utilization, or other factors. This may not be the case at first, and as the automation script is a software product like any other, its upkeep and maintenance will be required from time to time, as is the case with the one(s) which it helps create.

Testing

If a set of submitted changes was approved and accepted only based on a successful build, the resulting guarantees would not be very assuring. While the ability to create an executable output out of source code is an important prerequisite that also entails, e.g., the ability to resolve build-time dependencies, it does not reveal much about the quality of the application nor of its ability to perform as expected. For this purpose, an elaborate testing procedure needs to exist. The most important decision origin of the integration process practically depends on the test suite: *'does this software (still) work?'*

Software testing is a complex topic, and, as a whole, most definitely cannot be scaled into the CI process for all but tiny software projects. Many kinds of tests are impossible to automate or impossible to automate reliably. Other kinds have hard requirements upon the testing environment or simply execute for longer than the integration process can reasonably tolerate.

Unit & Component Tests

The developer is expected to write unit tests in complement and alongside functional units. They usually provide *test doubles* for all dependencies of the unit under test, whether external (like a database system) or internal (another unit). Test doubles are often provided as mocks or fakes. Fakes are implementations of an interface that are programmed to take some shortcuts so that the original dependency can be excluded, such as using in-memory storage instead of communicating with a database. Mocks are instances without programmed behavior that still match the interface, but their members only follow a pre-programmed scenario — e.g. method `GetName` should always return `TestName123` and method `GetTemperature` should return `20` before `GetName` is called and `21` after it is called at least once.

The absence of real dependencies lends itself to running these tests within the CI process. They can be executed in parallel, evaluated in a very short time, and run with very

low requirements upon the environment. Some programming paradigms, such as test-driven development, even go as far as to dictate writing unit tests before implementing the functionality itself to avoid the introduction of untested code. Regardless of the programming approach one chooses, at least 80% of functional code should be *covered* by the accompanying unit tests [15]. The term is coined by the *code coverage* metric, which signifies the percentage of code that has been executed during a particular invocation of the program and is nowadays provided by the absolute majority of testing tools and frameworks.

The idea behind component tests is to assure that the intercommunication of units still yields the expected results, even with some dependencies present. External dependencies are still faked or mocked, but some internal bindings are left as they are.

Non-Functional Tests

Despite the present limitations, several additional checks can be made during the CI process that will help assure the quality of builds or the process itself [15].

Some of the criteria have already been mentioned. However, their enforcement belongs here, such as performing execution path analysis to assert that a given percentage of the code is covered by unit tests or verifying that the tests' execution is not taking too long.

Other factors that come into consideration when deciding to prevent a build from passing integration are compiler warnings, which typically signify the presence of unwelcome code structures or inconsistent code style such as bad indentation or non-adherence to naming conventions.

2.2 The Deployment Pipeline

An environment that is continuously integrated features build and verification steps that are streamlined for performance and reliability. These incorporated processes provide a certain degree of assurance that the application's components should perform up to expectations, but there is still much to be gained.

First of all, these expectations are set forth by the developer, who, despite their undoubtedly best intentions, may have limited knowledge of the application domain, which may affect the expectations' correctness. Moreover, some of the acceptance criteria are out of the developers' scope, and for many, verification consumes more time and resources than is available in the integration phase. Finally, the teams responsible for the remainder of the delivery process will not have it easy if their entire input is the application binaries. It may be unknown or unclear how they are to be deployed. The deployment may also naturally fail for some of the builds, and when a defect is discovered in the end, developers receiving build feedback this late may make it more difficult for them to resolve.

Having continuous integration in place establishes the first stepping stone towards the so-called *deployment pipeline*, the abstract embodiment of the entire software delivery process, illustrated in the figure 2.1, into which developers deliver source code, and which, at some points in time, hands the released application off to its intended users [15].

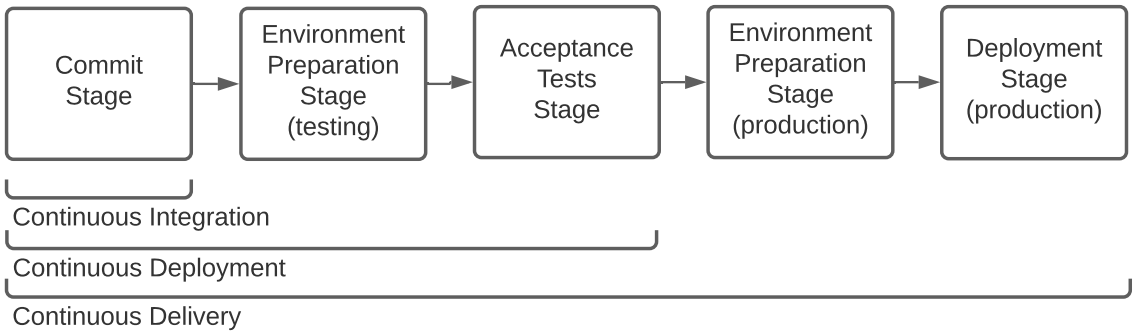

Figure 2.1: Stages of the deployment pipeline and the transitions between them. Indicated below are the extents to which they are usually adopted in an environment that is claimed to be integrated, deployment or delivered continuously.

Continuous Deployment

Every delivered change set triggers the initial commit stage, whose purpose and mechanisms have been detailed in the previous section 2.1. In summary, the application is built from source and subjected to fast, automated testing targeting small logical units. Assuming that the particular version of the application passes all defined checks, it becomes a potential release candidate and can transition further through the pipeline.

When developing software, the change of priorities, where ensuring that the software is in a working state gains precedence over introducing new features, is the keystone of continuous delivery [28].

With progress through the individual stages of the pipeline, the confidence in the respective version rises, and so does the willingness to begin its human verification. Teams involved in the delivery process are enabled to work continuously. For example, software testers can delve into any build that passes the commit stage instead of waiting for one that is proclaimed a release candidate when no process exists that would yield these on an ongoing basis.

Testing Environment Preparation Stage

The constant stream of release candidates that are ready for acceptance testing could on its own easily overwhelm any reasonably sized team responsible. Therefore, all developers are encouraged to submit their changes several times a day. This calls for the application of the same principles that originally led to this situation, i.e., automation of processes, else they risk failing to keep up with the pace or with the expected performance.

The primary responsibility of the testing deployment stage is to establish an environment in which the application can run. For best results, this environment should be as similar to the expected production environment as possible. This includes preparing instances of database servers, message brokers and other dependencies that the application needs to function. While there is room for exceptions, e.g. if some external dependency does not provide an adequate testing instance, all introduced changes come with a possibility that critical bugs and incompatibilities will not be discovered until the very last stage.

In an analogy with the commit stage, the involvement of manual operations is undesirable, as it can lead to the introduction of errors of human nature and others arising from outdated or incorrect documentation [15].

The stage must prepare the environment in a suitable way for automated deployment by both its expected users in the following stage. Therein, many of the tests are initialized automatically. Still, their success anticipates a follow-up by a human test engineer, whose expertise is best used elsewhere than getting the environment set up and ready, even more so given the procedure's mentioned full automation. The deployment is recommended not to take any inputs other than the version selection itself [15].

Acceptance Tests Stage

Where unit tests and component tests are of technical character and aim to answer *'Does the program behave as the developer expected?'*, the goal of acceptance tests is to answer *'Does the program do what the customer asks of it?'* The aspects that need to be examined to make up the answer can be split into two categories.

Functional acceptance tests are closely tied to the use cases that the customer has for the application in development. They can commonly be formulated into a *given-when-then* model that specifies the preconditions to establish, events to simulate and postconditions to assert [15]. Two types of testing paths, analogically with the execution path a computer will walk through during the test case, are distinguished — *happy* and *sad*. The happy path most often follows the original specification verbatim and verifies that the system under test executes the feature from a valid initial state. Sad paths, sometimes also called alternate paths, then verify that the system also behaves correctly when the initial state is invalid or when the execution fails.

Non-functional acceptance testing exists to assert the fulfillment of non-functional requirements. To illustrate, these can include security or availability, e.g., the number of requests per second that the application can handle. While it may be possible to automate the evaluation of some of these requirements at least partially, their nature seeps into the hard-to-completely-automate testing territory. Notable additional examples include exploratory testing, where a human operator navigates the application in an attempt to observe odd or even provably incorrect behavior, or usability testing, which often has some of the intended users interact with the application to execute the specified use cases, to expose behavior or design that impedes upon the user experience [15].

Despite being unable to automate all tests that one desires to make use of, and therefore unable to condition accepting or rejecting submitted changes based on their result, the tests for which this does not hold still yield great benefits. Most particularly, having the tests automated means that developers receive feedback much faster and that software testers get their hands on builds where their human qualities are actually indispensable and do not waste time and effort on obviously failing versions [15].

The automated portion of the stage, together with the environment's initial preparation and deployment, should not take more than two hours to provide feedback. As was the case with low-level tests, the respective acceptance tests do not generally depend on each other and are usually easily parallelized [15].

Artifact Repositories

By definition, anything that is produced by the software development cycle is an artifact. That includes documentation, configuration, guides, test execution reports and executable

files, sometimes referred to as binaries. While retaining all these outputs has a purpose on its own, e.g., tracking development progress or making them available for download by authorized users, there are additional points to consider.

A majority of artifacts are produced by the various stages of the deployment pipeline and therefore are in a one-to-one relationship with a changeset that originally caused the pipeline to start at the commit stage. None of the aforementioned artifacts can generally be said to be compatible across various builds, which means they need to be versioned.

The naive approach of versioning them alongside the source code presents many disadvantages, the most significant of which would mean there is more than one changeset associated with a build, specifically no less than two — one with the source code and one with the outputs. Another drawback to consider is that code versioning systems are designed to maintain an extensive history. Still, many artifacts become unimportant and may be discarded in time, such as when the build they were produced from fails [15].

Moreover, stages of the pipeline typically depend on artifacts produced by their antecedents, e.g., the acceptance testing stage needs to access the executable files of the application so that it can be started and verified. Although it would be possible that the stage repeats the build step to obtain the required files, this operation would introduce a risk of generating a different executable every time this happens. While the risk of this introducing a different behavior is fairly low, the build process typically consumes valuable resources. Either way, stage artifacts usually need to be stored at least temporarily for the duration of the pipeline and for successful builds. It might be preferred to retain them for longer periods of time, e.g., to allow later manual retrieval of the exact executable files for the very same reason.

Artifact repositories are a special kind of version control systems that exist to address the above requirements. Many existing continuous integration server solutions provide these in an integrated manner; however, standalone systems are available as well.

Metric Tracking

Centralizing all processes related to the deployment of the software within the deployment pipeline also turns it into a viable source of information. The delivery team can leverage this to extract performance indicators, sometimes also called metrics. A universal example applicable for any continuous integration environment is the already mentioned cycle time. Other commonly tracked metrics include the test coverage ratio, code complexity, commit stage duration or the number of builds per day [15].

These values provide only limited validity on their own. Having two projects where the first has a longer cycle time than the second does not immediately imply something wrong with the latter. However, if the commit stage duration metric also exhibits significantly higher values, it may be worth looking into the potential issue. As metrics are by definition measurable, it is possible to identify possible causes, attempt to rectify them and consequently see if the executed changes helped shift the metric value towards the acceptable range.

Having these metrics recorded, tracked and monitored throughout regular operation can also inform of issues with the infrastructure or upgrading dependencies such as libraries, compilers, interpreters or runtimes. Deviations from the baseline data can signify degradation that could cause the software's delivery to be delayed if otherwise unnoticed or uncorrected.

Visibility

As more people become involved with the deployment pipeline, the importance of everyone having easy access to its state and outputs rises. Providing feedback is one of the cornerstones of the entire continuous methodology. Developers need to be made aware of whether their changes passed the commit stage. The operations staff is interested in the status of ongoing deployments. Everyone is interested in a subset of the recorded metrics to see how their immediate area of interest performs. These feedback streams are but the bare minimum. Overall, a failure in any stage is owned by everyone, as is any success. The principle of personal responsibility is frowned upon [15].

In addition to being available on-demand, visualizations of the deployment pipeline's state, the progress of every build, metric graphs and data should be broadcast. It is not uncommon to see dedicated television screens in developer offices showing these charts. There are reports of build status being announced via office speakers using text-to-speech algorithms and light towers similar to traffic lights designed to show the last build's status. While these solutions are perhaps extravagant and borderline comedy, they serve a purpose if they truly assist their users in receiving feedback [15].

Finally, the recurrent theme of discipline presents itself again and applies to the previous section regarding metric tracking. No amount of omnipresent data visuals or red flashing lights will ever cause the situation to improve independently. The process participants need to maintain discipline, listen to the indicators and act upon them as necessary.

Continuous Delivery

A build of the application that has successfully passed the acceptance tests stage and therefore kept its status of a release candidate has fulfilled all requirements that the deployment pipeline can verify. In that capacity, there is no reason not to deliver the version to the customer.

Objections that the delivery may fail, cause data corruption, behave incorrectly or perform poorly could be considered valid. However, it is possible to dismiss their basis by improving the testing process until all stakeholders are content with its scope and introducing a reliable and tested rollback procedure. Once that happens, candidates for the production deployment stage are released to the customer continuously. If an issue occurs, the production environment can be reverted to an earlier build just as easily. This is what constitutes continuous delivery [15].

Whereas the introduction of continuous integration requires consensus primarily among developers and that of continuous deployment mostly happens only with additional cooperation from the infrastructure management team, continuous delivery spans all participants of the software delivery process. However, the benefits gained from the antecedent and already implemented continuous integration and deployment approaches are likely to become a strong motivator that is tangible, as opposed to those promised from the considered continuous delivery.

Many of the utilized integration and deployment principles and thoughts re-manifest. The disadvantages and friction resulting from seldom deployments are addressed by embracing them. Frequent software delivery is mostly incompatible with long-lasting processes, while fast processes must remain effective to obtain approval from any rational management team. Besides that, transforming delivery from a rare event with a significant build-up to an everyday occurrence is linked to reducing stress and burn-out [7]. As before, tooling alone cannot force processes to be continuous as that requires shared effort.

Deployment & Recovery Plans

Even the most scrupulous testing procedure will not account for all factors that weigh in when deploying to production. Therefore, it is imperative to create and maintain a plan determining the actions to be taken following the deployment. Metrics once again play an important role. When compared to the data gathered from having deployed previous versions and considering the estimated impact of delivered changes, significant deviations may imply unexpected behavior. With the help of proper monitoring infrastructure, it is possible to estimate a negative performance impact, e.g., in terms of requests handled per second.

Once an issue is found in production that is deemed too severe to be left present or mitigated using configuration, a need arises for the deployment to be reverted to an older version known to perform well, such as the previous one, so that the provided services are restored to the customer. Two factors greatly influence the complexity of the rollback operation [15].

The first is the data that the application works with. Generally, database schemata are often versioned. Each version is accompanied by upgrade and downgrade scripts, which handle upgrading and downgrading the database in correspondence with the application's model for it. This pattern works for many cases but is unfortunately not strong enough to, e.g., reverse deletion of rows or columns from a relational database system, where a restore from backup becomes necessary.

The second factor is the application's architecture. An application that is ultimately deployed as a monolith makes both directions of the deployment process much easier than these deployed as one or more distributed systems, where dependencies among the individual parts of the system, deployment or startup order might have to be met for it to function correctly.

Naturally, numerous approaches help deal with these constraints. If the scale of the system and its data allow it, the blue-green deployment technique is a good example, as it allows for zero-downtime upgrades and downgrades of the production system [15]. The production environment is made available in two instances designated as green and blue. The green environment contains the last deployed good version of the application, and the blue environment becomes the destination of the production deployment stage. Incoming requests are handled by a router, which normally communicates these to the green environment. Once the blue environment is ready, the router's destination is reconfigured, and the green environment is no longer used to handle requests. In case of detected issues, pointing the router back towards the green environment restores functionality immediately, yet the blue environment can remain available for investigation.

2.3 Configuration Management

The share of applications whose function cannot be configured in any way is meagre. Given that the purpose of configuration is to adjust the exhibited behavior, it is necessary to keep track of expectations of every deployment environment, as expressed by their configuration set for the application. Lack of record of these settings would likely lead to a discord between what the testing suite expects and how the application actually behaves. The recommended approach is to use version control systems to manage configuration just as they are used to manage the source code [15].

This allows everyone — developers, testers and users alike — to reap both primary benefits: to be able to retrieve the latest configuration set for any given environment and to be able to review the changes made between any two points in time. However, an authoritative configuration storage system, combined with a deployment mechanism that applies it reliably, helps steer the whole deployment towards reproducibility. A single deployment procedure applied to two different environments will likely not yield two instances of the application that function equally if the procedure gives too much leeway for non-controlled factors. Non-deterministic deployment results are only welcome in the context of continuous improvement efforts where they should always be investigated, fixed and verified as they undermine the trustworthiness of the integration process and all that follow it.

To elaborate on the previous two paragraphs, various kinds of configuration can be split into several categories, all of which warrant addressing for the aforementioned reasons.

Application Dependencies

Much like the overwhelming majority of applications can be configured, it is fairly rare for them not to depend on another piece of software. To illustrate, even C applications require their standard library, while other ecosystems such as Java, .NET or Python usually require their virtual machines or interpreters to work. Furthermore, this is only the bottom layer, as today's popular principles of code reuse and modularity mean there are often several dependencies on different components and third-party libraries, even in the simplest of applications [15].

The handling of dependencies varies between development ecosystems and often even between their individual build tools. However, no matter if these can, at least to some extent, be handled by native tools or if one has to opt for an external or in-house option, dependency management should aim to meet these goals:

1. **Have the full dependency graph on record.**

   An undesirable change in behavior is just as probable to arise from a change in own source code as it is to propagate from a change in one of the dependencies. Although a robust test suite would, in an ideal scenario, discover these early on, locating the root cause of such a defect could prove difficult when no means of tracking changes of versions of dependencies are available. On the contrary, ensuring that every build operation writes the actual versions of all used dependencies to some kind of registry, or at least to the build log file, constitutes a solid foundation for the resolution of dependency induced bugs.

2. **Explicitly state optimism for accepting changes.**

   Software that serves as a library or a component does not have an immediately apparent feedback channel that, e.g., end-user applications or APIs have, in the form of deployment and subsequent verification steps. Still, assuming the availability of a full dependency graph, it is possible to enumerate all downstream components, i.e., those that depend on the library in question, and initiate their integration process for any new version that has passed all acceptance criteria.

   While the idea of continuously incorporating all upstream updates theoretically ensures the best results possible, granted that most updates improve the behavior or performance, there are other factors to consider. Large dependency graphs may result in perhaps an excessive number of integration cycles whenever a single node cas-

cades its update, [15]. Additionally, some environments may be bound by regulatory or self-imposed requirements such as code auditing scoped to include dependencies which could prove too intensive on human resources, not to mention the computational costs associated with building, testing and deploying these changes. For this purpose, Chaffee suggested an approach of *cautious optimism* which extends edges of the dependency graph to designate whether the integration cascade of upstream changes is desired and prevents failing builds from being pushed downstream [15].

Modern package management systems such as *Maven*, *NuGeT*, *pip* or *NPM* allow developers to specify their software dependencies and retrieve them at the appropriate time, i.e., as part of the build or deployment phases. The desire to accept new versions of dependencies is specified for each dependency as some kind of version interval or according to semantic versioning specification. The former is often implemented as a combination of predicates such as *at-least*, *at-most* or *latest*, which instructs the package manager as to which version of all available ones to retrieve [5].

The latter states that version identifiers shall be composed of the major, minor and patch numbers. Developers are then obligated to adhere to these numbers' semantics, which can be summarized for illustration into the following points [26]:

- When introducing backwards-incompatible changes, the major version number must be incremented, and the remaining must be reset to zero.
- When introducing new features that are backwards compatible, only the minor version number must be incremented, and the patch version number reset to zero.
- When introducing backwards compatible patches for incorrect behavior, only the patch version number must be incremented.

Strict adherence to semantic versioning principles makes it straightforward to estimate the scale of impact arising from integrating an upstream update. For some platforms, automatic tooling for the comparison of API surface exists, which can be used to disprove backwards compatibility implied in a semantic version number. However, their rate of adoption appears very low [5], leaving the responsibility on the developers' shoulders.

3. **Ensure availability of all dependencies.**

   Unfortunately, even having a full backup of the dependency graph, with or without Chaffee's attributes, is not salvatory. This is doubly true when using package managers that reach out onto the Internet to retrieve packages and their metadata. These remote repositories usually uphold some retention policies to prune old versions that are no longer supported, impose rate limits for item retrieval, and their availability may fluctuate or even cease over time. The inability to retrieve the required version of even a single dependency may manifest into failure to reproduce an older build, e.g. in a disaster recovery scenario. However, even normal operation could cause a rate limit to be breached and thereby causing the package manager's central repository to no longer answer requests until a certain cooldown period passes.

   One of the solutions to the missing dependency problem is to check dependencies into source control alongside the source code itself. This could be done twofold: by storing a dependency's source code or compiled output/package. The former approach merely

moves the problem at hand down a level unless the dependency has none on its own. It also increases the build time, which is critical to remain in the previously mentioned acceptable range. The latter may, in time, reach scalability issues, as the underlying source control systems vary in quality and performance when it comes to working with binary files [5].

A second option lies in the fact that package managers often have configurable options regarding the data source, commonly in the form of a remote URL. Many of these back-ends can be self-hosted, and the build process can update this custom repository instance with all dependencies during a post-build operation. Additionally, an artifact repository system, such as *JFrog Artifactory*, can be set up to function as a local cache [16]. Package manager client instances running on developer PCs and build machines are then configured against the self-managed instance, ensuring that any accessed package is made available locally. The apparent trade-offs of having an additional service to manage, especially in relation to disk usage and custom retention policing, mean it is up to each organization to decide if the added value of assured dependency availability is worth the required resources.

Environments

No matter how elaborate a management approach is taken for application configuration and software dependencies, it will still leave too many factors unaddressed. To achieve the desired goal of reproducibility in the maximal possible scope, it is imperative to account for the build and deployment environments in every aspect of the software delivery process. After all, the source code and resultant executable files have no value on their own if there is nowhere to make use of them.

While environments for both building and running the application can definitely be managed ad-hoc, this is an anti-pattern. Environments break. Once that happens to an ad-hoc managed one, the lack of its accurate and up-to-date specification translates into the lack of an inherent approach to take when attempting to restore it to operation. Furthermore, an attempt to recreate the environment instead of finding and eliminating issues with the existing one is bound to fail for the same reason. Disregarding the potential implications of a production environment failing with no way to estimate time to recovery, this approach is antithetical to continuous deployment. Mismanaging the build, testing and production environments in the described manner reliably sabotages all serious attempts to keep them similar to each other or to even continuously deploy the application in a clean environment [15].

To enumerate all attributes that reliably define an environment would be difficult and yield a very lengthy list; however, some are fundamental and should never be left out. It is important to define both software and hardware requirements. As for the latter, the bare minimum would be the count and type of CPUs, with the same applying to expected storage devices, memory, network interface cards and the networks they are connected to. Accounting for the software perspective should always include the operating system and its version and configuration [15]. Any middleware used by the application must also be considered, be it a database server, a message bus or another kind.

The need to manage these environments is likely to arise not only for those who implement continuous deployments and delivery but also for those[1] who provide the infrastructure that the former group builds upon. It is recommended to involve the infrastructure team in the software development process from its beginning. The feedback it is likely to provide will help alleviate incompatibilities of the application with the resources expected to support its delivery by shaping one or the other into conformance. Moreover, the infrastructure staff may already be using some solution and may be willing to share their knowledge. In the same way that the application depends on its environment, the environment depend on the underlying infrastructure. Addressing only one half of the problem will cause the other to render the entire effort ineffective at best.

Deterministic and reproducible deployments may succeed, but it will be impossible to make proper use of them if any part of the infrastructure hinders or prevents these attempts, e.g., a network device misconfiguration, intermittent connectivity loss or the version control system having an unplanned outage. To achieve the determined goal, it is crucial to assure that both aspects are monitored, self-correcting and defined via version-controlled configuration [15]. With respect to the scope of this thesis, only the last of these aspects is discussed further.

Environment Definitions

One can take many approaches when attempting to establish a process that reliably transforms configuration into environments. Barring the thought of a manual process whose qualities were already shown to disqualify it, two essential options remain automated remote installation and virtualization [15].

Automated remote installation is not substantially different from the manual process in the sense that even the latter (hopefully) implies the presence of a list of exact steps to perform when setting up the environment, having started from a fresh installation of the operating system. Automated installation solutions exist for operating systems, no matter if speaking of the UNIX or the Windows family, and the same can be said about the systems' initial configuration [15]. Inputs of these processes take the form of configuration text files whose versioning is not considerably different from source code or other kinds of configuration.

Environment definitions in source control can be subjected to continuous integration. Their updates can trigger a commit stage that performs the defined steps and ultimately results in the host machine meeting the requirements of a build/deployment environment. Then, a naive idea could be to establish a backup of that machine's disk drive and restore that backup to any deployment target computer. This backup can then be treated the same way an application's executables would be, e.g., passed on for acceptance testing or stored in the artifact repository.

The next level in terms of benefits that can be reached once an automated means of environment setup exists is platform virtualization. To quote, *platform virtualization means simulating an entire computer system to run multiple instances of operating systems simultaneously on a single physical machine* [15]. The obvious benefit has already been mentioned in the quoted definition, as a single physical host can be used to run many virtual machines simultaneously and in almost complete isolation. This also removes the

[1] While the principle of approaching infrastructure as a service provided by third parties is a popular one, especially in regards to cloud computing, it fosters the silo culture, described in subsection Team Cognition and Culture of section 3.3.

need to provide a physical machine for every deployed environment. Platform virtualization also allows configuring the specifications of virtual hardware on which the virtual machines run, e.g., computing power or available memory, as long as these values stay within the range of the physical machine. Simulating at the hardware level also means this solution is mostly agnostic to the operating system that is operated on it.

Even when no automated environment setup process exists, virtualization could be used to hide that fact. Given that virtual hard disks are mostly implemented as files and configuration of virtual machines is commonly exportable, this pair of resources can be used to create as many new clones of the environment as needed at the push of a button, while their antecedent processes can be perfected over time. Such a deployment also gains a significant and welcomed assurance of idempotence, provided that the only uncontrolled factors relate to the virtualization technology and the physical machine.

Pipeline Scripting

The last piece missing from the big picture, once the application has its dependencies and configuration versioned properly and a means of provisioning a specific environment to build, test or deploy the application, are the tools that make these three activities happen. While often starting as a trivial invocation of a Makefile, these scripts often grow in time and should be managed in source control [15].

It could be debated whether it does not suffice to version these scripts and their dependencies as part of build environments, like compilers, software development kits and third-party tools with their own configuration and dependencies. The scripts relied upon by all stages of the deployment pipeline should be as similar to those used in production as possible, as all stages following the commit stage are expected to run in a production-like environment [15]. Storing the scripts in a separate repository makes more sense when considering the ease of the use case of developers updating these scripts in their own working environments.

It is recommended to create a dedicated script for each logical step of the deployment process, be it building the executable files, running unit tests or uploading resultant artifacts to their repository. These steps should never be performed by any means other than via these dedicated scripts; the automated pipeline stages should use them in the same manner as developers in their working environments. Their isolation in regards to logical tasks inherently helps with the separation of concerns and clear definition of inputs and outputs [15].

2.4 Impacts on Performance and Team Culture

An extensive questionnaire-based study has looked into connections between organizational processes, structures, information flows, delivery performance and several predictors of happiness of the organizations' members/employees. The study identified continuous integration, configuration management, continuous delivery and source versioning among the drivers that not only correlate with but even predict improvements to any team's *software delivery performance* [7]. This metric has been defined as a composite of the following:

1. **Delivery Lead Time** — the average amount of time required for a submitted changeset to be successfully deployed to a production environment. The design portion of the lead time has been omitted intentionally as the authors have found it too variable

and difficult to measure uniformly. The introduction of this metric has been inspired by the *Lead Time* of the lean theory, which can be successfully applied to software engineering much like to the automotive industry where it originated and can drive comparable benefits [24].

2. **Deployment Frequency** — the average size of the time interval at which (most commonly several) changesets are deployed into production. The study authors have again found inspiration in the lean theory's *Batch Size* and replaced it with a similar, less variable and better measurable alternative. While subjects that have fully adopted continuous delivery will exhibit deployment frequency inversely proportional to their delivery lead time, this component of the metric helps distinguish those who deploy less.

3. **Mean Time To Recovery (MTTR)** — the average amount of time required for a service to be restored to operational status following an incident.

4. **Change Fail Rate** — the rate of production deployments consequential to changesets that have passed all implemented testing procedures yet still cause the application to behave incorrectly and require remediation by an additional follow-up deployment that either reverts the previous changes or fixes the discrepancies in behavior.

Cluster analysis of the collected answers that the study authors have coded yielded three distinct groups of respondents labeled as *high, medium and low performers*. Throughout both iterations of the study in the years 2016 and 2017, high performers have displayed the *best* coded answers that correspond to short delivery lead times, high deployment frequencies, shortest MTTR's and low change fail rates. In 2017, compared to low performers, high performers experienced 440× lower delivery lead times, 46× higher deployment frequency, 170× faster MTTR and 5× lower change failure rate [7]. Additionally, these results support the theorem that software development performance (components 1 and 2) and reliability (components 3 and 4) are not inversely proportional and can be achieved simultaneously.

Furthermore, the study results discovered a predictive relationship between 24 listed qualities and capabilities that, when implemented correctly, predict achievement of generally desirable traits: besides predicting the software delivery performance itself, predict fewer burnouts, problematic deployments, better organizational culture and job satisfaction.

2.5 Existing CI/CD solutions

Most modern continuous integration and deployment/delivery implementations employ an agent model where a server reacts to incoming changeset or another form of trigger and delegates execution of stage scripts onto registered build machines. A software service runs on these machines that is able to accept these delegation requests, execute the job script and report back with the job artifacts. This approach is popular thanks to its abstraction from the underlying infrastructure as it removes the need for developers to manage much of the workload. Moreover, it enables the notion of having dedicated scripts for each step of the pipeline versioned alongside the source code itself [15].

A repeating pattern is that of declarative pipeline configuration. Instead of having a script that represents the entire pipeline, this approach is applied only onto individual stages or jobs that comprise them, while their dependencies on each other and expectations regarding execution order are specified declaratively [6, 18]. This approach makes it easier

to investigate failures of the build system as every step of the pipeline is declared clearly and isolated from others. Additionally, declarative specification of stages and jobs allows the build system to parallelize the pipeline's execution as appropriate with respect to the interdependencies and to the availability of resources at the time of execution.

Naturally, the available solutions vary in licensing/pricing and deployment options. The first solution described below, Jenkins, is available for free and under an open-source license and is most commonly deployed on-site and provides a comprehensive suite of features to execute scripts of any kind in reaction to commits, with little regard for their actual purpose, be it building, testing, deployment or delivery [18]. The second solution, GitLab, additionally provides hosting and management of Git repositories and enables connecting CI workloads [6]. Its editions are available for on-site deployment but are also offered as a hosted solution for teams that do not wish to manage their own instance. Both cases vary by pricing and availability of features [6]. Finally, there are solutions similar to Jenkins in the sense that their primary purpose is that of a CI build system, but are offerred as managed services, often with a free tier, such as *Travis CI* or *CircleCI*. These are commonly used in open-source communities like GitHub's, but their characteristics are so similar to the two aforementioned solutions that they are not described in detail [30].

Jenkins

Jenkins was found as the most popular CI/CD solution based on a literature study [28]. Its second version brings official support for the declarative pipelines, but the former version's support for imperatively scripted ones is not discontinued. Among others, improvements have been made to the means of organizing jobs — the former centerpiece — into pipelines as continuous integration and deployment became more prevalent. The declarative approach constitutes a shift from performing most of the configuration in Jenkins' web interface towards declaring them in a special *Jenkinsfile* that is more easily version controlled [18], in compliance with the tenets of configuration management.

Alongside the primary machine used to run Jenkins itself, the system is most commonly used in the distributed mode, where the master node is accompanied by agent nodes — systems that have been made available to accept and run incoming jobs. Every node is actually configured with the desired number of executors, all of which handle incoming jobs, allowing for nodes to work on more than 1 job at a given time [18].

A variety of options can be used to ensure the start of configured pipelines or jobs, spanning the polling of version control repositories, periodic timers, time schedules, a cascade of another successful job and including setup of a so-called webhook HTTP URL, access to which by any upstream system will act as a trigger [18].

Artifacts produced from within Jenkins jobs can be managed using multiple natively supported options, one of which is the already mentioned JFrog Artifactory. Filename patterns are often used to differentiate artifacts of importance from intermediate outputs and can then be stored in a dedicated artifact repository or on a designated file server [18].

Jenkins did not fall behind the rising trend of using containers in conjunction with continuous integration and delivery. The most common containerization technology used in this context is Docker [30], for which Jenkins offers first-class integration tools, essentially allowing job executors' implementation in the form of containers [18].

The status of all pipelines and other features is accessible through the web interface in the forms of dashboards. This excerpt of features, the system's open-source license and plugin extensibility provide solid reasoning for its popularity.

GitLab CI

GitLab is a product that started as a Git repository management tool and subsequently grew to accommodate various workflows related to Agile development [6]. Repository hosting and management in combination with features that support continuous integration and delivery make GitLab a candidate for an all-in-one solution to the entire software delivery process. However, integration tools are available that allow delegating some responsibilities onto other services in favor of the built-in features, such as for using Jenkins for continuous integration [6].

In terms of GitLab CI — GitLab's own implementation of job system — the overall architecture is in many ways similar to that of Jenkins. Agent software called *GitLab Runner* runs as a daemon on a set of build machines. It maintains a registration to the GitLab application server, which is then, in turn, able to orchestrate job execution via the means of individual runners. A wide variety of runner software exists, ranging from native executors for both Windows and Linux to those that integrate with virtualization software like VirtualBox, containerization software like Docker or even container orchestration software like Kubernetes [6].

GitLab CI pipelines are also configured declaratively. A special file *.gitlab-ci.yml* is expected to exist in the root folder of all repositories that wish to opt-in to continuous integration features. It specifies most aspects of the pipeline: triggers, stages, individual jobs and others. The contents of this file direct the CI system's behavior. The Git revision that triggers a build, e.g. via changeset delivery or a branch name in case of a timed trigger, always contains its own configuration for the CI system [6].

GitLab is available in two editions: community and enterprise. The former is distributed under the MIT license, while the latter is offered as a proprietary product richer in features than its open-source flavor [6].

Chapter 3

Versioning of Continuously Integrated Source Code

Collaborating on source code is difficult without proper tooling that tracks history in addition to the most recent state. Reconciling multiple changes made to a single source code file would first require determining what changes were made, then determining the goal of these changes and finally finding out how to merge these together while retaining their intended functionality. The mechanism of code versioning systems could be illustrated as maintaining a directed graph where nodes represent a significant state of all contained files and directories. Edges of this graph would then represent sets of changes delivered by developers, causing the state of the contents to transition into another. The term *repository* is generally used in software development to refer to the entirety of this graph, sometimes including the system used for its operation.

Version control systems introduce the concept of branching, where at a certain point, two or more new states of the repository are created based on a common origin state. This feature allows for development to continue on every such branch in a versioned manner, all the while permitting these branch versions to be changed independently of the others. The first major advantage of this approach is that despite having common history, work on both so-called branches can continue, and changes made to either branch do not affect the other. The second advantage is that this approach allows one to maintain a practically unlimited number of logical repositories with isolated histories to coexist in a single physical repository or system.

The decisions made when choosing a version control system to use and when defining which branching and merging strategies to follow have serious implications on the software development process, especially in regards to continuous integration, and will be described more closely in the rest of this chapter.

3.1 Version Control Systems

The very first version control systems that emerged had no notion of networking and worked exclusively on local file systems. Additionally, the concept of pessimistic synchronization was a widely used one, where for every object (file or directory) that someone wished to change, they first had to acquire its exclusive lock. The unit of atomicity was originally that of each contained file, as was the granularity of change tracking. Every file had its own distinct history graph [15].

For many years after its release in 1988, the *Concurrent Versions System* (CVS) has been the most popular and used version control system, as no real alternatives have been available free of charge at the time. Many of its aspects are now considered disadvantageous, such as the loss of file history for renamed files, stemming from the already described file-based approach which the system inherited from its predecessor. However, it pioneered other features that are now part of every modern versioning system, namely support for the client-server architecture and the introduction of optimistic concurrency [15].

Revisions as the Unit of Atomicity

Subversion was created to be a better version of the CVS and is still one of the systems in use today. A majority of its features and improvements arise from no longer having all operations atomic at the level of every file, but instead per revision, which is Subversion's preferred alternative for a changeset, used throughout this text to refer to a set of changes that a developer wishes to deliver to the repository. This delivery operation is commonly called making a *commit*, and it is not uncommon to see these three terms being used interchangeably.

Although there were many benefits, especially in relation to performance over CVS, changesets becoming the unit of atomicity was the most important. It constituted redemption from the world of Subversion's predecessors, which lacked this guarantee, where the delivery process more often than not resulted in an inconsistent state. Merge conflicts used to happen at the level of individual files. Therefore, submission of changes could end up with some files updated to the proposed state and some failing to merge conflicts. Their resolution required manual correction and re-submission, while a third party would see the repository in a nonsensical state while this effort was underway. Subversion put an end to that behavior, as only the changeset as a whole could be rejected, helping ensure that the state of the repository remains valid.

A further consequence is that it becomes easier to analyze any given changeset in a Subversion repository, as files are treated like their children instead of parents. All changes made across the repository are aggregated into a single changeset identified by an increasing integral revision number. It is no longer necessary to use another mechanism for correlating contributions in various parts of the whole.

Distributed Version Control Systems

Subversion represents a centralized version control system, where an authoritative server always holds the most up-to-date instance of the history graph. Contributors operate in the role of clients, communicating with the server to access or manipulate the history of the repository. While this allows for easy access control, it mandates that every project hosts and maintains this server copy. In the case of Subversion, the commit process and most other operations require that connectivity to the server is available. As a result, it is not possible to work offline [15].

Recent years have seen an uptick in the popularity of distributed versioning systems such as Git and Mercurial. Their distributed nature abolishes the need for an authoritative primary server handling synchronization and record keeping and the related pattern of having an incomplete client-side working copy that only reflects a limited view of the repository's history. Every repository is a full-featured instance that can be operated autonomously. While a corporate environment will likely have an instance of the repository

designated as primary, used by employees for the purposes of work delivery, this is done merely by convention and has no foundation in the used version control system.

The distributed nature is well suited for and commonly used in the development of open-source projects. A contributor does not typically submit their changes directly to the upstream repository. Instead, they clone it to obtain their own writable working copy of the project's source code. The means of working in temporary teams that are reshaped as volunteers gain and lose interest in contributing is well documented by the Linux kernel, whose versioning needs eventually led to the creation of Git. Proposed changes were submitted to the project's maintainer in the form of patch files that summarized what changes were made to which parts of which files. The maintainer then had the discretion of accepting the proposal, asking for additional changes or rejecting it. It is not unheard of for projects to diverge into two versions over a major disagreement to a particular set of changes, which is made significantly easier by using a distributed version control system [15].

Nowadays, community sites that enable free hosting of Git repositories commonly support the creation of so-called *forks* and submitting of *merge requests* or *pull requests*. Forks are the aforementioned writable copies of repositories. Merge requests and pull requests are a feature that commonly provides a space for discussion of the proposed changes, but ultimately, when the request is accepted, leads to a merge of the branch that hosts the proposed changes into another; this automates the step of composing the list of changes and sending them to the maintainers.

In regards to drawbacks coupled with this kind of versioning systems, a commonly mentioned one lies in the basic principle of every contributor obtaining a full in-depth copy of the entire repository. In addition to the most recent state of the source code, all previous are accessible as well. When, e.g., access credentials or other sensitive information are accidentally checked into a distributed repository, there is no way to remove them from the local copies of individual contributors. Others appear as a result of the distributed character, such as that auditing and enforcement of policies is much more difficult than in a centralized system. However, little effort is required to use, e.g. Git with a conventionally designated primary server which oversees the implementation of all necessary policies and additionally sets off the deployment pipeline.

3.2 Branching and Merging Strategies

As has already been explained, continuous integration mandates for all developers to submit and integrate their changes no less than once a working day. Prolongation of the delay before merging the changes back to the originating branch is associated with numerous drawbacks, many of which stem from not having the changes integrated. This malpractice means there is a lack of feedback in regards to architectural differences and dependency clashes that would otherwise be discovered. Furthermore, the longer a branch remains divergent from the main one, the more changes are likely to accrue on both in regards to continuous integration. That, in turn, only exacerbates the unavoidable unpleasantries bound to happen in the form of merge conflicts.

Moreover, regardless of whether or not continuous integration is in place, having a large number of branches simultaneously alive in a repository that designates one of them as primary is generally ineffective. Whenever a branch is merged with the main one, the other branches usually become obligated to accept these changes. If that were not the fact, these contributions could render the main branch inoperable at the very least, as the changes' interoperability with the main version could not be tested prior to the merge.

Mainline Development

When considering which strategy to take when taking on continuous integration, the starting point should be that everyone shares a single main branch to which they deliver their changes frequently and then take on stabilization efforts so that the deployment pipeline passes, which they consider a part of their responsibility. This strategy is called the *mainline development* and means a single branch is designated as the main branch. Efforts from most development activities should eventually be merged into the main branch. Therefore, the use of branches is not forbidden per se but is discouraged for regular development.

It is clear that this strategy closely follows the definition of continuous integration and should therefore come with its benefits, including more frequent and more comprehensive feedback. However, deliveries to the main branch will unavoidably result in failing builds, breaching a major principle of continuous deployment that would expect all builds to be passing.

In order for the recurrent integrations to remain bearable, the amount of submitted changes needs to be reasonably small, and their character should not have too large of an impact. This emphasises the importance of the design process: all tasks are suddenly required to be split to meet this requirement. At the very least, the number of integration-complicating deliveries must be kept as low as possible.

Concentrating the entire development team on a single mainline branch is likely to encounter issues when scaling unless the application is well designed into components with proper boundaries. While not always the case, it is customary that support is provided for certain releases of a certain age. As only the single mainline branch is recognized as a relevant origin of source code, usage of techniques such as feature hiding is required. These, in turn, have to be configured, risking an increase in complexity that has to be addressed during design.

Branching by Abstraction

The problems connected with introductions of major changes can be alleviated by designing and implementing an abstraction layer that resides upon the functionality that is the subject of replacement. The remaining portion of the application is adjusted to use this layer exclusively when accessing the underlying functionality. Deployments are configured to access the legacy implementation while the replacement is being worked on at an arbitrary pace. Once completed, deployments are reconfigured to no longer access the legacy implementation, which can then be removed, perhaps along with the abstraction layer if it is no longer deemed necessary.

The above principle is noticeably not so much a strategy for branching as against it, but provides an illustration as to how to reconcile complex changes with the mainline strategy. Interestingly, it is also applicable when incorporating continuous integration into existing projects. The share of code not participating in all deployment pipeline stages would decrease in time as components are abstracted away and rewritten.

Nevertheless, situations may emerge where creating an abstraction layer is impossible due to technological constraints or tight coupling between components. Only cases like these should warrant using an actual branch, figuratively rewinding time back to before the adoption of continuous integration complete with merge and integration conflicts.

Branching by Releases

Where delivering software is not done continuously, an exception exists for the creation of long-lived branches. Notwithstanding the exact nature of processes in place to support the delivery, it is safe to assume that these need access to that version of artifacts that was designated for delivery. On the other hand, an insignificant segment of the delivery team will intend to work on other tasks, dutifully integrating these onto the mainline. This dissent has formed the strategy of creating a branch off the mainline whenever a release of particular importance is done.

This strategy constitutes another procedure that is not in violation of the mainline strategy but merely its complement, as no intensive development is permitted to happen on release branches, save for fixes of critical software defects, which are then merged back into the mainline. Merges are not permitted to happen in the opposite direction; situations that warrant these instead take the form of another release branch.

Branching by Features

From a certain size of an organization, having all developers deliver to a single branch in accordance with the mainline strategy could later require significant effort to reintegrate a build to a passing status. At a certain point, addressing the required effort may be deemed more important than the wish to unconditionally adhere to known best practices. Alternatively, the desire to ensure that the mainline always results in an always passing build may overtake the former.

The foundation of this strategy lies in establishing a branch for every feature that is worked on. This is best compatible with iterative software management methods that typically imply such features are developed in cycles spanning at most days or a few weeks. These branches spike out from a certain state of the mainline branch, and all work related to the feature is done on them, while the mainline remains in its original — presumably passing — state. It is crucial for all feature branches to accept mainline changes daily to avoid too large of a drift-off. Once the feature is deemed ready, testers can assess it while it still resides on its own branch, putting in place an additional safeguard against polluting the mainline.

Once the feature is done and merged, its comprising changes are often collapsed into one changeset. Each node on the mainline's history graph semantically represents a completed feature or a bug patch. The notion of not sharing unfinished features via the mainline with others and potentially with the customers, as is the case with the mainline strategy, is appealing to some as well.

The disadvantages are similar to what has already been described earlier. Adding to the inevitable merge conflicts that will require resolution, varying in spent time and effort, enforcing this strategy requires that all branches deemed merge-eligible are reviewed by someone knowledgeable enough in the project to assess all potential implications.

3.3 Repository Models

Although features of the utilized version control system play a significant role in shaping the parent software delivery process, the VCS is ultimately a tool whose usage may vary. While the software engineering industry is familiar with software design patterns and architecture, their proliferation to the field of version control is yet to come [2].

A not insignificant quantity of literature exists that discusses one or more possible approaches to scaling version-controlled repositories. In essence, there appear to be two schools of thought. One advocates for storing everything in one repository while the other supports having multiple as found necessary. Unfortunately, most of these sources are partial and only deal with definitions, benefits, and drawbacks of either of these approaches. Moreover, they are often based on publications whose authors describe how either approach enabled their corporate employer to reap most benefits as opposed to how they have been handling things earlier on [3]. Objective comparisons are a rare find and most commonly take the form of gray literature or literature reviews [2, 3].

Despite the sheer number of sources, there appears to be a general consensus in terms of the problem's definition [3]: What challenges arise when managing vast source codes of increasing complexity via version control systems, and how is it possible to address them?

Monorepo Model

The monorepo model is by large a formalization of the existing default approach to source code versioning. As the name suggests, it entails operating a single shared repository used by all developers in an organization, emphasising shared tooling, dependencies and ease of access to any project's sources [17].

A more detailed viewpoint then further classifies existing monorepos by size and content diversity into *project monorepos* and *monstrous monorepos*. The former encompasses repositories dedicated to a single software project that is rich in modules or components, while the latter describes the approach taken by numerous large organizations aiming to concentrate all of their source code in a single repository, spanning multiple projects that are not necessarily dependent or even related to each other [2].

Barring the fundamental characteristic of centralization, the following traits are also consistently attributed to monorepos: [2, 17]

- **Standardization** — the set of tools used by the individual stages of the deployment pipeline is easy to enforce as only one central source of truth subject to auditing exists. Attempts to introduce new tools, or adjust the behavior of existing ones, could easily be noticed and thwarted by the review processes, which only have to account for a single repository.

- **Visibility** — stemming from the aforementioned characteristic of centralization, every member of the organization is typically granted read access to all parts of the repository. This allows everyone to find up-to-date usage samples for any internal or external APIs. At least in the case of internal interfaces, when introducing backwards-incompatible changes, it is trivial to look up and correct broken usages, as they can all be found in one place.

- **Synchronization** — the repository model supports the adoption of the mainline development strategy with all its properties and benefits.

- **Completeness** — following the notion that monorepos are expected to contain projects in their entirety, the source code that represents them could easily be built at any time, granted that all dependencies are accounted for, regardless of the means thereof. Content retrieved from the repository can present complete input information for the deployment pipeline.

Multirepo Model

On the opposite side of the spectrum lies the multirepo model. Where the monorepo model revolves around having a single central repository, the multirepo model encourages having multiple [17]. Alternative definitions exist and differ in the detail of cardinality, such as that every system component or library should reside in its own source code repository [3].

Due to the lack of sources in regards to attributes of multirepos, the following proposals are based on the attributes of monorepos and adjusted to reflect multirepos' handling of the same areas:

- **Flexibility** — while a certain set of rules will always need to be present to assure a basic level of compatibility among individual multirepos, developers working on a project can utilize significant freedom of customizability [17].
- **Weak Consistency** — developers can no longer contribute to a mainline as the approach is perpendicular to the thought. No single mainline exists, except by convention, e.g. in a repository responsible for assembly and delivery of the whole application.
- **Encapsulation** — each multirepo intentionally contains just a scoped subset of the entire application. Typically, each multirepo features its own variant of the deployment pipeline with at least the commit stage and some extent of acceptance testing.

Comparison of the Qualities of the Monorepo and Multirepo Models

It is difficult to formalize the advantages and disadvantages related to the individual repository models, even if such claims are readily available in the cited literature. Although not disputing the benefits being ascribed to them, they can often be interpreted in a way that discards their exclusivity to the particular model. Other times, a simple solution can be proposed that serves as a counter-example to the exclusivity. The monorepo model arguably finds itself better positioned in that particular aspect, requiring no additional work [17]. However, certain mentioned qualities are a matter of preference, and different parties could perceive them conversely.

A fitting summary for the qualities below is that a clear-cut decision is impossible to make and lies in deciding whether the benefits of a particular model outweigh its drawbacks [17].

Ability to Scale

The aspect of scale is very much the only one that is unquestionable and uncontentious. All of the three major corporations that declare to be using monorepos internally — Google, Facebook and Microsoft [3] — have reached points at which their version control system of choice struggled to keep up with their software delivery process.

Google currently uses a custom in-house versioning system called *Piper* that provides a hybrid solution that has a centralized mainline but supports local clones in a manner not dissimilar to Git [25]. Facebook contributed to Mercurial (a distributed version control system), whose performance adjustments and extensions allow it to support the high-volume workloads of the company in its monorepo [13]. Finally, Microsoft uses their fork of Git to version their monolithic codebase, which they have extended to allow for file virtualization that allows on-demand retrieval of data objects. Obtaining a clone of the repository with

Commit Count	250 000+
Daily Push Count	8 421
Daily PR Count	2 500
Active Branches	4 352

Repository Clone Size	300 GB
File Count	3 500 000+
Changes per PR	tens of thousands
Conflicts per PR	thousands

Table 3.1: Statistics regarding Microsoft's monorepo with Windows source code, collected over the first 4 months since switching to their modified fork of Git [14].

this feature enabled then does not need to download the gigabytes worth of history [14]. The level of scale at which the systems no longer function with acceptable performance is illustrated with Microsoft's data in the table 3.1.

Splitting their monolithic source code repository into many multirepos is an approach that could, in the first place, remove the need to scale at all. The repositories could be split into parts small enough for the underlying version control system to perform efficiently.

Visibility & Velocity

A mixed-method study was conducted at Google, looking into the benefits of monorepos and multirepos as believed by the engineers working at the company. The results published in 2018 have shown that for study participants with prior commercial multirepo experience, visibility offered by Google's monolithic codebase was the most important benefit. Its high regard was also prevalent in the remaining groups of participants, ranked 2nd by participants with only open-source experience with multirepos [17].

The study's coding translates visibility to the ability to easily search through the codebase. Furthermore, it was an enabling factor for three additional high ranking benefits as believed by participants. Their definitions, taken from [17], follow:

- Code Reuse — looking up documentation/implementation of APIs
- Usage Examples — looking up examples of how to use APIs
- Easy Updates — migrating clients of an API to the latest version

Moreover, participants of the study cited visibility as a factor of their development velocity in the context of monorepos. They believed the offering of visibility in conjunction with the three use cases mentioned above enabled them to perform faster than they would in a different setting [17].

However, the study admits on its own that their *"survey results may have a major confounding factor from Google's internal developer tools"* and that *"engineers may have rated the monolithic codebase highly when, in fact, it was the developer tools that they had a strong preference for"* [17]. Its results also show that while the monorepo model has received praise for reducing the cognitive load (the difficulty of understanding code) as a result of visibility and the internal developer tools, the same was attributed to multirepos as a result of the smaller size of the source codebase [17].

Additionally, a different source authored by Google engineers cites *"code complexity, including (...) difficulties with code discovery"* as a disadvantage of the monorepo model. Noted is also the need to continuously maintain and develop these tools to keep up with the company's pace [25].

Among the benefits of visibility is also included that of understandable layout and ease of access, arguing that having the entire source code structured in directories of a single

repository simplifies understanding the big picture of how the application works and how its components interact [25]. However, hosting solutions commonly support a form of nesting projects in namespaces that offer a very similar feature even for multirepos, such as so-called project groups, in the GitLab server case [4]. It is also found beneficial that file at any path can be used as their unique identifier, [25], which is also not exclusive to monorepos, as long as the path includes that of the parent repository.

Team Cognition and Culture

A non-technical aspect that may affect how well will an organization respond to adopting either repository model is its culture. One of the sources even goes as far as to suggest that a part of the models' alleged technical benefits is fallacious, merely changing the approach taken to individual issues [3] instead of actually resolving them.

In support of this theory, various adopters of monorepos proclaim philosophies of unity — *"It's best to do one thing really, really well"* in case of Google and *"We Are One Team"* in case of Twitter — while adopters of multirepos express near opposites: Netflix reportedly favors culture of *"freedom and responsibility that empowers engineers to craft solutions using whatever tools they feel are best suited to the tasks"* and Amazon reportedly goes to such lengths where multiple teams work on the same projects and compete against each other [3].

Amazon's culture constitutes a textbook example of the *silo culture* where departments close themselves off in silos, often not only figuratively, and refuse to deal with issues that are not cut clear respectively to their official duties. All the while, the presence of silo cultures has been shown to negatively impact team productivity and other performance metrics [3, 7].

On the contrary, adopters of the monorepo strategy report positive impacts thereof towards team cognition — *"a critical mechanism for facilitating knowledge activities within the software development process"* [3] — suggesting that the repository models may have vast consequences on how teams communicate and calling for a study of this implied correlation, which has not yet been published [3].

Dependency Management

The coexistence of all projects next to each other in a monorepo simplifies the management of dependencies. Maintaining dependencies checked into the repository, regardless if done in the form of source code or binary consumables, is a fairly reliable method of ensuring that only one version of a dependency is in use at any time. This helps reduce the possibility of a diamond dependency conflict occurring, as illustrated in the figure 3.1 [25]. An application whose sources are kept in distinct multirepos but is ultimately deployed in a monolithic manner is likely to require additional care to detect these conflicts prior to deployment.

However, for distributed applications, the aforementioned method of dependency management can actually comprise a limitation — services running on separate hosts could be affected only by their direct dependencies. As such, a diamond dependency detection algorithm would fail their deployment pipeline without a reasonable basis.

Furthermore, the ease of adding dependencies is cited as a disadvantage of the monorepo model. Every introduced dependency adds to the risk of downstream breakages and typically increases the build time. On the contrary, more complex dependency management leads to a more careful design of APIs and a better thought-out dependency graph. All errors are harder to rectify later on [2].

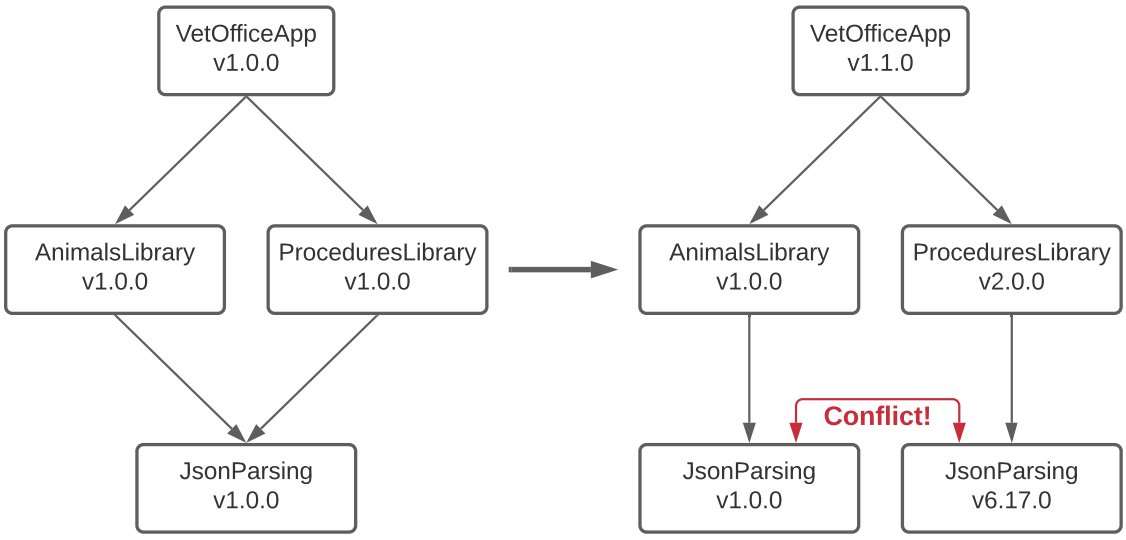

Figure 3.1: In many scenarios, typically when an application is deployed as a monolith, it is not possible for two versions of the same executable application or shared library to coexist. Instead, steps must be taken to ensure the diamond pattern's integrity, especially if developing in a multirepo environment, where this issue may only manifest once the entire application is assembled or deployed to a testing environment.

In a monorepo code base, contributors that introduce backwards-incompatible changes to a library are expected to migrate all callers [2]. A sample of engineers at Google, when asked about this practice, had conflicting opinions. Some were happy not to have to deal with this process on their own, while others did not appreciate the code under their hands changing constantly [17]. Additionally, they appreciated multirepos' control over the dependency set and over the release frequency [17].

Development Tooling

The quality of tooling that is provided to developers to aid them in the software delivery process may be of equal importance as the repository model of choice [17]. Taking into consideration that the monorepo model's support has de facto been the goal of the first version control systems, an absolute majority of existing toolchains either comes with first-class VCS integration for monorepos or can be used all the same, provided that the sources are first made available via the local file system.

However, starting at a certain scale of the repository, the performance of these tools may begin degrading, as was the case for the underlying version control systems. IDE features like code indexing, and code completion may be unable to cope with the number of source files, warranting that these tools are either adjusted or replaced with custom, sufficiently performant alternatives [25].

Having source code split into multirepos does not entirely address this issue. While traditional tools are more likely to function satisfyingly on segments of the whole code base, additional work may be required so that their usefulness is not limited to these particular segments. Continuing with the code completion example, the feature would be significantly

lacking in convenience if it was unable to offer suggestions based on other multirepos. At that point, the issue of scale re-manifests.

Immediate vs. Eventual Consistency

Given that their characteristic of synchronization grants monorepos the guarantee of atomicity, even large-scale refactorings are in theory deliverable at once. In a single commit, a breaking change can be introduced and all impacted callers updated, along with the respective test suites, as all of these reside in a single repository [2].

Performing the same large-scale refactoring on a multirepo application would be different in many aspects, all stemming from the fact that commits are atomic at most on the level of individual repositories, depending on the underlying VCS.

Where the properties of monorepos enable delivering a single commit encompassing all changes, the multirepo model makes this much more difficult if each repository has its own deployment pipeline. In order to ensure the changes' full propagation, a dependency graph spanning all affected repositories must be established. Changes must then be submitted in an order that begins with repositories whose nodes have no predecessors, then cascades onto their successors, and repeats until no unvisited nodes remain.

Considering that every change delivery triggers a separate deployment pipeline, the delivery of changes that span a significant number of repositories is likely to take a respectively significant amount of time. While the global delivery is in progress, the affected repositories may also accept changes from other sources, causing merge conflicts that may require additional resolution.

While the aforementioned complexity is undeniable, with it comes the possible benefit of specific dependency versioning that, in a sense, builds on the fact that every component must keep exact versions of its dependencies on record. So long as an exactly specified version of the dependency is available for retrieval, it makes no difference that its mainline version — the most recent stable version — contains changes that may be incompatible with the rest of the system. This gives way for a diamond dependency conflict to occur, but as already mentioned in Dependency Management, some application architectures are resistant to it.

Chapter 4

Comfort-Inhibiting Factors of Multirepo Development

Software development carried out in a multirepo environment is likely to be affected by at least part of the aspects detailed in the section Comparison of the Qualities of the Monorepo and Multirepo Models found in the previous chapter. As mentioned previously, most of the available literature merely compares the two repository models, [3] while no peer-reviewed sources detailing multirepo or its operation are available.

The assignment of this thesis was developed in cooperation with a partner company. One of their development teams pursues establishing a novel, internal developer ecosystem (hereafter within this chapter referred to as *the ecosystem*). This effort entails the creation of an infrastructure library that would enable developers to write plain programs uncluttered by the underlying infrastructure's internals and technologies. There were many motivators that all eventually converged in this team's mission, and their enumeration here would be difficult. The goals are, however, much more clear and explicable — at least the few mentioned below, believed by the author to be most relevant.

The new ecosystem under development strives to integrate continuously. Many related software projects, whose components are planned for gradual adoption, have suffered from painful integration periods not entirely dissimilar to merging hell, where the delivery pipeline that follows the commit stage (and sometimes also the commit stage itself), or the prerequisite merging of changes, are prone to time-consuming or effort-consuming processes whose timelines breach those recommended when integrating continuously [15].

Furthermore, as the number of technologies involved in the related projects grew in time, so did the collection of support tooling responsible for the projects' operation. While working reasonably well, many of these tools were created in-house. Even though some have public origins, most of them are not popular enough to grant the benefit of new employees being familiar or at least aware of their existence. This led many software engineers to ponder: Are we really the only software development company that has to tackle these issues? What tools are used by others; is it possible to overcome these issues differently, without any additional tooling? This was the basis for the second goal: utilising proven, open-source software wherever applicable.

Last but not least, there was a desire to achieve a more resilient versioning model, allowing for assembly of the application from its modules in various versions while avoiding the need to rebuild them on every occasion of that process. While conditional builds and the caching of executable output thereof is possible with monorepos [25], applying this

strategy on individual multirepos — as opposed to directory sub-trees of a monorepo — has no additional setup and doubles as a clear boundary of change detection.

4.1 Identification

Based on the personal experience of the author gained while participating in the ecosystem's development, the following list of routinely performed developer actions has been composed. These activities portray many of the disadvantages of the multirepo model described in earlier chapters.

1. **Navigating a hierarchy of multirepos** — in the current form, some of the ecosystem's architectural components, which form a single logical unit, actually reside in several interdependent multirepos.

   To illustrate, one repository could maintain a definition of its interfaces, depended upon by its users. The purpose of a separate repository would then be to hold code relevant to the component's configuration. The implementation repository, depending on both previous ones, would contain the business logic and low-level tests. Finally, the last repository would be used to manage the integration test suite.

   Based on the applied degree of componentization, a significant number of repositories may be created, such as four in the previous example. While this may seem small on its own, there is not really an upper boundary of components that a system can have. Even in a system comprising of 20 components, designing and managing a hierarchy of 80 repositories so that they are easy to access might prove challenging.

2. **Establishing local repository clones** — despite all measures that an organization may take in order to avoid the creation of functional silos, an anti-pattern to performance described in the chapter 3.3 (Team Cognition and Culture), software engineers may still tend to prefer working on a particular subset of components. Consequently, when such an engineer one day gets to work in an area that they previously have not contributed to, they may first need to establish local clones of the repositories relevant to their task.

   For this reason, cloning a repository is not a one-time action that a software engineer performs as part of their onboarding process but potentially an everyday process.

3. **Working within local repository clones** — once the relevant clones are established and potentially updated so that their state matches that of the designated central repository, the software engineer is able to start working on the task they have been assigned to.

   The issues of scale mentioned in chapter 3.3 (Ability to Scale) apply here. To reiterate: in a monorepo, standard tooling (e.g. code auto-completion, editor highlighting, static analysis tools) would likely work correctly until the project in question was too large on its own or had too many dependencies. Conversely, multirepos may, in a sense, encounter opposite issues, as the projects they contain typically cannot refer to each other via local file system and configuring these tools to understand this indirect dependency may prove difficult.

4. **Pushing changes from local repository clones** — once the changes required to accomplish the task at hand are done, it is necessary to deliver them back to the cen-

tral repository to begin the integration process and the subsequent delivery pipeline. However, the change submissions may be sensitive to timing.

Continuing from the example of 4 multirepos described as part of a workflow 1, imagine a task that involves adding a configuration option changing the component's behavior. While the interface repository would probably see no changes, the others would — the configuration repository would require that the new option is added, the business logic repository would need to implement its handling, and a new test case for the integration test repository would likely be warranted as well.

If all of these changes were submitted simultaneously, a race condition would occur, allowing the possibility of one or more of these repositories being built based on an outdated (or missing, depending on implementation details) version of their dependency.

5. **Full-text search through source code contained in repositories** — when investigating defects or non-standard behavior of any software application, a commonly used technique is that of correlating captured diagnostic messages contained in log files to the code path in an attempt to trace the program's execution, so as to locate the faulty unit of code.

   Application of this technique in a multirepo environment may be more difficult than in a monorepo, as developers generally have no incentive to maintain up-to-date local copies of all repositories that would allow traditional search tools to be used.

6. **Updating source code to accept upstream updates** — this workflow entails updating versions of a component's upstream dependencies or updating pipeline definitions to a new version when they become available. The ecosystem provides predefined CI pipeline templates for every officially-supported programming language, which are then in turn merely included by individual projects using their name and version number. It thus can be treated in a very similar manner to application dependencies.

7. **Ensuring that own changes are cascaded to downstream dependencies** — there are situations that warrant traversing the downstream portion of a component's dependency graph and immediately ensuring that a new version is used, or, perhaps, that a certain previous version is not.

Additionally, a basic set of user comfort indicators was established to allow for their analysis in regards to the above workflows:

1. **Feedback Time** — the amount of time the provision of feedback takes once a step of the workflow is performed.

2. **Complexity** — affected by the interdependency of individual steps; the number of steps it takes to complete the workflow.

3. **Mundanity** — affected by the number of steps it takes to complete the workflow when the steps themselves are of simple nature, but numerous; repetitiveness of steps.

4. **Context Switching** — the need to switch contexts while performing the workflow, caused either due to conceptual variance in steps or due to their completion time giving space to switch contexts and work on other tasks in the meanwhile.

A user study has been conducted with aims to identify workflows that are believed most inconvenient and to identify which of the indicators are a believed cause of that characteristic in the eyes of the software engineers who perform these workflows daily. In summary, management of dependencies (both workflows 6 and 7) ranked as the two most inconvenient, followed by fulltext search through multirepos (workflow 5).

The form used to conduct the study is available in the appendix,A and the results are available in appendix B.

4.2 Existing Solutions

Regarding the results of the study outlined in the table B.2, several conclusions can be drawn as to what factors the study participants think are behind the inconvenience of some of the workflows. In particular, data for this question are available for the following workflows:

1. **Ensuring that own changes are cascaded to downstream dependencies** — all participants that ranked this issue as most inconvenient or uncomfortable attributed this to the workflow's complexity. Its mundanity ranked second, and the need to switch contexts ranked last.
2. **Updating source code to accept upstream updates** — both participants that ranked this issue first attributed this to the workflow's complexity. At the same time, one of them believed that feedback time, mundanity and the need to switch contexts also had a negative effect. Additionally, one of the participants recorded that the workflow is prone to errors.
3. **Full-text search through source code contained in repositories** — the participant that ranked this issue first attributed this to the workflow's complexity and mundanity, again with equal importance for both factors.
4. **Working within local repository clones** — both participants that ranked this issue first attributed this to the workflow's complexity and the need to switch contexts, expressing equal importance for both factors.

Various sources have been analyzed in efforts to find at least partial solutions to possible root causes of the aforementioned factors that have a negative effect on the workflows listed above. The remainder of this chapter discusses these discovered solutions, their applicability in the case of the development team of the ecosystem, and a general case where applicable.

Git submodules

Git repositories come with a native feature called *submodules* that allows for the inclusion of a repository in another. Once a directory is designated as the submodule's root directory, a remote URL is provided along with an optional revision identifier, the included repository can be accessed and manipulated. This is a significant advantage as opposed to if the submodule repository contents were copied and pasted into the designated folder. Submodules are recognized as full-fledged Git repositories with all relevant features, such as change tracking or pulling and pushing changes from/to the included repository [4].

It is important to mention that when initialized with a submodule, the parent repository does not retain anything more than the root directory path, the upstream repository's URL

and the revision identifier. This, among others, means that their history graphs are kept separate. It is possible to use a modified version of the submodule repository, but doing so requires establishing and hosting a fork to allow other developers access to these adjustments [4].

Utilization of Git submodules can help abstract away the complexities of storing source code in multiple repositories, whether related to the multirepo approach or the hybrid approach. Furthermore, even when presented with a monolithic Git repository, submodules may be used for external dependencies that are desired to be built from source alongside their dependent projects. When the top-level repository is cloned, whether for an automated build job operation or by a developer, submodules can be initialized, which ensures the working tree is up to date according to the revision identifier stored in the top-level repository. In other words, the submodule contents are established on the file system [4].

In summary, Git submodules allow for subtrees of a multirepo forest to be checked out locally and, to an extent, treated as if their contents indeed resided in a single logical repository. File system presence enables the usage of arbitrary tooling to perform bulk operations, be it a simple text replacement performed on the working tree or an invocation of an external tool that would otherwise have to be run against individually checked out copies of the respective submodule repositories. Developer tooling, such as analyzers or compilers, then need not even be made aware of the submodule or the underlying complexities of multirepos, potentially improving the workflow of working within already established multirepo clones.

Additionally, submodules lend themselves to utilizing file system based search tools available for general use cases, such as the UNIX utility `grep` or Git's built-in variant of it — `git grep` — which is able to search through the commit tree in addition to the working directory's current state [4]. Clearly, this comes at the cost of maintaining a local copy of the repositories and updating them whenever the need for searching arises.

The idea of submodules is not exclusive to Git, and other version control systems usually provide similar functionality. In the case of Subversion, a similar mechanism is offered in its *externals* facility [15].

Git-meta

Git-meta is a functional layer built on top of Git that aims to improve the Git submodules' usability when it comes to treating a multirepo subtree as a monorepo [23]. Submodules' original intention has never been to present them as similarly as possible in regards to the usual flow when manipulating a pure repository, i.e., pulling changes, committing their own and pushing them to the upstream repository. As a consequence, there are some new commands to learn or at least configuration to apply when starting to work on a Git repository that contains submodules [4].

At the cost of having to prefix commands with the `meta` keyword, such as using `git meta push origin master` instead of the built-in variant, Git-meta aims to provide enough sugar (as a parallel for, e.g., syntactic sugar) so as to erase the boundary between working subtrees that belong to the top-level repository and those that are part of a submodule. According to the extension's mission, no changes are made to the internals of the underlying repository. All implemented commands merely wrap those provided by Git in an appropriate manner, ensuring that any valid Git repository can be used with Git-meta and vice versa [4].

The extension's documentation proposes that the top-level repository be called a *meta-repo*. In contrast, all repositories included by means of submodules are called *sub-repos*,

essentially establishing their mapping for the terms *monorepo* and *multirepos* in the context of managing them from a single repository. The benefits gained in regards to user comfort are similar to those described for Git submodules in the previous section, but perhaps even more prevalent, as this extension enables their easier use and reportedly improves performance of some operations [23].

GitLab Multi-Project Pipelines

As per the feature's documentation, its purpose is to enable implementing workflows in the CI/CD pipeline in cases where these workflows including dealing with cross-project (i.e. cross-repository) dependencies [11]. Any job of the pipeline can constitute what has been described with Jenkins in section 2.5, where the success of a job can trigger the execution of another. Where GitLab only used to allow the same concept within the scope of a single pipeline, this feature allows pipelines to interact across projects/repositories [11].

The feature's intended use case mentions the deployment of applications based on the microservice architecture [11], essentially facilitating an orchestrated deployment. This basically matches the problem description of the workflow of propagating changes to downstream dependencies. Projects' pipelines could, as part of a continuous delivery stage created thereby, investigate dependency acceptance criteria (see section 2.3 — Application Dependencies). If appropriate, attempt delivering their own (just produced) version to the sources of a downstream dependency, reaching the workflow's automation.

Although only available for GitLab, the mentioned parallel of Jenkins job triggers shows the same approach could probably be taken even with other CI/CD solutions. However, even with GitLab, there is a significant drawback to the solution. The pipeline is defined statically in a file versioned alongside the project's source code [6]. Every downstream dependency wishing to accept updates automatically would have to adjust this configuration file for that purpose, arising a need for the file to be maintained and possibly causing issues at scale if the number of downstream dependencies was significant.

As long as these drawbacks are not considered too serious, however, the solution has the potential to be fully automated, remediating the workflow's identified inhibiting factors of complexity, mundanity and need to switch contexts.

GitLab Group Merge Requests / Super Merge

An issue was opened in GitLab's tracking system in 2017, calling for a feature that would establish a single source of truth when requesting delivery of changes to multiple repositories [10]. The proposal then details an issue that shares much with the workflow of cascading changes downstream, which, in a multirepo environment, requires that a separate merge request is opened in each individual repository. This makes it — at the very least — unclear where to discuss changes that span more than one repository. Noted is also the difference in the workflow regarding atomic commits that are only possible in monorepos or the need to establish an order in which the individual change requests should be merged [10].

The proposal has ultimately not been implemented. The issue was closed in 2019, and the request was split into two feature requests — the first concerns cross-project merge requests which closely resemble the original issue [12]. The second then mentions a smaller subset of features, namely the ability to link merge requests together in linear order so that they could be merged as a whole [9].

Unfortunately, the latter is still listed as under development and not yet available in any of GitLab's licensing tiers. The former and more elaborate feature call even states that the

company has assigned it a low priority due to its complexity and existence of workaround features, such as the already mentioned scripting within CI pipelines or Git submodules [12]. As such, it remains to be seen whether the features are eventually implemented and made available to the general public.

While these features would be a welcome addition that would likely positively influence the mentioned workflow, being specific to GitLab and Git repositories still leaves much to be addressed.

4.3 Problem Analysis

Following the existing solutions identified in the previous section, their scope and possibility of lessening the related workflows' inconvenience-causing factors make it easy to see there is still much left to address. According to the study detailed in the previous chapter and in the thesis's appendices, the two workflows that are most inconvenient and infringing upon developer comfort have been left unresolved. The best-identified chance at their partial resolution lies in a feature that's under development and is likely to end up as a proprietary feature of the CI/CD solution for which it is being created — GitLab.

Analysis of the ecosystem's cross-project dependencies has shown that even in its fairly early phase of development, there are many projects to build as part of the ecosystem's release process and a significant number of dependencies that require building some projects before others. The process is currently done manually. Granted no serious merge conflicts are encountered during its execution, its completion takes a dedicated software engineer's entire working day. Based on this description alone, the process could be regarded as orthogonal to continuous delivery as it can get.

The problem was initially assigned to correspond to the entire scope of the ecosystem's delivery, which currently spans about 200 source code repositories. Analysis of the entire process yielded the graph pictured in the figure 4.1, which displays the high-level steps necessary to deliver the ecosystem. The entire process can be split into several parts:

1. **Build Environment Preparation** — this encompasses nodes 1–6. The CI/CD pipelines of projects throughout the ecosystem use a set of Docker images as an environment in which the commit stage and some of the user acceptance test stages run. To ensure that these environments are not stale, a full release process starts by rebuilding them.

   The first node produces a base environment with only the essentials, such as Git tooling or certificates of internal CA authorities. The resulting environment then serves as a base environment for the programming language used for scripting within the pipeline, produced by node 2. Consequently, these scripts can be rebuilt (3) and the relevant configuration source updated with a new revision that enables their use from within CI jobs (4).

   Once an updated version of the CI scripts is available, the remaining build environments specific to every supported programming language can be generated as well (5). Once all environments have been generated, a different configuration source is updated, this time to enable running CI jobs on the created set of Docker images (6).

2. **Core Platform Rebuild** — this encompasses nodes 8a,b–10. These nodes abstract away the complexity of traversing a full dependency graph that involves several groups

of projects of diverse technology stacks. These projects need to be rebuilt in a non-arbitrary order to incorporate the changes in updated CI scripts and the underlying build environments produced during the previous part of the release process.

This node encompasses most of the mentioned 200 source code repositories. Detailing them in the graph would introduce unnecessary and irrelevant clutter. The projects' nature is not significantly relevant to the release process itself, save for their mentioned requirement of a specific order of execution as dictated by their interdependencies.

3. **Acceptance Testing** — the nodes 11a, 11b, and 11c represent three of the many projects that rely on the platform internally. While some are dedicated testing suites, the platform tooling is released with a set of usage-demonstrating examples that are updated as the platform evolves. These samples come with their own test suite and provide an additional check that the platform is able to fulfill its purpose.

4. **Merging** — the nodes numbered 12, 12a and 12b represent merging. The ecosystem team enforces a policy of maintaining the mainline's passing build status, enforced by a derivative of the feature branching strategy mentioned in section 3.2. All changes must first be delivered to a feature branch and trigger a successful CI pipeline before these changes can be merged into the mainline.

   For this purpose, a clean variant of the release process would perform all steps that precede the 12th in these feature branches. Only after it is established that the entire stack can be built and passes all implemented checks is it reasonable to merge all these feature branches. Moreover, it is worth mentioning that while an automated tool has no inherent problems in opening 200 branches and subsequently merging them, a human operator is likely to take shortcuts and merge immediately, potentially polluting the mainlines with defective changes. In addition, while this policy ensures that the mainlines maintain sources of working artifacts, doubling the number of CI pipelines (one before merging and one after) also, on average, doubles their duration.

5. **Configuration Release** — finally, once all environments and tooling passed all checks and have been merged into the respective mainlines, it becomes possible to create a final set of configurations. They are not semantically different from those that have undergone aforementioned testing procedures. However, the ecosystem mandates that every run of the pipeline updates the version number. This must therefore be reflected in all configuration packages created until this point. The nodes responsible for these operations are numbered 13 through 16.

6. **Downstream Deployment** — in order to gain external feedback on the just-released version of the ecosystem, it is desirable to ensure it is deployed to all interested users, potentially setting off additional release plans.

   There are two primary configuration packages that describe the public portion of the ecosystem that comprise candidates for downstream release. The configurations package from node 17 encapsulates the versions of internal libraries, which become true dependencies in downstream projects, in the sense that they are used as application dependencies. The CI templates package from node 18 then concerns the development side of downstream projects, potentially bringing support for additional programming languages or improving that of existing.

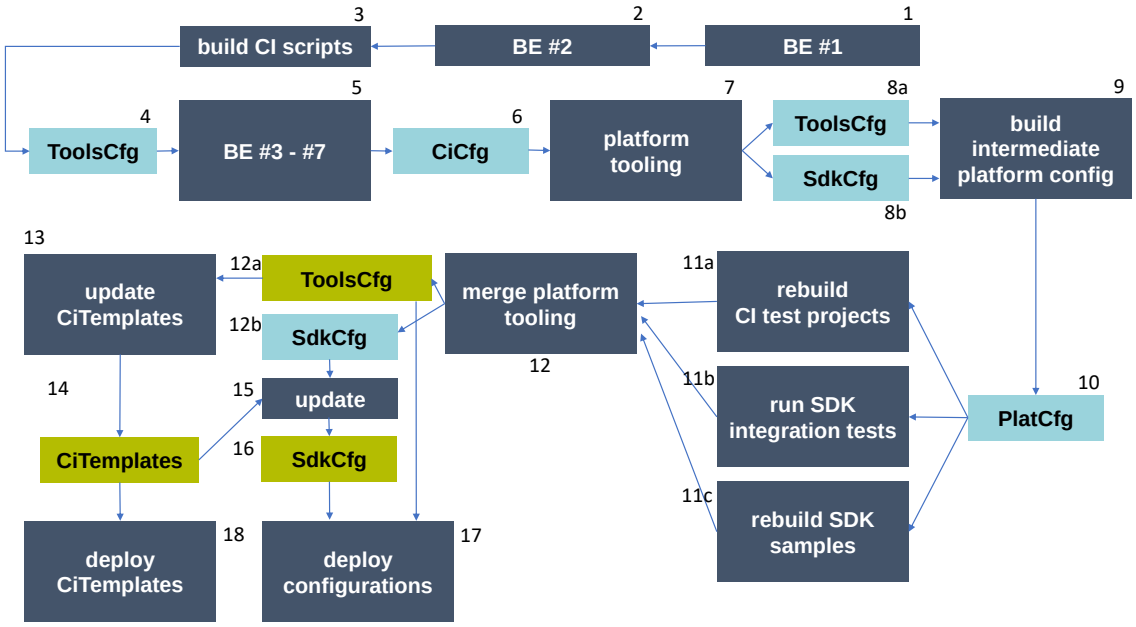

Figure 4.1: High level graph of steps that need to be performed in order to consider the ecosystem released, leaving no doubt that any and all internal changes have been propagated, even to most distant projects of the full dependency graph. Dark blue nodes of the graph represent rebuilds of particular projects or execution of steps described in their label. Light blue nodes represent intermediate artifacts. Lime nodes represent configuration sets that describe the released ecosystem. Individual nodes are numbered by the intended execution order.

It has been agreed upon with the development team to design a solution that will make it possible to automate the first — build environment preparation — stage of the release process, that will serve as a proof of concept which will then be expanded to support the remainder of the entire release orchestration.

4.4 Requirements Specification

Accounting for the properties of the release process described in the previous section, the recommendations arising from the CI/CD process and results of consultation with the ecosystem team, a set of requirements has been established for the proposed solution to follow.

- **Resumability** — given the sheer number of repositories involved in the release process, the probability of a merge conflict occurring in its duration is very high. When a human operator orchestrates the release, they are able to pause at any point of the process for any reason, e.g. when a possible defect is detected or when one of the hundreds involved CI/CD pipelines fails.

  Once the issue is resolved, a manually orchestrated process is free to continue. It is imperative that the proposed solution honors this possibility in regards to pausing, resuming and retrying the individual steps that represent any executed workflow.

- **Security** — it is desirable for the proposed solution to honor the access rights of the user who initiates a defined workflow. Described in context of the ecosystem, not all employees have write access to the core repositories at their disposal. Allowing them to initiate a process that uses a privileged service account would allow them to circumvent the repository access lists, presenting a potential security vulnerability.

- **Hosted Service** — even if accounting only for the CI/CD pipelines involved in pushing and consequently merging changes to the mentioned 200 repositories and assuming that every constitutive pipeline ran for only 5 minutes, the time duration estimate would amount to $200 \times 5 \times 2 = 2000\text{min} \equiv 34\text{hrs}$.

  The solution should therefore be made available as a hosted service accessible from the company's internal network. While this does not forbid the possibility of being deployed locally on a machine of a software engineer, it is necessary to allow for the implemented orchestration to continue during off-work hours when such a device is typically offline.

- **Configurability** — the proof of concept solution should only cover rebuilds of build environments, as previously mentioned. However, the solution should be designed in accordance to the future plans that span the entire release process. Furthermore, the solution should avoid vendor lock-in towards Git and GitLab and allow for future expansions should any of the two mentioned technologies be replaced with another.

- **Extensibility** — the solution's design should be open for extensions to accommodate new technologies as they appear. A solid foundation should be provided as staging grounds for these extensions, and as few restrictions as possible should be imposed in regards to their operations.

Chapter 5

Pipelining of CI/CD Pipelines

Considering the system's designed purpose of orchestrating varying source code versioning systems and CI services, such an ideal system would provide foundations that would, to the greatest extent possible, ease the task of developing and maintaining extensions responsible for coordinating these systems and services. In addition, end-users who only use these extensions to automate workflows involving repositories that they maintain should not have to become knowledgeable about the system's internals. On the contrary, a brief tutorial and perhaps a few starter job templates should comprise a sufficient introduction.

The following sections describe the created system's components, the direction and justification of undertaken design steps, and the features that they offer for other components to utilize.

Entities & Terminology

The concepts and justifications that follow later on in this chapter appeal to various terms that refer to entities present in the system. Their brief explanations follow below for ease of understanding. A more detailed description typically follows in the section or sections of involved subsystems.

A **variable** is a concept owned by the system's data model. It refers to a named object with a designated data type that can be used to store a value. It implements a mechanism of change notification and has a parent variable scope.

Variable scopes are essentially named collections of variables of varying data types grouped together by intent or ownership. In addition to variables, a variable scope can recursively contain other scopes, forming a tree structure with a root variable scope, where edges signify a parent-child relationship.

A **variable path** refers to a variable's location in the memory model's tree. It is composed of an ordered collection of names that always has at least two items. The last item corresponds to the variable's name, and the penultimate name identifies the parent variable scope. If applicable, the remaining sequence of names corresponds to the traversal through the tree of variable scopes, starting from the root scope, that leads to the variable's direct parent.

Binding is a mechanism that subscribes to changes in one or more source variables. The source variables' values are obtained and used to calculate and assign a new destination value to one or more destination variables whenever that happens.

The orchestration of operations works in a way comparable to computational graphs. However, instead of purely arithmetical operations, the nodes found in the graph are referred

to as **tasks** and provide stateful execution of arbitrary programs with an unconstrained number of input output variables. The previously mentioned concept of bindings could then be regarded as the interpretation of edges in such a computation graph, carrying out data transfer between individual tasks. As for the tasks themselves, their contract is not too complicated:

- Each task must have a unique identifier.
- Each task must have a variable scope for input variables and another for output variables.
- Each task must have an output variable that denotes whether it had already finished executing, and if so, then whether successfully or not.
- Each task must have a collection of predecessor tasks. Only after these tasks successfully finish their execution may the task commence its own attempt. The collection may be empty.

A **job** is a top-level aggregate that encompasses all run time resources required for the realization of an orchestration operation. The more apparent components include the memory model (the root variable scope and its children), the set of created bindings and task executors, or the event bus. However, individual subsystems and plugins may bring in their own dependencies necessary for their internal function.

An **input document** is a text file that fully describes a job. Once provided by a user, the system is capable of parsing such a file and creating a job instance that corresponds to the contained description.

5.1 Plugin System

To better facilitate extensions and modifications of the system's behavior, the implementation shall ensure loose coupling of components and make use of dependency injection, a set of techniques related to loose coupling that help utilize the consequential benefits, such as configurable composition, life cycle management, and interception of objects [27]. Additionally, implemented classes shall only realize their dependencies by means of standalone interfaces instead of depending on a particular instance that implements it. The combination of these traits should enable extensions to alter the original behavior to a great extent.

In order to create an instance of a service provider — an object implementing the centralized life cycle management functionality, responsible for the creation and disposal of objects — a collection of service descriptors must typically be established, where each descriptor instructs the provider of the service type, its implementation, and the latter's life cycle [27]. The registration of services implemented by the memory model, event system, input document schema management, and the basic constructs of input document loading and expansion will be handled by the seam component which will implement an interface for interaction with the system.

The remaining components are considered non-essential and are instead loaded as plugins. The system provides a simple plugin entry point interface. Individual plugins are expected to take the form of .NET assemblies, in which exactly one class shall implement that interface. The activating component (currently the API server) is then responsible for

locating and loading the necessary assemblies, instantiating their entry point classes, and incorporating them in the service registration process. While the plugin is free to register its internal services, it primarily registers its implementations of interfaces provided by other present system components and thereby extending or adjusting their behavior.

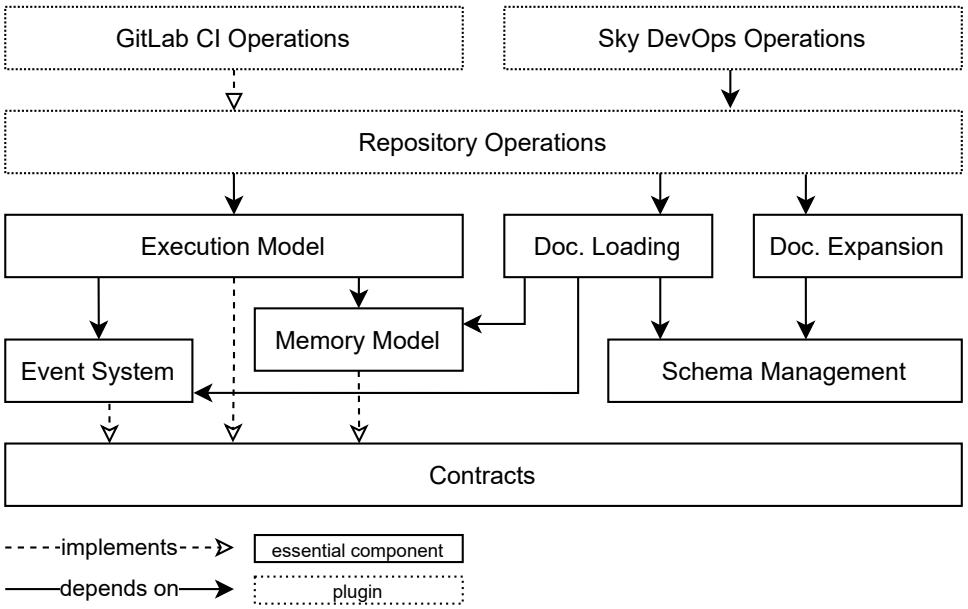

Figure 5.1: Overview of the system components and their relationships.

5.2 Document Processing

Two sets of requirements affected the design and implementation of input documents. The first set comes from the system internally on behalf of the subsystem responsible for processing them into materialized job instances prepared for execution. Besides containing all mandatory information, the job description should be conveyed in a level of detail that closely matches their actual representation. This would take a burden off the loading subsystem by keeping the complexity of its behavior low. That behavior would mostly consist of creating an in-memory copy of the text-described schema by creating appropriate class instances and the assignment of respective member fields. The second requirement set then arises from the needs of users, who, as previously mentioned, need not care for the system's internals and for the desire to keep their implementation simple and succinct. As it happens, these characteristics actually match the users' own expectation for the input document: an uncomplicated description of what needs to be done and where it needs to be done, largely ignorant of the complexities involved in putting them in effect.

The conflict is apparent but not unresolvable. After all, a parallel could be drawn to high– and low-level programming languages, some of whose distinctions are very similar to the problem at hand. By introducing an intermediary that would expand a high-level description, short and concise enough to be readable and writable by human users, into a low-level one that is, on the contrary, verbose enough for the loading subsystem to handle, all of them expressed demands are satisfied.

From the variety of markup languages currently available for the denotation of configuration, XML was chosen as the technology to power both the low-level and the high-level

schema. Numerous factors contributed to this decision, ranging from objective qualities to subjectively perceived fitness for the assumed role:

- **Adoption** — Due to XML's popularity, it could be speculated that almost every active programming platform likely supports dealing with XML documents in some way or another. In the case of the C# language, the .NET framework's standard library provides comprehensive first-class support, removing the need to rely on a third-party library [1].
- **Schema Definition & Validation** — Although published separately to the XML standard and only in the form of a recommendation, XML schema documents (XSD) were established as a mechanism to describe and constrain the types of XML elements, their attributes, and contents found in described documents. Data documents then elect to reference these schemata, enabling automated tools to determine whether their contents comply with the definitions or referenced schemata [8].
- **Namespaces** — The primary intent of XML namespaces is to allow equally named elements from multiple dialects or schemata to coexist in a single document, as every element can be adorned with an identification of its origin namespace [29].
  The following arguments support the usage of this feature in regards to the requirement of extensibility:
  - When developing extensions to the system, developers need not concern themselves with whether a particular name of their choice is already in use. Likewise, implementation of a name conflict avoidance solution, like prefixing, would be made redundant.
  - The loading subsystem is able to make use of the identification of origin to determine which component is responsible for processing a particular part of the input. The same can also be used to gracefully fail the loading process if the provided identification is not known instead of producing a job without some parts that could not be loaded.

Schema Management Subsystem

Strictly speaking, the expansion of high-level documents into their low-level counterparts is a process distinct from the low-level documents' loading. The fact that they use a common data serialization language is seen as a coincidence. Honoring that, each of the two processes resides in a separate component. Nevertheless, this coincidence is an implementation detail that requires both the loading and the expansion component to access the XSD document. Duplicating it across multiple .NET assemblies was deemed undesirable, prompting for the creation of another subsystem that would act as a registry of input document schemata.

The subsystem publishes an interface for schema registrants, i.e., plugins whose functionality requires extending the already present schema of input documents. It registers the core schema through the same mechanism and provides a service that aggregates these registrants so that the loading subsystem does not need to aggregate the registrations on its own.

Document Loading Subsystem

It is hard to picture a plugin extension to the system that would expand on its behavior with no configuration whatsoever. As an absolute majority of configuration parameters found in the implemented solution are specific to individual jobs, the corresponding input documents become the most appropriate location for that configuration to be stored. Thus, it becomes apparent that a mechanism is required to manage the delegation of input documents' object model trees across the component boundary.

The loading subsystem publishes an interface for other components to implement if they wish to participate in the loading process. The intent is for every recognized element type to have a separate loader class implementation. The contract prescribes that the following capabilities are implemented:

- The loader implementation must report on the object type that the loader class is designed to instantiate and configure.
- The loader implementation must be able to inspect an XML element and report whether it is able to parse it into the reported resultant object type.
- Finally, when given an element deemed compatible in accordance with the previous item, the loader should create a new instance of the reported resultant object type, configure it appropriately and ultimately return that instance.

In addition to this interface, the loading subsystem also publishes a service that aggregates all registered loader implementations and initiates the delegation process upon the first level of child elements: the global scope of variables and the collection of tasks. Thereby, invoked loaders return a result immediately or recursively call other loaders in order to process nested content. If the loader encounters an unrecoverable error, it reports that it could not process its assigned element, and other matching loaders (if any) are allowed to make their attempt. Elements left unprocessed processed by all registered loaders interrupt the loading process with an error.

Not only to serve as an example of how to implement the above functionality but also to ensure that essential function is always present, the subsystem implements and on its own registers loaders of the memory model, namely of variable scopes, integral variables, string variables, conditional string variables, and bindings. The memory model is loaded in three stages: the corresponding instances are first created, then interconnected via described bindings, and only then are the variables assigned the requested values where applicable. Bindings subscribe to changes in their source variables and generally ignore any value that the variable already contains. Assigning values (and therefore generating these change notifications) after all bindings are created and activated guarantees that the memory model is initialized into a consistent state.

Document Expansion Subsystem

As previously explained in-depth in the section 5.2, the expansion subsystem exists to improve the experience of users interacting with the system and to abstract away some of its internals. Its purpose is to accept a high-level document specification of the job to perform. These high-level specifications give off a more declarative appearance than the low-level counterpart. They involve two areas of focus: the first details the changes that

need to be done, while the second describes the repositories and thereby the environment in which the change execution shall take place.

Multirepos can often be classified into clusters by technology or by the kind of artifact that their CI pipeline produces, e.g., a Docker image or a Python package. At least within a single organization, repositories found in these clusters are then likely to use a very similar directory structure, pipeline script, and settings. These similarities will then, in turn, affect the shape of operations orchestrated on them, causing them to differ very little or not at all. For example, in an orchestrated job to update dependency versions across a set of repositories that all use a single package management system, the most likely difference would be the location of the file used to store dependency versions. For this particular example, the input document could be reduced to an ordered list of repository paths.

As was the case with the document loading process, the expansion process is implemented to work on a DOM representation of the input document. Various components of the system are able to register their expansion providers, who, upon invocation, search the tree for instances of their replacement subjects and mutate the document as necessary by changing, adding, or removing nodes from its tree. This repeats for as long as any of the providers has changes to perform: the document's expansion is considered complete once there is nothing left to expand.

The most prominent advantages of the high-level document concepts make themselves apparent with more complicated expansions described in the section 5.3. Even so, the core component that is the expansion system's cornerstone already registers some convenience-related expansion providers.

As it stands, the expansion is currently implemented as a set of C# classes that operate upon the XML DOM tree. When designing the expansion process, a standardized means of these operations — XSL transformations — was considered but ultimately rejected in fears that their computation power would ultimately be found insufficient to fulfill the requirements of expansions or that they would be too difficult to express by potential adopters. It is now apparent that at least the former of these fears was unfounded. As for the latter, it could be argued that the current C# implementation reduces the number of technologies that potential contributors have to be knowledgeable in.

It is worth mentioning that users naturally cannot be prevented from implementing their own extra intermediary that would translate a custom high-level schema, realized in an arbitrary notation, to either one understood by the system. Conversely, an effort to improve user comfort in this manner would be a mere expansion on the path pursued by the author in this aspect.

5.3 Execution Engine

The subsystems described up until this point guarantee that an input document that is valid can be parsed and transformed into a job instance. The exact process of determining validity would be composed of many steps. The document must not reference any XSD schema documents not loaded into the system, as that would signify either a version mismatch or an absence of a required plugin. The remaining schemata are used to validate that the document is in a proper format. Finally, there is space for semantic errors to occur when the document is being parsed, such as a binding referencing a variable at an unknown path.

Nevertheless, a valid document would execute along the happy path and produce a properly initialized job instance. As previously mentioned at the start of the chapter, the job instance aggregates all components required to handle the execution of tasks. These com-

ponents that ultimately comprise the system's execution model are numerous and explained below.

Event System

The C# language already includes a robust event system that revolves around the concept of event arguments, a strongly typed object that carries state information describing an event's occurrence. The language's *event* keyword provides an effortless setup of the observer design pattern, allowing observers to both subscribe and unsubscribe from a particular event. Both operations operate upon a delegate: *a method reference with a specified type signature* [1].

However, one of the design decisions made for the execution engine was to keep it single-threaded. With respect to the system's characteristic of being an orchestration tool, it is intended to delegate all the intensive work to other systems. The remainder is not believed to warrant concurrent execution. That being said, this design aspect does not ban any component from performing some of its workloads in parallel. It merely means that it has to take into account that synchronization might be required when interacting with the rest of the system.

A second design decision was for the event system to be asynchronous. From a certain viewpoint, this goes hand in hand with the desire to ensure that only one thread is dispatching events at any given time. The built-in event system operates synchronously and invokes all subscribed event handlers immediately, as part of the operation responsible for notifying observes about an event's occurrence. Additionally, spare for pessimistic concurrency mechanisms, a synchronous event system could have difficulties providing any sort of consistency guarantees. If an external system published an event, a synchronous system could notify its observers while another event is already being handled. Should the external event's handler, for example, decide to react to a variable's value, an already executing handler could rely on an assertion that it remains unchanged, possibly causing a data conflict.

At the core of the implemented event system lies an event bus: an object tasked with operating an event system compatible with the two mentioned design aspects. Its responsibilities are to facilitate the recording of events as they happen and to appropriately notify subscribers of these event occurrences. To carry out the first of these duties, the event bus maintains a set of queues, one for each event priority level. The system currently supports two of these priority levels, low and high, and their importance will be cleared up in the following section regarding the memory model.

A dedicated thread then handles the other duty of notifying known subscribers about these event records in the correct order. This thread starts and stops alongside the job itself. While it is running, it periodically checks whether an event is present at the head of the queues in the order of their priority — the high priority queue first, the low priority queue afterward. If an event is available, it is removed from its parent queue, and relevant subscribers are synchronously, one by one, notified and allowed to execute to handle the event's occurrence.

In conclusion, the implemented event system honors the following guarantees:

1. Published events are processed asynchronously; the operation of publishing of an event does not block.
2. High priority events are always handled before low priority events.
3. Events of the same priority are handled in the other in which they were published.

4. No more than one event handler is executing at any given time.

Memory Model

The memory model — comprising of variable scopes, variables, and bindings, as already explained at the beginning of this chapter — could be visualized as a directed graph. Its nodes would represent individual variables. These nodes would be interconnected by edges representing the individual data bindings, whose role is to propagate these values through the graph of variables as required by the upper-level components. Variable scopes would not have a visible role, as they primarily group together variables of interest to make them easier to locate.

The C# language recognizes two categories of data types: reference types and value types. The former support assignment of the special `NULL` value, signifying that the value is actually invalid or missing. Value types, on the other hand, do not have a value with similar semantics. The closest match would be the default value [1], However, using it in the same manner would be inappropriate. For example, the default value of the most prominent integral data type (*System.Int32*) is that of 0, and assigning this value a special meaning would impose severe restrictions on all users of the memory model.

To discern whether a value was assigned to a variable, regardless of its underlying data type, the memory model takes inspiration from functional programming and its *Maybe* data type, implemented by the C# library *Optional*. With its help, each variable either has an assigned value of the intended type or is assigned an instance of an error type that explains the lack of value. The wrapper itself is a C# structure, which means it is treated as a value type and is immutable [20].

Another introduced concept is that of variable readiness. It is somewhat related to the same issue of whether the variable has been assigned a value. However, as opposed to querying the *Optional* wrapper for whether the value is present, variable readiness interprets some error states as transient and non-fatal. Additionally, the assigned value may implement an interface that lets it adjust the returned readiness state based on its own logic. Variables report their readiness to be one of the three following states:

1. **Ready** — a valid value is assigned
2. **Pending** — no value is assigned; however, the embedded error description reveals that no value has simply been assigned yet. As such, the variable is pending an assignment.
3. **Failed** — no value is assigned, and the embedded error description indicates that the task or binding responsible for assigning the variable encountered an error.

Memory Model Consistency

In order to fulfill its role as one of the cornerstones of the execution engine, the memory model expands on and profits from the consistency guarantees of the event system. By utilizing the event system for the assignment of values, the memory model guarantees that variables never change values while another event handler is executing. Consequently, executing event handlers will always observe the same value of variables at all points of their execution.

In order to deliver on this guarantee, the event bus is used to actuate all state-mutating operations involving the variables. In order to assign a value to a particular variable, users

publish a request event whose state includes the variable's path and the target value. The memory model handles the event and ultimately publishes a change notification event that specifies the path of the changed variable and both its old and new values.

Binding System Consistency

The memory model comes with one more guarantee. When a variable changes and that variable is one of the source variables of an existing binding, the system ensures that by the time the next low priority event is handled, the destination variable is updated with the binding's logic or its readiness state is *pending*.

This has significant implications on the task execution model. As long as its behavior uses a low-level event as a starting trigger, it is able to make safe assumptions about its inputs, especially if these inputs are data-bound to another task's outputs. In this way, a changed output is guaranteed to either contain a consistent value or to be invalidated, at which point the task will wait for a value to be eventually assigned.

If one were to observe the events processed by the event bus, it would be revealed that the memory model actually generates many more events than just the last one that informs of the value assignment process's completion. This internal mechanism is partially responsible for upholding the second guarantee of consistency and consists of the following steps:

1. The memory model receives a low-priority variable value change request event $E_{Assign}(P, V)$. Let's assume that the value V is defined, i.e., is not erroneous. The event data include the variable path P and the target value V. The corresponding event handler does the following:
   (a) A random transaction identifier I_{TX} is generated.
   (b) An internal value-setting event A with high priority is published via the event bus.
   $$A = E_{Set}(P, I_{TX}, \text{Unconfirmed}(I_{TX}))$$
   (c) An internal value-setting event B with low priority is published via the event bus.
   $$B = E_{Set}(P, I_{TX}, V)$$
2. The memory model is asked to handle the event A, causing a special error state of $\text{Unconfirmed}(\text{I}_{\text{TX}})$ to be assigned to the variable. This is accompanied by a change notification event A', with high priority, being published to inform all interested subscribers that the variable at path P had changed its value to this erroneous state, which holds onto the transaction identifier I_{TX}.
   $$A' = E_{Changed}(P, \text{Unconfirmed}(\text{I}_{\text{TX}}))$$
3. The memory model is asked to handle the event B. The value must at this point contain the special error state $\text{Unconfirmed}(\text{J}_{\text{TX}})$.
   The event can be handled in one of two ways:
   (a) If $I_{TX} = J_{TX}$, then the internal transaction completes by changing the variable value to V, again accompanied by a corresponding change notification event B', this time with low priority.
   $$B' = E_{Changed}(P, V)$$

(b) If $I_{TX} \neq J_{TX}$, then another internal transaction has been started (and its own high priority event A duly processed before this process's low priority event B) and as the target value is no longer consistent, and no action is taken.

The above algorithm hides an important detail, thanks to which it actually fulfills the second guarantee of consistency, one that is better visible in the diagram 5.2. The involved events do not have a predetermined priority. Instead, the priority is derived from whether the target value (in case of E_{Set}, E_{Assign}) or new value ($E_{Changed}$) is erroneous due to being unconfirmed — high if it is, low if it is not.

This way, had an intermediary variable existed in between the source and the destination variables of a binding, and the binding actually were two so as to maintain the original data connection, the algorithm would still hold the second guarantee. The high priority events would traverse through an arbitrarily long chain of bindings before any of the queued low priority events could be processed.

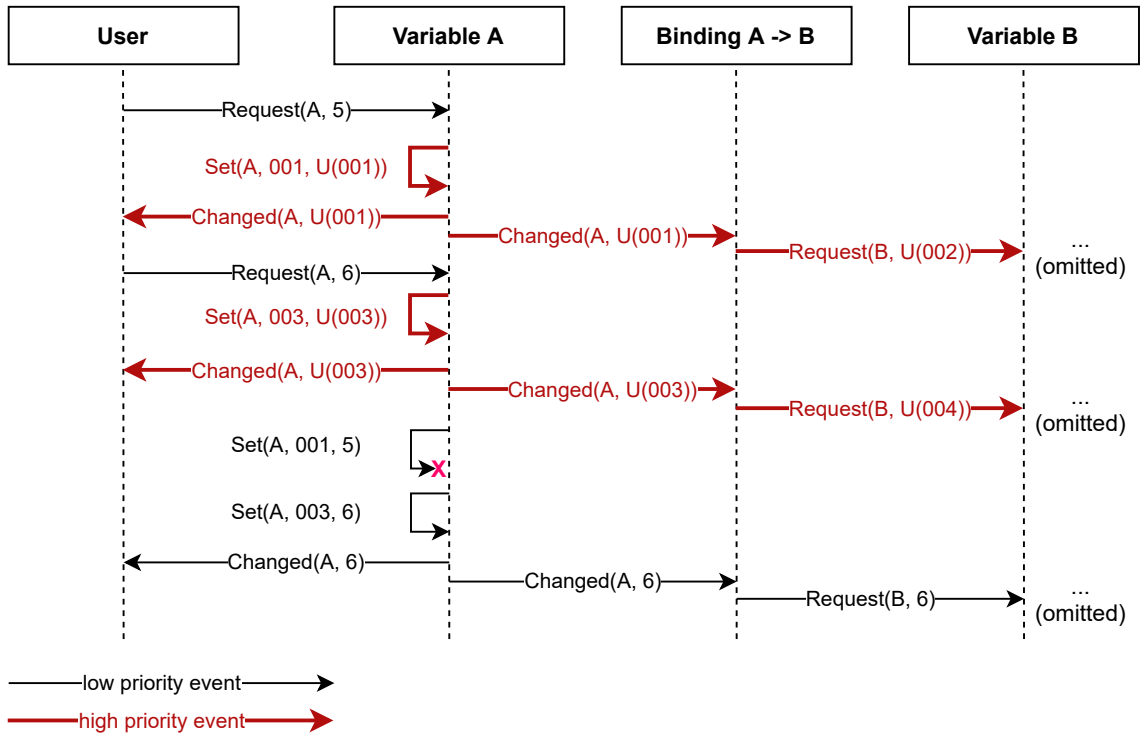

Figure 5.2: A sequential diagram, showing how the system handles assigning the value of 5 to variable A when that process is interleaved by another assignment request. A binding that mirrors the value of variable A to B while upholding the consistency guarantees.

Execution Model

The last component of the execution engine is the execution model itself. The event system and the memory model, on top of which the execution model is built, come with several consistency guarantees that allow the execution engine to focus on the execution of the expected operations — exactly as intended.

Considering the implemented event system's asynchronous nature and the need to separate the task state from the behavior of its executor, it only makes sense to consider finite

state machines as a possible execution model. The events that their subsystem publishes from time to time map well to the concept FSM's transitions; in both cases, the expected outcome is for some behavior to happen, such as inducing a transition to another state or firing off a secondary event or events. This comparison, of course, assumes an adjusted formal definition of FSM's so that the input alphabet consists of event types instead of symbols.

Thankfully, several C# class libraries are available that provide a framework for the setup and usage of event-driven state machines. Among them is *Automatonymous*, which beneficially adheres to the expected pattern/approach of having a singleton instance for the machine's behavior and operating upon a separate instance used to hold state [22].

In order to gain access to the benefits of the consistency guarantees, it is necessary to synchronize the state machines' transitions with a low-priority event so that all high priority events have already been handled and the consistency of variable values ensured. To achieve this, a base state machine has been prepared that custom task executors merely extend. Instead of transitioning directly to the target state, the base state machine transitions to an internal '*pending*' state, publishes a low-priority event, and only executes the originally requested once that event is handled by the event bus.

Furthermore, the states and their transition behavior inherited from the base state machine implement the boilerplate contractual obligations, whose fulfillment is required from all executors. The base class additionally provides shortcuts for the setup of some commonly needed behavior, such as:

- waiting for predecessor tasks to be specified and until they finish executing
- subscribing to changes in the task inputs' variable scope
- starting a long-running external operation and requesting that the machine be notified when that operation completes,
- ensuring that the scope of input variables temporarily stops (and queues for later processing) accepting changes, so that the input values keep their values for as long as the operation that uses them runs

Contractual Obligations

As much of a great fit may the state machines appear, special care was taken when designing the interfaces that make up the task execution contract so that experimenting or replacing them with a different construct does not warrant rewriting a majority of the system. In summary, the object responsible for executing a task must be able to:

1. report about the task's life cycle, as in, whether a task's execution is stopped, starting, running, or stopping
2. start a stopped task
3. stop a running task
4. report about the task's current progress/operation
5. report a list of commands that the task is capable of accepting in its current state
6. accept a command that the current state is capable of accepting

The FSM implementation of task executors fulfills these obligations by, respectively:

1. determining whether the task's state machine is in the special states that correspond to them being started or stopped, or whether any of these operations have been requested but not yet fulfilled
2. raising an event that the machine reacts to by transitioning from the start state
3. cancelling any pending asynchronous operation and transitioning to the stop state
4. returning the name of the current state
5. determining which events is the current state capable of handling
6. raising such an event

Repository Operations Subsystem

Based on the previous sections, it could almost seem like just a coincidence that the created solution's destiny is to orchestrate operations across a set of multirepos. However, the openness may become useful with the invention of a new, revolutionary approach to source code versioning or build systems that do not play well with current state-of-the-art principles, which are reflected in the following design.

The repository operations subsystem is the first non-essential component of the system and also the first that the system loads in the form of a plugin. It establishes a basic set of tasks, portable across source code versioning and CI systems, whose purpose is to deliver changes to a source code repository and query the connected build system for the result of the corresponding build/integration pipeline.

Alongside the tasks' definition, this plugin also provides the corresponding XML schema for processing input documents, the tasks' state classes, and their executors. However, each of these tasks contains an input variable for a *repository operations provider*, which is used in accordance with the strategy pattern to abstract away the differences between integrated systems. The tasks' logic is assumed to be portable enough for the future integration of additional systems to require drastic changes to how their executors operate. This subsystem publishes only such a provider's contract and lets another plugin implement it.

What the subsystem does implement in addition to publishing the relevant contract, though, is the concept of *submittable changes*: an operation performed on the source code repository. The set of submittable changes is the cornerstone input variable that dictates what changes to submit to the manipulated repository. As is the case with tasks and tasks executors, submittable changes only represent the declaration of what has to be done, and the rest is left for the change executor. To give an example, the basic supported change types include manipulation with a YAML document or overwriting contents of a file.

The repository operations subsystem provides several task types for other components to use as building blocks. Their use, however, is not mandatory; these other components may as well use their own constructs if that is deemed beneficial. The provided tasks types and their purpose is as follows:

- **OpenChangesTask** — used to establish a feature branch or a similar concept supported by the specific code versioning system so that a destination exists towards which changes can be submitted.

- **SubmitChangesTask** — manipulates the previously established state of the repository by executing all of the configured changes and submitting them to the destination. Following a successful submission of changes, waits for the triggered CI pipeline to complete and asserts that it completed successfully.
- **MergeChangesTask** — merges the green-lit feature branch into the configured main branch and waits for the triggered CI pipeline to complete successfully.
- **VerifyArtifactsTask** — contacts an artifact storage system (which is also abstracted into a different strategy provider) to verify that an artifact with a specific identifier indeed exists.

  Build systems often involve a number of standalone components that communicate asynchronously. Following a successful build job from a CI pipeline that, e.g., uploads a Docker image, a non-deterministic delay may occur until that image becomes available for download.

GitLab Operations Subsystem

While GitLab *started out as a tool only for source code management*, it evolved into a powerful tool capable of helping *from managing an initial idea to building and testing source code, all the way from development to production* [6]. In version 12, its features range from the hosting of Git repositories to issue tracking, a built-in CI/build system, container registry and more [6]. In addition, these features are optional and can be turned off in favor of a custom standalone service that only integrates with the features that have been left enabled [6].

The solution's pilot environment makes use of GitLab's repository hosting and CI system features. Among other implications, this enables a somewhat simplified approach to authentication, which is typically required to submit changes to source code repositories. Unless configured otherwise, which is not the case for the pilot environment, GitLab supports Git's HTTP transport method [4] via the HTTPS protocol [6]. Users then authenticate themselves via the HTTP protocol's native features, using their username and an access token (or less preferably, their password) [6]. As for the rest of GitLab's API, such as that pertaining to the CI system, it is also served over HTTPS and uses the same means of authentication. This is convenient as when configuring the subsystem, only a single set of credentials is required.

Summarily, the subsystem implements the *repository operations strategy* published by the similarly named component. Additionally, it extends the document loading process to expose this implementation for jobs to use by registering a schema provider and an appropriate loader. As a convenience feature, an expansion provider is included as well, which can be seen at the end of the sample document C.1 and applies the strategy to all tasks present in the job, instead of requiring that to be done manually. The list of implemented operations follows below with a brief description.

- **Branch creation** — uses the GitLab's HTTP API to create a Git branch with the provided name or a random name if none was supplied to the strategy instance.
- **Change submission** — establishes a temporary local clone of the repository via a wrapper for the `libgit2` library[1] that allows using it from inside the .NET ecosystem.

[1] https://github.com/libgit2/libgit2sharp

Change executors are invoked over the local file system, and a commit is created and pushed to the remote Git repository.

- **Branch merging** — uses GitLab's HTTP API to create a merge request and waits until it is in a state that allows merging. Depending on the setup of individual repositories, this presents various options: immediately, once the CI pipeline passes, or after another custom rule permits it. Still using the API, the merge request is then approved and closed programmatically.
- **Pipeline ID lookup** — takes as an input the commit ID that was expected to trigger a CI pipeline, then queries the API until a corresponding CI pipeline is found. This generally takes a few retries as GitLab is a distributed system with a background job scheduler [6] that can be busy or overloaded.
- **Pipeline status lookup** — uses GitLab's HTTP API to query for the previously looked up pipeline's status and reduces it to whether it is still in progress, completed successfully, or failed.

Sky DevOps Subsystem

Keeping in mind that the entire solution was designed primarily for the pilot environment and to fulfill its requirements, it does not come as too big of a surprise that the final subsystem only adds expansion functionality. All of the necessary building blocks and functionality are already present. All that is left to solve is to adjust the solution so that it better fits the environment where it is deployed and its users.

One of the aspects that could measure that fitness would be the purpose of introducing a document expansion system in the first place: how long and complex, and thereby infringing on user experience, will an average document be? In this regard, the only constraint to a developer's imagination is that the surrounding components expect a properly formatted and schemata-referencing XML document. Where this subsystem suffices with what configuration it is given in the input document, nothing prevents a more ambitious candidate, perhaps one from the near future, to obtain additional metadata from elsewhere, such as dynamically generated or elsewhere stored dependency trees from a dedicated network service, simplifying the document even further.

As was foreshadowed in the section 5.2, multirepos found in the pilot environment present a finite amount of flavors, which, in addition to the type of project whose sources are stored in that repository, typically determine these source's directory structure, locations of important files, and kinds and identifier formats of artifacts that these multirepos' CI pipelines yield.

Named after the team that has become the solution's first adopter, the Sky DevOps subsystem exploits this categorization of multirepos for the team's benefit. The policies and rules applicable in the target environment are easy to implement as part of the expansion process, empowering developers to focus on and specify only what is essential: the changes they need to make and where.

A real sample of a high-level input document is available in the appendix C. It shows that the higher-level document can be dauntingly verbose, even for a job comprising of a fairly simple set of operations. While the changes that the referenced document requests — for a value to be changed in a YAML document file — appear too minor to have a noticeable impact, it would not really be that much of an overstatement that this resolves about a

third of the release process originally analyzed in section 4.3. GitLab's CI system uses YAML files checked into repositories for the setup and configuration of their CI pipelines [6]. As the pipeline definition scripts are considered to be a part of the release, ensuring their propagation to all projects is an important step of that process.

Furthermore, it makes even more strengths of the expansion subsystem apparent. This component transparently implements several custom policies and conventions at no or even net negative cost to the user and without any modifications to the system's core components:

- **Versioning policy** — the Docker image tags hold a semantic version number, which is automatically incremented with every change and which includes a pre-release tag composed of the branch name if the artifact is built from a feature branch (i.e. was not yet merged). The expansion process automatically adds this to the list of requested changes.
- **Naming convention** — the version number is assumed to exist in a conventionally named file, as the document did not manually override it.

  Moreover, while the *VerifyArtifactsTask* puts an appropriate part of the orchestrated operation on hold until the resultant Docker image is available for download from the registry, nowhere did the input document need to specify the image's full path, but only the components that cannot be inferred otherwise. The expansion assures that the version number (compliant with the above versioning policy) is used in place of the Docker image tag name.
- **Merging strategy** — as long as the second `<sdo:docker />` element references the first as a base, to denote that the second project's image is based on the first, the tasks resulting from the expansion will appropriately reference each other as predecessors. This will affect these tasks' execution order and effectively render the job compliant with the imposed merging strategy, dictating that the master branch shall only contain properly integrated code.

  For purposes of the situation found in the document C.1, the base library is only assumed to work (and thereby permitted to merge into the mainline branch) if the pipelines of its downstream dependencies complete successfully.

5.4 System Interfaces

Although the two document processing components, combined with the execution engine, amount to a robust, flexible, and extensible solution, they cannot be used on their own. They build into class libraries targeting the .NET Standard 2.0[2] and do not come with any interactive interface to actually process documents or to start and monitor jobs.

From a design standpoint, this forced the solution to implement all necessary functionality as part of the headless components, thereby preventing the leaking of critical services into the layer responsible for the user interface. Nevertheless, two such interfaces were created once the underlying system was deemed stable.

[2] C# projects can choose from a multitude of target platforms. Class libraries targeting .NET Standard version 2.0 can be used across major operating systems and specific .NET runtimes [1].

Web API

With respect to the solution's requirement of supporting the mode of operation of a hosted service, the ubiquitous means of exposing the system's class libraries' functionality was to expose a web API in the deployment environment's network. This approach carries on the so far upheld notion of openness and extensibility, as this allows any kind of user, human or programmatic, to create, monitor and control jobs.

The task of interconnecting the developed implementation with a web API server has been made easier by ASP.NET Core, a C# web application framework by Microsoft that provides first-class support for the creation of web APIs [19]. The solution includes a web API project, which provides a full-fledged HTTP server providing the API functionality. The implemented endpoints could be split into two categories:

- **REST endpoints** — expose the running jobs as a set of resources that can be manipulated by the individual methods of the HTTP protocol in a manner compliant with the REST principles of web APIs [19]. To illustrate, these endpoints implement operations such as creating jobs by uploading input documents, obtaining a list of running jobs, or detailed status of a specific one.
- **Real-time endpoints** — provide means of bi-directional real-time communication of clients with the API server to facilitate change notification. These endpoints were implemented using ASP.NET Core's SignalR library, which dynamically uses the most appropriate available technology for this kind of messaging, but typically utilizes WebSockets transport as long as both the client and the server support it [21]. To illustrate, these endpoints allow clients to subscribe to changes of a specific job. The server then informs the clients about select events published by the event system, such as variable change events.

As the solution is built upon the technique of dependency injection and Microsoft's dependency injection extensions, its incorporation into a web API application using ASP.NET Core was fairly straightforward and not particularly extraordinary. It ultimately does little more than setting up the system on startup and delegating incoming requests to one or more underlying components to prepare an answer.

Web Application

A single-page application developed with the React framework in JavaScript is provided to both demonstrate the web API's usage and provide a more appropriate interface for human users. The provided functionality reflects the available endpoints. It enables users to upload new input documents and browse existing jobs. It is also possible to view details of a specific job, control its execution status, observe the memory model, view the status of tasks, and relay commands to their executors.

Chapter 6

Deployment & Evaluation

Following the designed solution's successful implementation, it has since been deployed to production, where it has met with success as well. This has been achieved by rigorous testing during the whole development process. As a consequence, the system demonstrated its ability to complete the expected tasks many times before and much sooner than its first version was released for general use of the pilot team.

6.1 Deployment Options

Given their characteristic of being web applications, both interfaces developed to facilitate communication with the system communicate via the HTTP protocol. As such, their deployment can take the form of HTTP servers.

In the case of the API server, the primary build result is a .NET Core console application, taking the form of a DLL file. As the ASP.NET Core web application framework is used, the application, upon launching, starts an instance of the framework's custom HTTP server called *Kestrel* [19]. The deployment process therefore consists of securing a host computer with the .NET Core runtime in version 2.1 or compatible, and invoking the `dotnet Donkey.WebApi.dll` command or similar. Naturally, the server application relies upon other DLL files that are the build results of individual system components. These files must be located where the runtime can locate them, such as in the same directory as the web server's entry point assembly.

As for the web application, the build process transpiles its sources into a set of static files that can then be served from an arbitrary HTTP server. The configuration file only needs to be updated to specify an URL at which the API server can be reached.

Honoring the principles described in the initial chapters of this thesis, the solution adopts continuous integration and deployment. Both Git repositories in which the source code resides are configured so that every delivered commit triggers the start of a CI/CD pipeline. One of the repositories hosts the web application and executes build steps relevant to its technology stack. The other repository contains everything else: the core components, plugins, test suites, and the API server. As apparent in figure 6.1, the deployment stage is preceded by several other CI jobs which have to succeed in order for it to trigger.

In order to take advantage of the benefits described in the section 2.3, pipelines of both repositories are configured to produce a Docker image from within the packaging stage. Both images are prepared to expose a port on which an HTTP server listens, providing the corresponding functionality of that image. This allows the deployment job to be very

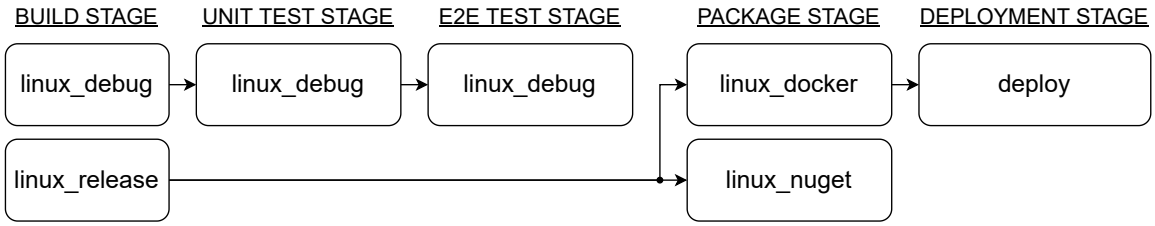

Figure 6.1: The API server's CI/CD pipeline visualized to show individual stages, jobs and the transfer of build result artifacts between them.

minimal. It only needs to instruct the host machine to download the target version of the appropriate image and cycle the container from which the service runs.

6.2 Test Suites

The solution's source code includes an extensive suite of test cases designed to verify the implemented components' correct behavior. While the coverage ratio and the approaches taken by the covering tests differ, the tests summarily provide a great degree of assurance that the product can deliver on its specified requirements. As apparent from the diagram in the figure 6.2, practically all components have their own dedicated test suite. The two exceptions contain only a limited amount of testable code and rely on their downstream dependencies for verification.

A majority of the implemented test cases were developed along with the functionality they test, in a fashion similar to test-driven development practices, and uses techniques described in section 2.1 such as mock and fake implementations of interfaces to achieve a sufficient level of isolation. Usage of these techniques was made easier by the constructor-based injection of dependencies by their interfaces.

However, in some cases, the coupling was too tight or the behavioral contract too complex to replicate or, while allowing the test to run, that replication took too large shortcuts. It thereby constituted too large of a deviation from the actual production use case. Nearly all of the implemented features build upon the memory model or, by extension, upon the event system, the latter of which introduces a degree of complexity due to its asynchronous mode of operation, complicating the verification of post-conditions. A part of the test suite originally used a simpler, synchronous variant of the event bus, but this ultimately allowed a defect to only surface in the End-to-End test suite. The differences of the simplified implementation then used by test suites of lower-level caused the issue not to reproduce in these environments. Therefore, the test suites were rewritten and no longer fake the functionality of the mentioned subsystems.

The designation of *unit tests* in the naming of the relevant source code projects and of the respective CI/CD pipeline stage is only a half-truth. The test cases' nature actually varies and includes both unit tests and integration tests, depending on the assumed point of view. Most of the tests typically instantiate a real instance of one or more underlying features and, as such, would better fit the label of integration tests. This permits a hypothetical bug in a depended upon component to incorrectly cause the dependent's tests to fail, as is typical of an integration test, but also brings forth all of that test type's advantages.

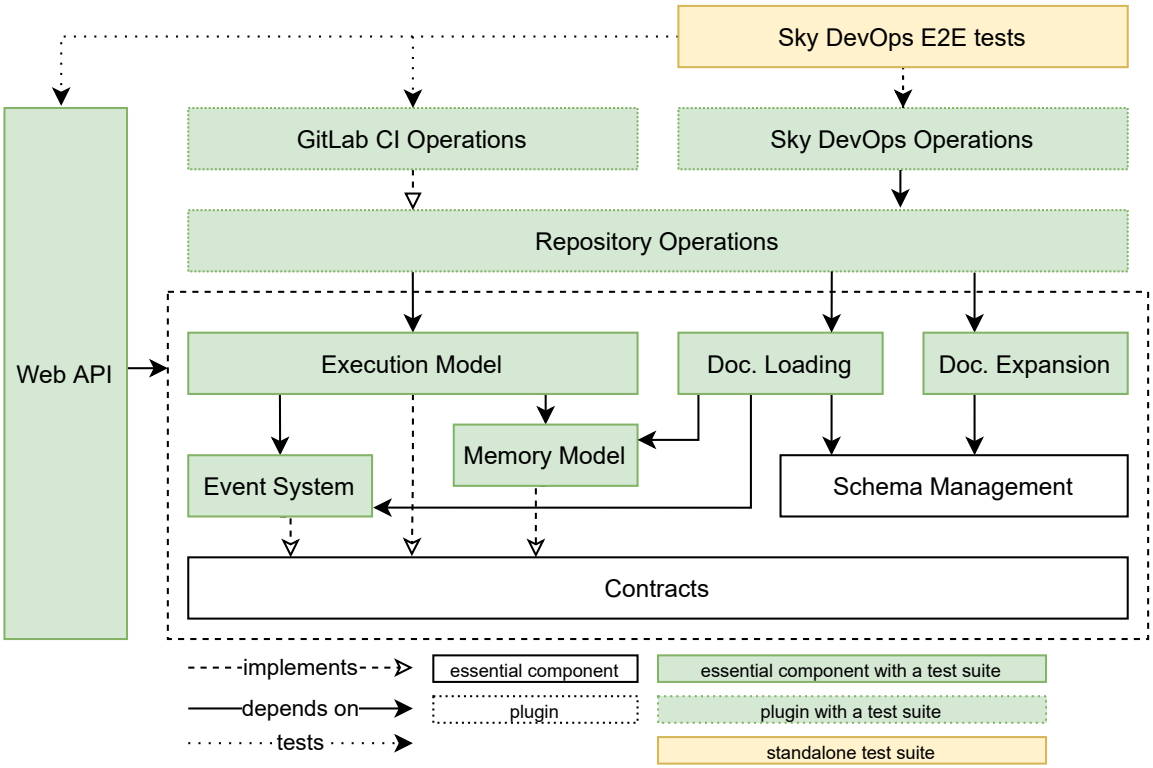

Figure 6.2: The coverage of system components by dedicated test suites.

Web API Test Suite

In addition to standard unit tests of controllers responsible for provided API endpoints, the test suite includes several *job flow* test cases. Their most fitting denomination would be that of end-to-end tests. They load a plugin that has nothing to do with orchestrating repository operations but provides an input document schema, some loaders, tasks, and task executors and use them to assert that the API server is integrating its subordinate system components properly. They verify that the API server fulfills its requirements of being capable of:

- creating a job instance based on a well-formed input document that requires expansion to take place,
- starting and stopping the created job's execution
- querying and issuing commands of the job's tasks
- reporting on the tasks' status
- reporting on the memory model's status

Sky DevOps End-to-End Test Suite

Finally, the solution also features a comprehensive end-to-end test suite ran only after all of the previously mentioned lower-level test suites pass. It has the ultimate scope, covering the entire system except for the web application. The web API is also leveraged

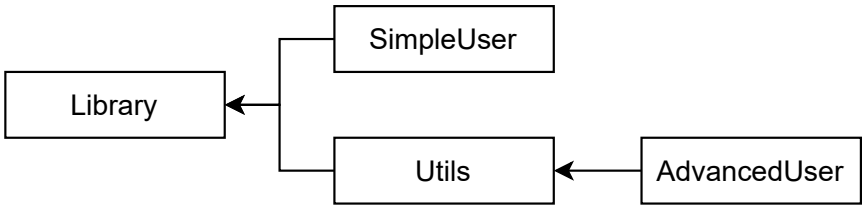

Figure 6.3: The dependency graph between test projects of the end-to-end tests suite.

here to deliver input documents in need of both expansion and loading. However, the documents now describe actual multirepo operations. In addition, the test suite establishes a test environment — consisting of real Git repositories, CI/CD pipelines, and produced artifacts — and allows the job to execute and run to completion. At last, the test suite accesses the artifacts resulting from the performed orchestration and inspects them to validate that all the requested changes were performed and in the correct order.

This suite's input documents address the greatest hurdle identified in chapter 4, constituting the inconvenience of cascading a change across downstream dependencies in a multirepo environment or the lack of such a cascade, where the responsibility is instead shifted onto the maintainers of these dependencies.

The test environment preparation phase establishes four repositories dependent on each other as visible in the figure 6.3 and consequently orchestrates a change delivery to the project depicted as the dependency graph's root. Each of these projects is initialized with a CI pipeline that produces a Docker image and uploads it to a container registry, mimicking the properties of actual build environments, whose build orchestration was the initial problem analyzed in the section 4.3 that the solution was created to solve. The capability to deliver on that requirement is verified every time the solution's feature branch is merged to mainline.

Chapter 7

Conclusion

Implementation of continuous nature into software integration, delivery, and deployment procedures has been shown to have positive impacts on an organization's performance, team cognition, and its other social aspects [7]. There are many prerequisites to consider before the first step in that direction should be taken — ranging from allocating time towards building the deployment pipeline to ensuring all team members are in the same boat. However, efforts invested this way are likely to manifest themselves along the way.

A CI/CD adopter might find themselves wondering about the individual repository models, deciding whether to opt for a single monorepo or to perhaps split their codebase into multirepos. This thesis provided a comparison of both models' features along with the inferred advantages and disadvantages. Multirepos' characteristics were described in detail to improve the knowledge gap of the currently available literature.

Based on observations in a software engineering team that utilizes multirepos, a set of common operations connected with development in such an environment was established. Subsequently, a user study was conducted to look into these operations' user comfort and factors that affect it negatively. The analysis of existing solutions related to the top two most uncomfortable operations, both related to the management of dependencies in multirepo environments, has shown a lack of universally applicable support tooling in the respective field.

For this purpose, the situation regarding dependency management lacking in user comfort has been analyzed and described further. A set of requirements was established, taking into account the needs of the aforementioned team and their current situation. A solution was designed to enable automation of the dependency-related operations and improve the user experience of related development workflows.

The designed solution was implemented and furnished with an extensive test suite that verified its ability to deliver on all instituted requirements. It offers an extensible orchestration engine configurable for a multitude of tasks, hosted as a web service. Nevertheless, this thesis only describes the first iteration of the solution's development. Having demonstrated its ability to orchestrate bulk multirepo operations in a way that allows better utilization of the involved software engineers' time and skills, it has become a valuable tool. It is being expanded to accommodate more types of repositories, artifact types, and build systems.

Bibliography

[1] Albahari, J. and Albahari, B. *C# 7.0 in a Nutshell*. 1st ed. O'Reilly Media, 2017. ISBN 9781491987650.

[2] Brito, G., Terra, R. and Valente, M. T. Monorepos: A Multivocal Literature Review. *CoRR*. 2018, abs/1810.09477. Available at: http://arxiv.org/abs/1810.09477.

[3] Brousse, N. The Issue of Monorepo and Polyrepo in Large Enterprises. In: *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*. New York, NY, USA: Association for Computing Machinery, 2019. Programming '19. DOI: 10.1145/3328433.3328435. ISBN 9781450362573. Available at: https://doi.org/10.1145/3328433.3328435.

[4] Chacon, S. and Straub, B. *Pro Git*. 2nd ed. Apress, 2014. The expert's voice. ISBN 9781484200766.

[5] Dietrich, J., Pearce, D., Stringer, J., Tahir, A. and Blincoe, K. Dependency Versioning in the Wild. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 2019, p. 349–359. DOI: 10.1109/MSR.2019.00061.

[6] Evertse, J. *Mastering GitLab 12*. 1st ed. Packt Publishing, 2019. ISBN 9781789531282.

[7] Forsgren, N., Humble, J. and Gene, K. *Accelerate – Building and Scaling High Performing Technology Organisations*. 1st ed. IT Revolution, 2018. ISBN 978-1942788331.

[8] Gao, S., Thompson, H., Sperberg McQueen, M., Beech, D., Mendelsohn, N. et al. *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. W3C Recommendation. W3C, Apr 2012. Available at: https://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/.

[9] GitLab Inc. *Cross project merge requests (&882)* [online]. Dec 2020 [cit. 2020-12-31]. Available at: https://gitlab.com/groups/gitlab-org/-/epics/882.

[10] GitLab Inc. *Group merge requests MVC (Super Merge) (#3427)* [online]. Dec 2020 [cit. 2020-12-31]. Available at: https://gitlab.com/gitlab-org/gitlab/-/issues/3427.

[11] GitLab Inc. *Multi-project pipelines* [online]. Dec 2020 [cit. 2020-12-31]. Available at: https://docs.gitlab.com/ee/ci/multi_project_pipelines.html.

[12] GitLab Inc. *One-click merge across multiple projects (&881)* [online]. Dec 2020 [cit. 2020-12-31]. Available at: https://gitlab.com/groups/gitlab-org/-/epics/881.

[13] Goode, D. and Rain. *Scaling Mercurial at Facebook* [online]. Jan 2014 [cit. 2020-12-31]. Available at: https://engineering.fb.com/2014/01/07/core-data/scaling-mercurial-at-facebook/.

[14] Harry, B. *The largest Git repo on the planet* [online]. May 2017 [cit. 2020-12-31]. Available at: https://devblogs.microsoft.com/bharry/the-largest-git-repo-on-the-planet/.

[15] Humble, J. and Farley, D. *Continuous Delivery*. Addison-Wesley, 2011. ISBN 0-321-60191-2.

[16] Itzhak, Y. *JFrog Artifactory as a Caching Mechanism for Package Managers* [online]. Dec 2019 [cit. 2020-12-31]. Available at: https://jfrog.com/blog/artifactory-as-a-caching-mechanism-for-package-managers/.

[17] Jaspan, C., Jorde, M., Knight, A., Sadowski, C., Smith, E. K. et al. Advantages and Disadvantages of a Monolithic Repository: A Case Study at Google. In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. New York, NY, USA: Association for Computing Machinery, 2018, p. 225–234. ICSE-SEIP '18. DOI: 10.1145/3183519.3183550. ISBN 9781450356596. Available at: https://doi.org/10.1145/3183519.3183550.

[18] Laster, B. *Jenkins 2: Up and Running*. 1st ed. O'Reilly Media, 2018. ISBN 9781491979594.

[19] Lock, A. *ASP.NET Core in Action*. Manning Publications, 2018. ISBN 9781617294617.

[20] Lück, N. *Optional — a robust option/maybe type for C#* [online]. Oct 2017 [cit. 2021-05-11]. Available at: https://github.com/nlkl/Optional/blob/fa0160d995af60e8378c28005d810ec5b74f2eef/README.md.

[21] Microsoft Corporation. *Introduction to ASP.NET Core SignalR* [online]. Nov 2019 [cit. 2021-05-11]. Available at: https://docs.microsoft.com/en-us/aspnet/core/signalr/introduction?view=aspnetcore-2.1.

[22] Patterson, C. *Creating a state machine* [online]. Jan 2017 [cit. 2021-05-11]. Available at: https://github.com/MassTransit/Automatonymous/blob/9ac7c4c16544ad80c6c51e9749b252b6f066d89b/docs/use/creating.md.

[23] Peabody, B. *User Guide for `git meta`* [online]. Mar 2017 [cit. 2020-12-31]. Available at: https://github.com/twosigma/git-meta/blob/ccb73a95881ea63d1658ef761dd63a2d17c93fde/doc/user-guide.md.

[24] Poppendieck, M. Lean Software Development. In: *29th International Conference on Software Engineering (ICSE'07 Companion)*. 2007, p. 165–166. DOI: 10.1109/ICSECOMPANION.2007.46.

[25] Potvin, R. and Levenberg, J. Why Google Stores Billions of Lines of Code in a Single Repository. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery. Jun 2016, vol. 59, no. 7, p. 78–87. DOI: 10.1145/2854146. ISSN 0001-0782. Available at: https://doi.org/10.1145/2854146.

[26] Preston Werner, T. *Semantic Versioning 2.0.0* [online]. Jun 2013 [cit. 2020-12-31]. Available at: https://semver.org.

[27] Seemann, M. *Dependency Ijection in .NET*. Manning Publications, 2012. ISBN 9781935182504.

[28] Shahin, M., Ali Babar, M. and Zhu, L. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access*. March 2017, PP. DOI: 10.1109/ACCESS.2017.2685629.

[29] Thompson, H., Tobin, R., Bray, T., Hollander, D. and Layman, A. *Namespaces in XML 1.0 (Third Edition)*. W3C Recommendation. W3C, Dec 2009. Available at: https://www.w3.org/TR/2009/REC-xml-names-20091208/.

[30] Zhang, Y., Vasilescu, B., Wang, H. and Filkov, V. One Size Does Not Fit All: An Empirical Study of Containerized Continuous Deployment Workflows. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2018, p. 295–306. ESEC/FSE 2018. DOI: 10.1145/3236024.3236033. ISBN 9781450355735. Available at: https://doi.org/10.1145/3236024.3236033.

Appendix A

Factor Identification User Study Form

Question 1

Please order the following tasks to reflect how convenient do you find their fulfilment to be using the tooling currently at your disposal. Please start with the least convenient workflow and descend down to the most convenient one.

Response Type: Ordinal Scale

Items to Order:

1. Navigating a hierarchy of multirepos
2. Establishing local repository clones
3. Working within local repository clones
4. Pushing changes from local repository clones
5. Full-text search through source code contained in multirepos
6. Updating source code to accept upstream updates
7. Ensuring that own changes are cascaded to downstream dependencies

Question 2

Is there a task that has been omitted from the first question that you would like to mention?
If so, please describe it briefly and note on which rank would you place it, had it originally been included in this survey.

Response Type: Free Response

Question 3

In relation to the task that you found most convenient, what aspects do you hold responsible?

Response Type: Multi-selection of predefined + 1 optional free response

Selection Items:

1. Feedback Time
2. Complexity
3. Mundanity
4. Context Switching

Appendix B

Factor Identification User Study Results

Response ID	Answer 1	Answer 2	Answer 3
1	7,6,5,1,3,4,2	-	2,3,4
2	6,7,1,5,4,3,2	-	1,2,4
3	7,6,5,1,4,3,2	-	2,3
4	7,6,1,5,3,4,2	-	2
5	2,4,3,1,5,6,7	-	2,4
6	5,7,1,6,4,2,3	-	2,4
7	2,4,3,5,1,6,7	-	2,4
8	6,7,5,1,2,4,3	sharing utility code between repos, 4th/5th	2,3
9	7,6,1,5,3,4,2	-	2,3,4

Table B.1: Verbatim answers received to the study conducted using the form in appendix A.

Coded Workflow	Order Sum	I1	I2	I3	I4	Prone to errors
Cascading downstr. dep.	24	0	4	3	2	0
Accepting upstr. dep.	26	1	2	1	1	1
Full-text search	31	0	1	1	0	0
Navigating hierarchy	33	-	-	-	-	-
Pushing changes	43	-	-	-	-	-
Working within local clones	47	-	-	-	-	-
Establishing local clones	48	0	2	0	2	0
Sharing util. code between repos	69	-	-	-	-	-

Table B.2: Workflows defined in section 4.1 and the one that was entered by a participant by the means of question #2, sorted in an ascending order by the sum of their order in study participants' recorded responses. Factor legend: I1 – Feedback Time, I2 – Complexity, I3 – Mundanity, I4 – Context Switching

Turnout: 9 out of 11 possible responses were received (82%).

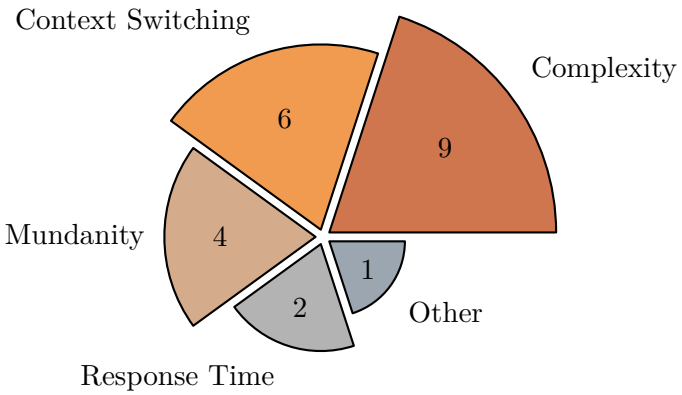

Figure B.1: The distribution of factors identified by the study participants as inconvenience–causing, regardless of the task that they found most inconvenient.

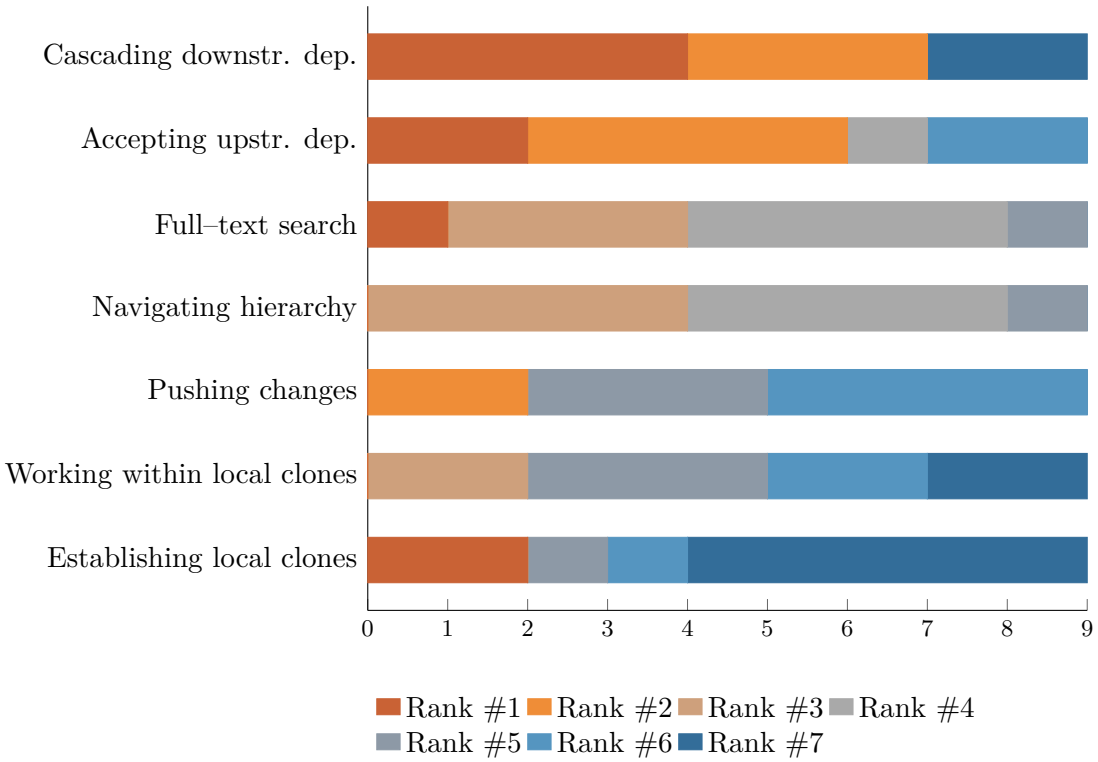

Figure B.2: The distribution of order of inconvenience that the study participants assigned to individual tasks.

Appendix C

Sample High–Level Input Document

To allow for a better reading experience, the sample high–level input document can be found on the next page, while the following paragraphs hint at the transformations that happen as part of the expansion process. Once all of the expansions are applied, the resultant document becomes low–level instead and can be loaded into a job instance. Unfortunately, the expansion growth is significant and the result would be impossible to show on a single page. If interested, please refer to the source code. The unit tests suite of Sky DevOps Subsystem contains several sample input and output documents similar to that in listing C.1.

Each of the two instances of the `<sdo:docker />` element in listing C.1 are expanded into five tasks, whose types are closely described in subsection 5.3 / Repository Operations Subsystem. The expansion honors rules and conventions applicable for the team for whose purposes was this particular expansion operation created. As such, these five tasks execute the below goals, while also verifying resultant artifacts (*VerifyArtifactsTask* following each triggered CI/CD pipeline):

- a feature branch is opened (*OpenChangesTask*)
- the described changes pushed (*SubmitChangesTask*)
- the remaining operations delayed until the CI pipeline automatically triggered by these changes spawns and succeeds (*SubmitChangesTask*)
- a merge request for the pushed changes is created, programmatically accepted (*MergeChangesTask*)
- the remaining operations delayed until the CI pipeline automatically triggered by the merged changes spawns and succeeds (*MergeChangesTask*)

The expansion process additionally ensures that every predecessor/dependency relationship, changes pushed to the parent component are not merged until it is confirmed that the downstream dependency can be built once it is updated to accept them.

Unless they are single–purpose, the values provided to the expanded elements' attributes are assigned to a newly created variable scope nested in the scope of global variables. Not only is it easier to monitor these values when they are grouped into this specially designated variable scope, it also is semantically appropriate, as the values are shared by all of the five tasks and the variables' ownership should not be forced upon any of them.

```
<dk:job xmlns:dk="http://sky.emea.thermo.com/Donkey.xsd"
        xmlns:repos="http://sky.emea.thermo.com/RepositoryOperations.xsd"
        xmlns:gitlab="http://sky.emea.thermo.com/GitLab.xsd"
        xmlns:sdo="http://sky.emea.thermo.com/SkyDevOps.xsd">
  <dk:globals>
    <gitlab:provider dk:name="GitLabProvider"
                     gitlab:hostname="gitlab.example.com"
                     gitlab:apiToken="123456789"
                     gitlab:username="john.doe"
                     gitlab:commitAuthorEmail="john.doe@example.com"
                     gitlab:commitAuthorName="John Doe" />
    <repos:dockerImageVerifier dk:name="DockerVerifier"
                               repos:hostname="cicd.example.com:9000"
                               repos:authentication="anonymous" />
  </dk:globals>
  <dk:tasks>
    <sdo:docker dk:id="base_library"
                sdo:project="dream_team/projects/base_library"
                sdo:imageRegistry="dream_team"
                sdo:imageRepository="base_library"
                sdo:imageName="linux"
                sdo:imageVerifierPath="globals.DockerVerifier">
      <sdo:changes>
        <repos:yamlNodeTextChange repos:file="library_data.yml"
                                  repos:path="library_data/sample_key"
                                  repos:value="New Value for Library" />
      </sdo:changes>
    </sdo:docker>
    <sdo:docker dk:id="library_user"
                sdo:project="dream_team/projects/library_user"
                sdo:base="base_library"
                sdo:imageRegistry="dream_team"
                sdo:imageRepository="library_user"
                sdo:imageName="linux"
                sdo:imageVerifierPath="globals.DockerVerifier">
      <sdo:changes>
        <repos:yamlNodeTextChange repos:file="simple_user_data.yml"
                                  repos:path="simple_user_data/sample_key"
                                  repos:value="New Value For User" />
      </sdo:changes>
    </sdo:docker>
  </dk:tasks>
  <dk:metadata>
    <repos:defaultProvider repos:path="globals.GitLabProvider" />
  </dk:metadata>
</dk:job>
```

Figure C.1: A sample XML document accepted by the implemented Sky DevOps Subsystem.