FH Vorarlberg
University of Applied Sciences

Shifting to Modern Software Development Infrastructures

Masterarbeit
zur Erlangung des akademischen Grades

Master of Science in Engineering (MSc)

Fachhochschule Vorarlberg
Masterstudiengang - Informatik

Betreut von
Dr. Peter Reiter

Vorgelegt von
Enes Eren, BSc
Dornbirn, 10.07.2022

Statutory Declaration

I declare that I have developed and written the enclosed work completely by myself, and have not used sources or means without declaration in the text. Any thoughts from others or literal quotations are clearly marked. This Master Thesis was not used in the same or in a similar version to achieve an academic degree nor has it been published elsewhere.

Dornbirn, am 10.07.2022 Enes Eren, BSc

Abstract

Legacy software systems are critical assets that carry vital business logic for companies. These systems however are difficult to maintain and extend because of the usage of outmoded technologies and procedures which make even minor changes to a challenge and therefore limit the organizations capability to stay competitive in the fast-growing technology industry. Yet, companies hesitate to switch to more modern software engineering approaches like DevOps, Continuous Integration and Delivery since there is an immense amount of literature and different concepts out there and researching all these topics and implementing the correct technology stack for their demands and needs would be a vast financial investment.

This thesis is focused on the evaluation and presentation of current literature in the field of software engineering concerned with state-of-the-art software infrastructures for software development and best practices.

As a result, this thesis explicates current principals of software configuration management, version control systems, branching and versioning strategies, dependency management, build automation, Continuous Integration and Continuous Delivery and modern software release management. Furthermore, as a proof of concept, a concrete and thorough strategy to shift from an existing legacy system to a postmodern automated software development infrastructure is implemented and presented, by applying several of the researched techniques and strategies.

Zusammenfassung

Legacy-Softwaresysteme sind Individualentwicklungen, die komplex sind und wichtige Geschäftslogiken für Unternehmen enthalten. Diese Systeme sind jedoch schwierig zu warten und zu erweitern, da veraltete Technologien und Verfahren verwendet werden. Selbst geringfügige Änderungen werden zur Herausforderung und dies schränkt die Fähigkeit vom Unternehmen ein, in der schnell wachsenden Technologiebranche wettbewerbsfähig zu bleiben. Dennoch zögern Unternehmen, auf modernere Software-Engineering Ansätze wie DevOps, Continuous Integration und Delivery umzusteigen. All diese Themen erforschen und den richtigen Technologie-Stack für ihre Anforderungen und Bedürfnisse zu implementieren, ist eine enorme finanzielle Belastung.
Der Fokus dieser Arbeit liegt auf der Auswertung und Darstellung aktueller Literatur im Bereich Software Engineering, die sich mit aktuellen Prinzipien und bewährten Methoden von Softwareinfrastrukturen für die Softwareentwicklung beschäftigt.
In dieser Arbeit werden daher aktuelle Prinzipien bezüglich Softwarekonfigurationsmanagement, Versionskontrollsysteme, Branching- und Versionierungsstrategien, Abhängigkeitsmanagement, Build-Automatisierung, Continuous Integration und Continuous Delivery sowie modernes Software-Release-Management erläutert. Darüber hinaus wird als Proof of Concept eine zur Umstellung eines bestehenden Altsystems auf eine postmoderne automatisierte Softwareentwicklungsinfrastruktur implementiert und vorgestellt, bei der mehrere der untersuchten Techniken und Strategien zum Einsatz kommen.

Contents

List of Figures

1 Introduction

Up until recently, integrating and delivering code was a time-consuming, laborious procedure. People from several teams had to get together to combine all of the code, solve any ad-hoc bugs, and then deploy it to the production environment. Only then the code built previously could provide business value [23] [87].

Nowadays, businesses must contend with fast changing competitive environments, expanding security needs, and scalability issues. Businesses must find a method to find a middle ground between the necessity for rapid product development and the requirement for operational stability. Yet still only fast software development is no longer adequate to compete in today's technology economy. Deployments must be more efficient, dependable and precise [87] [102]. A modern software engineering infrastructure allows for more frequent code changes while also improving and streamlining the product development cycle, which includes software release plans and automated software testing techniques, hence strategies like continuous integration and delivery (CI/CD) are implemented which allow for quick development updates while ensuring all the previously mentioned aspects [163].
If companies' software applications and infrastructures do not satisfy these digital business requirements these systems must be updated and enhanced to fit appropriately and to deliver higher business value. Systems that are not adaptable enough to keep up with the new needs of digital businesses might be expensive and risky, since these systems tend to be hard to maintain and adjust to modern business needs [103][102]. In spite of the fact that the advantages of modern software practices like Continuous Integration, Continuous Delivery and DevOps cannot be denied [36][51], many companies tend to hesitate to apply them, since shifting strategies and infrastructure come with needs of an initial financial investment and with obstacles such as efforts to meet required quality standards and delays related to testing limitations. These difficulties are even more increased when legacy applications lack proper documentation [113][103][134].

This is why this thesis proposes techniques for a state-of-the-art modern software development infrastructure, workflow and additionally techniques dealing with various types of re-engineering difficulties in order to modernize and re-platform legacy systems.

1.1 Modern Software Engineering Infrastructure

Modern software development is built on three pillars: flexibility, speed, and quality. Growing customer demand and the changing technical landscape have made software development more complicated than ever before, rendering traditional software development lifecycle (SDLC) methodologies, like the waterfall model, incapable of keeping up with the fast-paced nature of development [164]. Agile and DevOps practices have grown in favor as a way of providing these changing needs by adding agility and responsiveness to the development process without losing overall product quality. Continuous Integration (CD) and Continuous Delivery (CD) are two crucial components that aid in this process. It lets developers modify integrated development pipelines that span the whole software development process, from development to production deployments [162] [164].

1.2 Legacy System

A legacy system is a outmoded still in use computing software and/or infrastructure [156]. The system still may fulfill the requirements for which it was created, but it does not allow for expansion. A legacy system will only ever serve the company's current needs. The older technology prevents interacting with modern systems [156]. As technology progresses, most businesses are forced to cope with problems generated by legacy systems. A legacy system prevents a company from having the most modern capabilities and services, such more data management and automation, and keeps them in a slump in their business. [120] [2].
There are a variety of reasons why a corporation may stick with a legacy system. Modernizing to a new system necessitates an initial financial investment. Old systems may be built using obsolete technology, making migration difficult and there may be little to no documentation about the system, and the original developers may have left the company. Even the planning to change a legacy system and defining the requirements can be difficult at times [113].
Furthermore, preserving behaviour is a difficult task. It can be difficult to make changes and to maintain behaviour, especially when there are no existing tests for the old system [87][134].

2 Software Configuration Management

Software Configuration management (SCM) is the process of storing, retrieving, uniquely identifying, and modifying all artifacts important to a project, as well as the relationships between them [87]. It is a method for ensuring that a product's performance and functionalities are consistent with its requirements, design, and operational data throughout its life cycle, therefore ensuring an improvement in software quality. A configuration management strategy identifies how to handle all the changes that occur during the course of a project and ensures reproducibility of any parts of the project, including the version of the operating system, its patch level, the network configuration, the software stack, all the dependencies and their configuration [138][87].
It keeps track of the progress of systems and applications and consists of four major components: Configuration Identification, Configuration Auditing, Configuration Status Accounting and Configuration Change [62].
Configuration Identification is the act of identifying all of a project's components and ensuring that they can be identified quickly throughout the project life cycle. Configuration identification divides a project into smaller, easier-to-manage subprojects, such as design documents and unique graphic files [44][127].
Configuration Status Accounting keeps track of when, why and who makes changes to a project's source code. It ensures that as the product evolves through its life cycle, information about the product and product configuration information is captured, as well as historical transparency [45] [26].
Configuration Auditing is a procedure for ensuring that a project is on track and that the developers are producing exactly what is needed and it allows the tracking of advancement [109].
Configuration Change Control organizes team members access to project components so that data does not get lost, or unauthorized changes are performed. Most SCM systems have a check-in/check-out mechanism that allows user to write to a project file to prevent against lost changes. Current and prior versions of a file are identified and monitored, and user can request a copy of a previous version at any time [39] [127].

2.1 Version Control Systems

Within large, mature enterprises, requirements on software projects grow over the years and with every modification and technology addition, from many developers and application managers the underlying codebase of a project and it's files will evolve. With many contributors and changes the necessity to maintain the projects and it's code robustness is critical, in which version control systems (VCSs) play an important role [176] [25].
It is a technique for tracking changes of a group of files over time so that certain versions can be accessed. It allows to revert chosen files or the complete software project to a former state, analyse changes over time, determine who last edited source code that causes a problem, and when an issue was created [66]. Changes made in one section of the software may conflict with changes made by some other developer working on the same project at the same time. With VCS these issues can be identified and resolved so that the rest of the teams' work is not affected [87] [25]. To not intervene with the work of others every version control system supports a mechanism called branching (See 2.2). Branching allows to separate from the main development code and continuing to work on a copy of the main code [31]. Use of a VCS also means that mistakes are more easily to be recovered. Every item associated with the development of software should be version controlled such as source code, tests, build and deployment scripts, database scripts, libraries, documentation and configuration files [66][25]. Although it is feasible to develop software in the absence of utilizing version control, it exposes the software project to a significant risk. So, the question is not whether a version control should be used, but which VCS [87].

2.1.1 Centralized Version Control System and Distributed Version Control System

There are two sorts of VCSs: Centralized version control systems (CVCS) and Distributed version control systems (DVCS) [99].
In a centralized VCS a server serves as the primary repository, a storage for projects files, for every version of the project files. There is a single server that houses all the versioned files in these systems, such as Subversion, also called SVN and Perforce, and several clients that check out files from that central server as illustrated in Figure 2.1 [148][87][66]. When employing centralized source control, each user directly contributes to the one and only server, which allows for quick communication between team members on small teams. Therefore, collaboration and communication must be efficient if a centralized process is to be productive. [66]. A centralized workflow, like a client-server architecture, enables file

locking, ensuring that any code that is currently checked out is not available to others, and that only a single developer may contribute to the code at a time. Developers contribute to the central repository via branches, and the server unlocks files after merges [155]. This configuration has numerous advantages. For instance, everyone on the project is aware of what everyone else is doing to some extent. Administrators have finer control over who can do what and administering a CVCS is significantly easier than dealing with local databases on each client. However, this setup has a number of serious flaws. The centralized server's single point of failure is the most obvious. [155][66]. No developer is able to commit and save their work to whatever if the server is offline. The whole history of the project will be destroyed if the central database's hard drive malfunctions and there are no sufficient backups produced, with the exception of any individual snapshots that users may keep on their own workstations. [66].

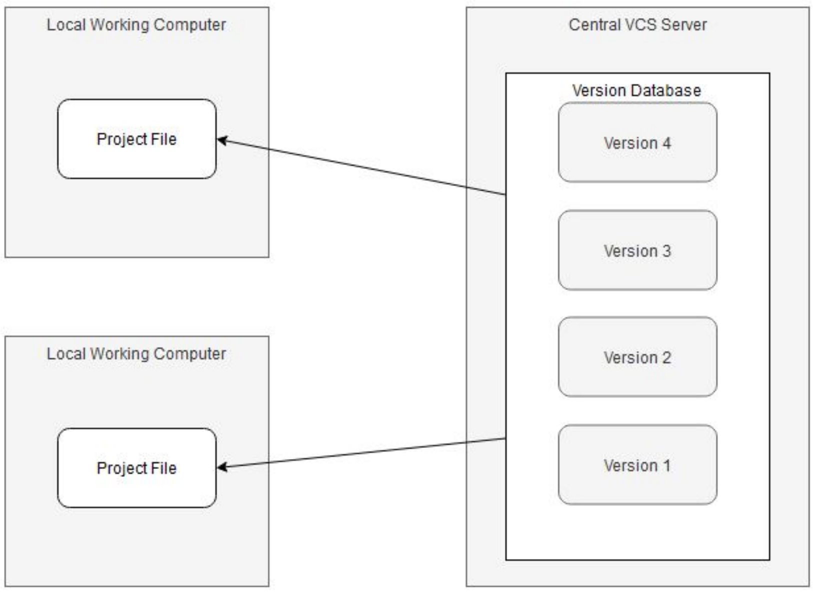


Figure 2.1: Centralized Version Control System

Distributed version control systems (DVCS), such as Git (See 2.1.2) and Mercurial (See 2.1.3, do not simply check out the most recent state of the files. User completely copy the repository, including its whole commit history, as shown in Figure 2.2 [66]. As a result, if any of the servers dies while these systems were cooperating through it, any of the user repositories may be transferred back to the server to restore it. Every clone is a complete backup of all data [66] [35]. DVCS allow users to change project files, also called committing, branch, and merge changes locally. The server just requires the differences between each commit and does not need to store a physical file for each branch. With each developer uploading code changes to their own repository and administrators putting up a code review policy to guarantee exclusively quality code merges into the main repository, distributed version control systems allow software development teams construct robust workflows and hierarchies [136]. Additionally, many of these systems work well with multiple remote repositories, enabling concurrent collaboration with distinct development teams throughout the same project. This makes it possible to develop processes like hierarchical models that are not offered by centralized systems [136] [87]. Yet, DVCS have a bottleneck if clients need to manage and work with binary files, since they require huge amount of spaces [99].

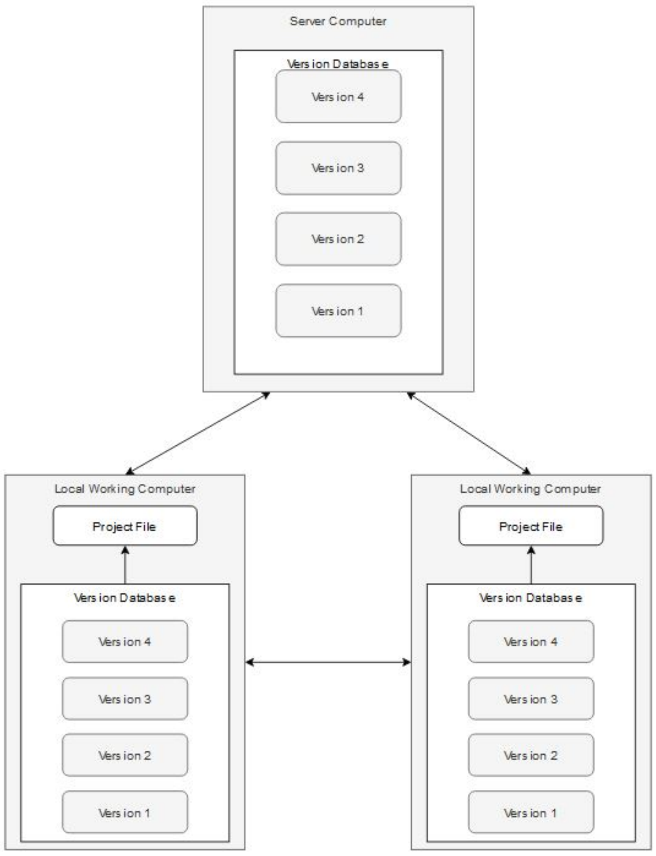


Figure 2.2: Distributed Version Control System

2.1.2 Git

Git is a Distributed Version Control System and like many of today's most popular VCS systems, is free and open-source. Rather than having a single repository for the whole archive of the software, as is the case with once-popular version control systems like Subversion (See 2.1.4), Git allows to have an entire copy of the code on the local machines of the developers and work on it without the need of remote connection to the repository. Git is also utilized to track changes in the source code, to allow numerous engineers to work cooperatively on one code base without intervening with each other and support non-linear development since parallel branching is allowed [66] [24].

The way Git sees its data is what sets it apart from any other VCS. Most other systems save data in the form of a list of file-based modifications. Other systems like SVN (See 2.1.4) and Perforce (See 2.1.5) consider the data they store as a collection of files, with changes made to each item over time which is called delta-based version control as shown in Figure 2.3 [66].

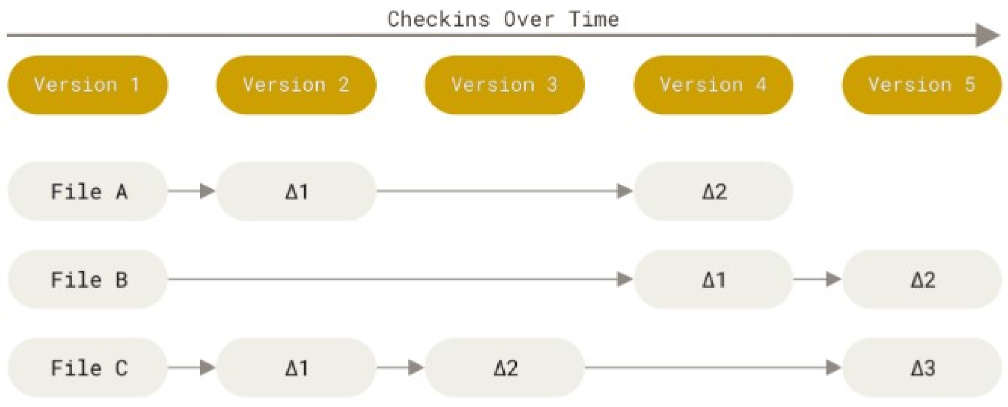


Figure 2.3: Delta Based Version Control
Source: `https://git-scm.com/book/en/v2`

Git views its data as a collection of snapshots of a small filesystem. With every commit, Git saves the current state of the files and keeps a reference on it. Git does not store the file again if it has not changed, instead providing a link to the prior identical file it has previously stored as shown in Figure 2.4 [25] [24]. That is why Git's branching system is extremely light, making branching operations almost instantaneous and toggling back and forth between branches quick. Git, unlike many other VCSs, promotes workflows that branch and merge frequently, even multiple times per day [66].

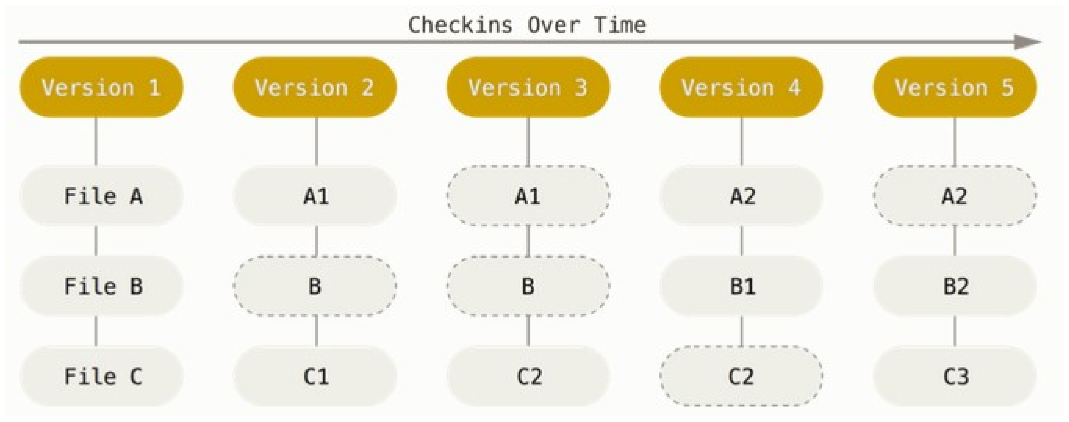


Figure 2.4: Snapshot of Files in Git
Source: `https://git-scm.com/book/en/v2`

Since Git allows working on local copies, most Git operations require just local files and resources to run. Information from other machines on the network is rarely required. Unlike CVCS, where most activities have a network latency costs [25] [24].
Advantages of Git:

1. It is simple to branch and merge: It is a part of the working process. Branching and merging are efficient and take up minimal storage. Branching allows to test features and ideas before releasing them to the public [66].

2. The workflow is adaptable: When compared to a centralized VCS, git has the ability to create a customized workflow [24].

3. Git cryptographically hashes its content in order to protect files from corruption due to disk or network failures [66].

Disadvantages of Git:

1. A steep learning curve: There are a lot of commands with a lot of choices, some of them are non-intuitive and require knowledge of git's internals [100].

2. Binary files: If projects contain often updated non-text files, git will grow bloated and slow [65].

2.1.3 Mercurial

Mercurial is a distributed source control management program that is free to use. It offers the ability to manage projects of any size. It is simple to use and difficult to break, which makes it excellent for anyone working with versioned files [111].
In Mercurial creating new changes and branches, moving changes around, and running history and status procedures are all quick. Mercurial is nimble and by mixing low cognitive overhead with fast operations. Mercurial's utility is not restricted to projects of a certain size [118]. While Mercurial repositories can have many development branches, branching is usually accomplished via cloning a repository in its entirety. Mercurial can produce independent lightweight clones of whole repositories in seconds utilizing hard links and copy-on-write techniques. This is a key feature of Mercurial's distributed model, since branches are inexpensive and easily discarded [106].
It is well-suited to scripting chores, and its clear internals and Python implementation make it simple to add new functionality via extensions, if the primary features are not enough [118].
The Mercurial system is built on a storage method called "revlog," or revision log, which strives to achieve O(1) seek efficiency for both reading and writing revisions while maintaining effective compression and integrity. To calculate the position of the appropriate record in the index and read it to find a specific revision is fast and simple [106]. It will contain a pointer to the delta chain's first full revision, allowing the entire chain to be read in one contiguous chunk. Therefore to locate and retrieve a revision, only O(1) is required. For each file maintained, it keeps a separate index and data file. Mercurial employs the method of storing a cache of controlled file sizes and timestamps for future commits to identify file changes quickly. When a changeset is checked out, Mercurial also informs which files need to be updated. It naturally keeps track of which changeset the current working directory is based on, and will have two such parents in the case of merge operations [106].
Commits are atomic and are properly arranged such that developers do not have to lock anything. When a changelog entry to the index is added, it becomes available to developers. Additionally, tracking is also done for commit operations [146].
Advantages of Mercurial:

1. Safe history: Mercurial includes only one single command that can alter history:`hg rollback`. Other DVCS such as Git allow altering the history of repositories, which can cause unwanted changes [100].

2. Extendible: Mercurial‘s features are easy to extend with Python [112].

3. Simplicity: Mercurial is easy to learn and use, which is useful for less-technical oriented users [100].

Drawbacks of Mercurial:

1. No partial checkouts, which is a big limitation for large projects [100].
2. Rolling back bigger changes is cumbersome [100]

2.1.4 Subversion

Subversion, also called SVN, is utilized to maintain current and past versions of software projects and is an open-source centralized version control system. It was created to help programmers coordinate their efforts. SVN allows to keep track of and collaborate with team members in the same workspace [173] [130]. Every file in the project is saved with a file location and a revision number by Subversion. Because it does not host any files, its data consists of links to files rather than actual files, meaning the repository with the whole file history is only available on the Subversion server. By default, SVN keeps all code and accompanying metadata on a single server. To retrieve a copy of the code from a certain repository, client systems must connect to the server, as shown in Figure 2.5. It also creates an XML log file, making it simple to examine what has changed and who made the changes. Any changes to a file can be undone [144].

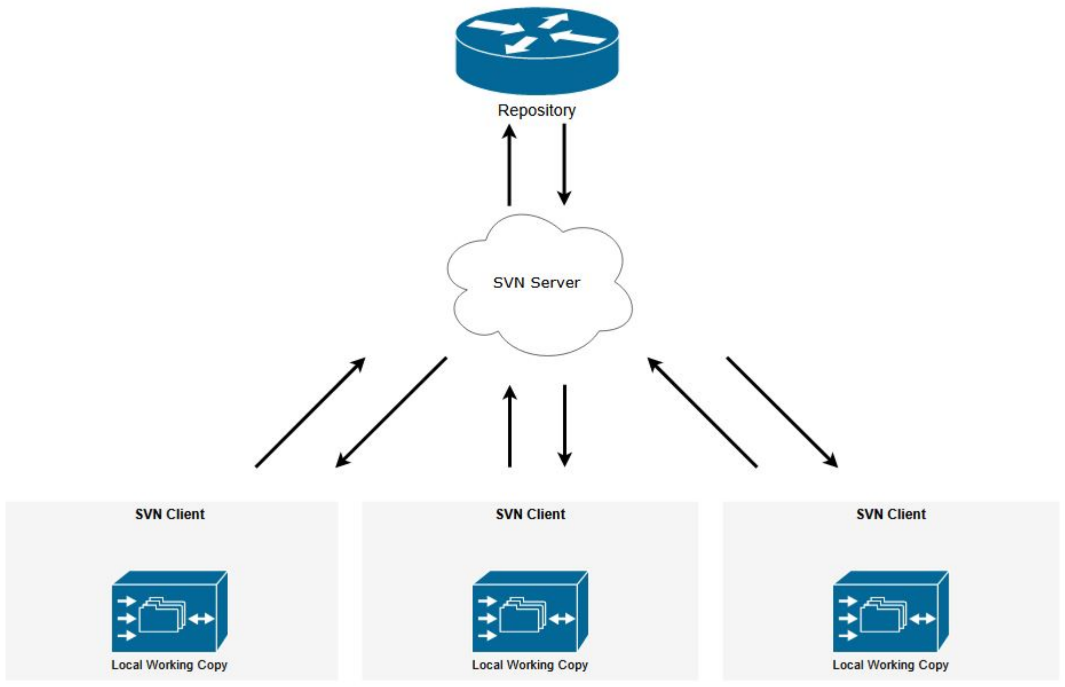


Figure 2.5: Components of SVN

Advantages of SVN:

1. SVN retains a complete revision history and allows to lock files that cannot be merged [131].
2. It enables symbolic link versioning as well as path-based authorization [173].

Since SVN is older there are some drawbacks of using it:

1. Because Subversion is centralized, it is not suited for open-source projects [131].
2. SVN is slower then Git [144]
3. Because the repositories are hosted on a single server, users must remain on top of updates [173]
4. Changes made since the last backup will be lost if the SVN server fails [173].

2.1.5 Perforce - Helix Core

Helix Core is a centralized version control system by Perforce. Keeping everything in one location guarantees that developers are always working with the most recent version. All modifications made by developers are committed to a central server, regardless of where they are situated [80]. Having a single copy of a project across an organization establishes a sole source of truth. Despite being centralized, Helix Core securely enables remote sites through replica and proxy servers. Because most activities are performed locally, this increases performance. Helix Core was designed to be fast and scalable. It has the capacity to process immense amount of transactions, files, and petabytes of data. Developers may check if they have the most recent version of a file on their workstation fast and conveniently [70] [121]. It also has exclusive locking for huge binary files. This keeps team members from stumbling over one another's work. Artifacts are kept with source code and other non-code resources and can be checked in with relevant source code in the same change list and since everything is stored on a single server it simplifies development and deployment workflows [80][70].
As shown in Figure 2.6 the key components of Helix Core are:

1. Workspace: Directories and folder on a local workstation where developers work on revisions of files and artifacts that are managed by the Helix Core server [80].

2. Helix Core application: An Helix Core application, like P4V [80], runs on the local workspaces of clients and make requests to the Helix Core server and fetches results, such as status information about artifacts and files [80].

3. Helix Core server: Responds to the requests of a Helix Core application, maintains files and tracks the state of workspaces [80].

4. Depot: The depot is a file repository hosted on the Helix Core server, which contains the complete history of every versions of all artifacts and files. A single Helix Core server can host multiple depots [80].

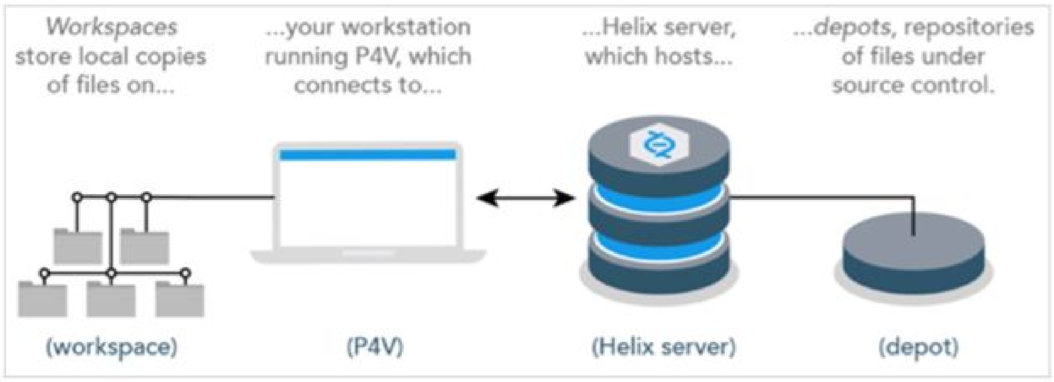

Workspaces store local copies of files on...
...your workstation running P4V, which connects to...
...Helix server, which hosts...
...depots, repositories of files under source control.
(workspace)
(P4V)
(Helix server)
(depot)

Figure 2.6: Perforce Components
Source:
`https://www.perforce.com/manuals/p4v/Content/P4V/Home-p4v.html`

The source code, artifacts and binary files that the Helix Core server manages are stored in a depot. Clients open and modify files in their local workspace. After that, they use a change list to send updated files to the depot after editing. The depot maintains track of all file revisions, both current and past [80].
Advantages of Perforce:

1. Managing Binary Files: Artifacts are kept with source code and other non-code materials in Helix Core, eliminating the need for an external binary file server, which results in a more simple workflow [121] [70].
2. Partial checkouts: Perforce supports partial checkouts [70].

Drawbacks of Perforce:

1. Cost: Perforce is more expensive compared to other version control systems [21].
2. Branching on per-file basis: Perforce creates an immense amount of metadata in the Perforce database for every branch that is created. Which may contribute to performance issues [21].

2.2 Branching

A branch is a copy of a code, managed in a VCS [125]. Through branching software development teams can work on separate portions of a project without affecting each other. Using branching, teams can better organize their work on a common codebase.

Most of the version control systems have own naming convention for the main branch which is the default branch of a project and contains the primary project state. Git‘s default branch is called master, perforce has mainline and SVN has trunk [31] [87]. Developers create branches, either directly or indirectly from the default branch to work in isolation. As a result, the entire product remains stable. It is best to keep these working branches up to date with changes in relevant code lines. Software branching establishes a link between the branch and the code from which it got created [125]. Other users may be contributing changes to the same primary code line while developers are working on their own branch [31] [125]. However, if branches are not properly managed, they can quickly grow bloated and unmanageable, undermining the goal of branching and version source control. Following a suitable branching strategy (See Branching Strategy) for all development needs is one of the best approaches to keep everything organized [177].

Branching Strategy

Branching strategies focus on how branches are employed, created and named in the development process. Simply stated, while working with a version control system for developing and maintaining code, a software development team employs a branching strategy in order to organize the VCS in use and keep it clean [78]. One major goal of a version control system is to enable a collaborative development platform without causing code to overlap or be affected. Each team member working on the same source code will eventually make incompatible changes [77]. With a version control system in use with a branching strategy, however, such disputes can be prevented. One of many keys to developing an effective modern infrastructure process is a well-implemented branching strategy, since it aids in the definition of how the delivery team works and how each feature, enhancement, or bug fix is addressed. It also simplifies the development and delivery process by letting developers concentrate on developing and deploying only the relevant branches of the product, rather than the complete product[177]. Since there are many strategies out there, the branching strategy selection depends on the project and user needs. This decision is influenced by factors such as the development approach, scale, and user preferences. Other considerations, like as CI/CD tools, also influence which branching techniques can be used in a CI/CD pipeline [77].

2.2.1 Trunk Based Development

Trunk-based development (TBD), commonly known as "mainline development," is a branching approach, in which all branches branch out from a single trunk/main branch.

All developers integrate their modifications directly into a shared trunk (master) regularly. This shared trunk is always in a safe and functioning state. As shown in Figure 2.7, with TBD developers pull code from this trunk, store it locally, and subsequently push it to the common trunk [177] [22]. This regular integration allows developers to instantly see one other's changes and react fast if any problems arise. Developers and release engineers rarely branch under this model. If the rare case occurs that they branch, they create a temporary feature branch that is evaluated and then merged back into the primary branch. By doing this, teams may avoid the difficulties of branching and merging [78] [22]. Additionally, TBD helps to keep production release flow as the project complexity and the amount of contributors grows [22].

Benefits of Trunk Based Development:

1. Better overview: Smaller iterations help teams to keep track of all changes, reduce code conflicts, and improve overall code quality [78].
2. Less merge conflicts: All other branches have a specified and restricted lifespan since the primary master branch is the only branch that is long-living. This reduces larger merge conflicts by ensuring that branches are not left in development for too long [78].
3. More efficient code reviews: Code review is more efficient with trunk-based development's quick, tiny commits. Developers can rapidly observe and review small changes with small branches [177].
4. Always in a deployable state: Trunk-based development aims to keep the trunk branch available for deployment at any time. This allows the team to deploy regularly to production and create new daily production release goals [22][177].

Drawbacks of Trunk Based Development:

1. Collision: Contention collision is one of the most difficult aspects of trunk-based development. It will be in a perpetual state of churn if too many individuals check changes into the mainline simultaneously. Developers can wind up stumbling over one another and ruining builds all the time. [177] [78]
2. Direct interaction with main branch: Because they are directly engaging with the shared trunk, inexperienced developers may find this technique cumbersome [177].

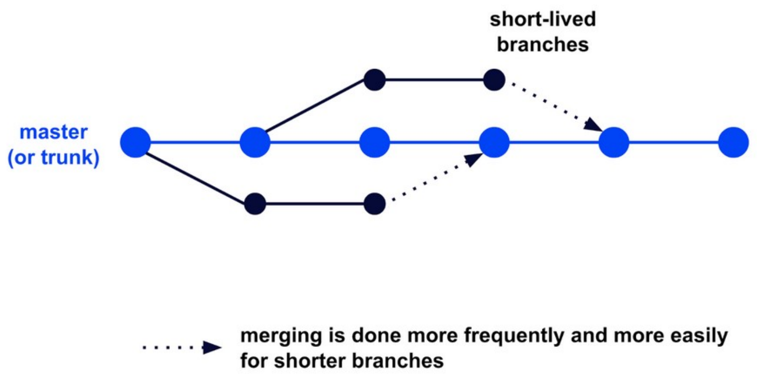


Figure 2.7: Trunk Based Development
Source: `https://www.optimizely.com/optimization-glossary/trunk-based-development/`

2.2.2 Release Branching Strategy

Most software development processes follow a simple pattern: code, test, release, and repeat. There are two issues with this method. First, developers must continue to write new features while quality assurance teams test presumably stable software versions. While the software is being tested, new work cannot be done. Second, the team has to support previous, released software versions. If a bug is detected in the most recent code, it is almost certain that it also exists in released versions, and customers will want to acquire that bug patch without having to wait for a major new release [69] [177]. The Release Branching strategy solves this with the following steps [77].

1. The development team commits all new code to the primary branch.
2. When the developers are sure that the current stage of the trunk is ready to be released, the trunk gets copied to a "release" branch, for example named "release/x" where as "x" can be any kind of string, depending on the versioning strategy (See 2.3) in use.
3. While the QA (Quality Assurance) Team is testing the newly created release branch, the developers can continue working on the new release on the trunk. Whenever bugs are located in either branch, before the release, fixes are ported into both branches.

4. After the testing phase, the "release/x" branch is copied to the "tag/x" branch, indicating a steady code, which will then be then released.

5. After the release, the branch is still maintained. While working on the newest features and releases, bug fixes continue to be merged into the release branch. When the development team agrees that the amount of fixes are sufficient enough, a new release is getting created.

2.2.3 Develop Branch Strategy

In this strategy, a permanent branch called develop is setup beside the master branch with this strategy. All work goes into the develop branch first. This is a secure location where new code is tested without risking breaking the project. Additionally, a testing strategy in place is needed to guarantee that merges do not introduce bugs into the primary branch. The main branch has the official release history, while the develop branch is used for feature integration as shown in Figure 2.8. The project's whole history will be in the develop branch, while a truncated version will be in the main branch [20] [177].

Benefits of Develop Branch Strategy:

1. As long as tentative development is done on the develop branch, the master branch remains in a stable state.

2. While a feature is being implemented, a hotfix can be applied to the master branch at any moment.

Drawbacks of Develop Branch Strategy:

1. Multiple features cannot be created at the same time.

2. It is difficult to remove and restore functionality using only the develop branch.

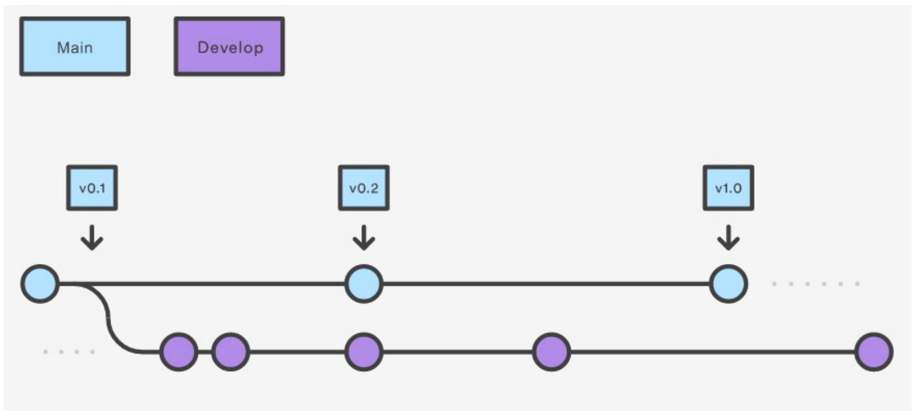


Figure 2.8: Develop Branch Strategy
Source: `https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow`

2.2.4 Feature Branching Strategy

Feature driven development, also known as feature-based development, divides a product's branches based on its features. Software teams plan, model, and integrate in terms of features. This approach is closely lined up with agile approaches and is a common GitFlow strategy and by splitting up development by features it can support teams to move quicker [78][30]. This also enables that, features can be tested independently and then included into to the mainline of the software project whenever they are ready to be merged into the primary branch, as seen in Figure 2.9. There is never a straight commit to the primary branch. Even the simplest modifications are made on a feature branch and then merged into the primary branch [78]. This encapsulation allows developers to work on a specific feature without interfering with the main codebase. It also guarantees that the primary branch will never include broken code, which is extremely beneficial in continuous integration setups. Using feature development also allows to use pull requests, which are a way to start conversations about a branch and lets other developers to approve a feature before it is merged into the main project [19] [77].

Benefits of Feature Based Development:

1. Stable code: Allowing developers to develop and work separately from the primary product can help maintain the stability of a code base [77]

2. Scalable: Large-scale projects can be managed more simply using feature-driven development. Innumerable amount of developers can be assigned to certain features. Code can then be tested more simply before being merging to the main code base [78]

3. Administrable: If the software development apartment consists of a huge amount of teams, which are topographically distributed, this approach allows for more control over where and what is being merged [78]

Drawbacks of Feature Based Development:

1. Long-living Branches: The aim of feature branches is for the branch to exist as long as the feature is under development. Long-lived feature branches may be cumbersome to merge. Because developers may operate in solitude for lengthy periods of time in order to prevent disputes. Furthermore, when a feature branch is left in development for too long, it may conflict with other branches during merging [77][78].

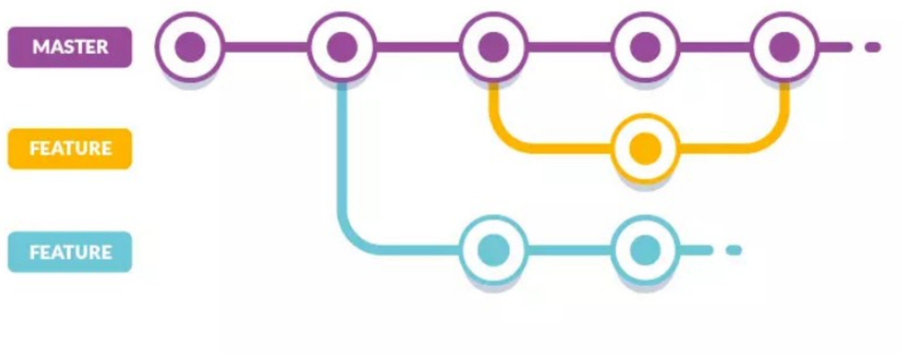


Figure 2.9: Feature Based Development
Source: `https://buddy.works/blog/5-types-of-git-workflows`

2.2.5 Gitflow

Gitflow combines existing branching mechanisms like Release Branching Strategy (See 2.2.2), Feature Branching Strategy (See 2.2.4) and Develop Branch Strategy (See 2.2.3) while expanding their functionality [85].

Gitflow consists of two primary branches, master and develop that have unlimited lifespan. The master branch is the primary branch, where source code is always in a stable and production-ready form. Develop is the primary branch, where the source code is at all times in a state where the latest development modifications for the upcoming release have been provided [177][6].
When the source code on the develop branch is stabilized and is ready for release, all the modifications must be integrated back into master and a release number assigned. Here the applied release number, depends on the current release number and applied versioning strategy (See. 2.3). As a result, every time modifications are merged back into master, a new production release is created [20].
Furthermore, Gitflow uses a range of supporting branches in addition to the main branches, master and develop, to enable parallel work across team members, arrange for production releases, track features and assist in quickly fixing issues in already delivered products [177].
The supporting branches are Release branches, Feature branches, which are inspired by Releasing Branching Strategy (See 2.2.2) and Feature Based Strategy (See 2.2.4), and on top of these Gitflow also uses Hotfix branches. Unlike the main branches, these branches have a finite lifespan because they will be eliminated at some point. Each and every branch has a distinct use and is constrained by strict criteria regarding which branch can serve as the origin branch and which branches may serve as a merge target. [6] [20].
Feature branches are applied to build new functionalities for upcoming releases. When developing a feature, the target version for which it will be included may be uncertain at the time. A feature branch survives for the duration of the feature's development, but it will be merged back into develop and then deleted [6].
Release branches are used to prepare new production releases. They also enable minor bug fixes and the preparation of meta data for a release. Because all the work is done on a release branch, the develop branch is free to accept features for the next major release. When the develop branch presents the desired state of a new release, it is time to branch out a new release branch from develop. At this point in time, all features intended for the current release have to be merged. All features planed for future releases may not be included, they wait until the release branch is split off [68] [69].
Hotfix branches are similar to release branches since they are used to prepare for an unplanned new production release. They result from the need to respond quickly to an undesirable state of an already released version of a product. When a major problem in a production version has to be fixed right away, a hotfix branch can be created by branching off from the master branch's matching tag that designates the production version.

The key point of hotfix branching is that team members' work on the develop branch may continue while other developers prepare a short production patch [6] [68].
Advantages of Gitflow:

1. Organized: Because this technique has separate and clear branches for particular goals, developers can better organize their work with the many sorts of branches. Additionally, clearly defined branches that aid in defining the test scope and allowing just specified branches to be tested [177].

2. Versioning: When dealing with numerous versions of the production code, Gitflow is ideal [68].

3. Scalable: Development is simple to scale. By separating features and guaranteeing that developers never has to block either development or master branch for release preparation, it facilitates parallel processing and CD [116].

Drawbacks of Gitflow:

1. Complex: Many branches with complicated regulations.

2. Maintenance: Maintenance is difficult because of the sheer amount of branches, especially long-lived branches [68].

3. Overhead: Depending on the complexity of the project, this method may overcomplicate source control [177].

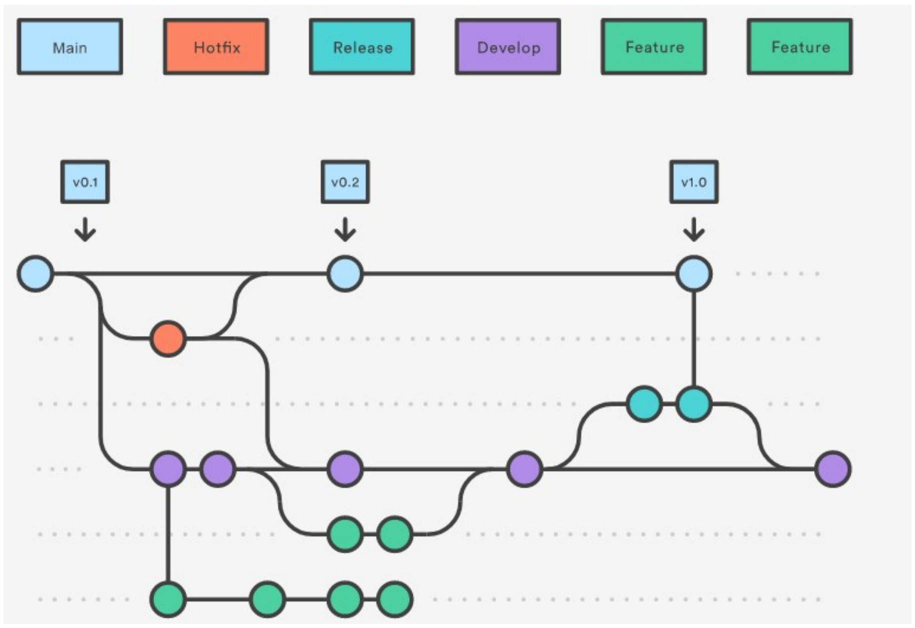


Figure 2.10: Gitflow
Source: `https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow`

2.3 Versioning

Software Versioning allows software development teams to keep track of the changes that are made to project files and simultaneously allow to identify each build artifact rapidly and easily. Using a software versioning strategy for projects helps everyone involved in producing and using an application to work more efficiently. Developers know exactly which features and changes are included in which version and customers can report the application version, which they use, when a e.g. a bug is found. Since there are multiple versioning strategies it is up to the needs of a product which versioning strategy to utilize [150] [175].

2.3.1 Semantic Versioning

Semantic Versioning is a defined method of providing meaning to software releases. It allows software developers to communicate crucial information about versions to their customers in a precise form. A SemVer version number is made up of three parts, separated by periods, Major, Minor and Patch as seen in Figure 2.11. Depending on what was modified in the revision, each number represents a different degree of revision. Reading from left to right, the number reflects the current Major release, the current Minor release, and the current Patch release. Each number version has its own interpretation [150] [58].

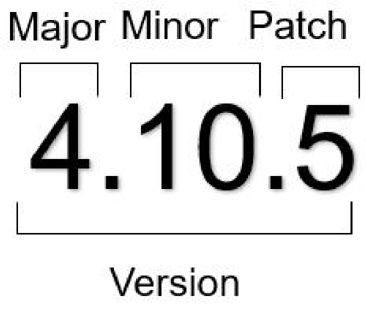


Figure 2.11: Semantic Versioning

A Major version has to be increased, if a release has any backward-incompatible breaking changes. This has the advantage of making it simple for anybody to determine if a new version will behave differently than a prior one.
The Minor version must be incremented whenever backward compatible functionality is implemented, which implies that customers should manage to update to a new minor version without encountering any problems [58][150].
A Patch release denotes that the code updates in this revision did not introduce any new features or API changes, and that it is backward compatible with earlier versions. It's most commonly used to denote a problem fix. Additionally, it indicates that the usage of the product did not change.
With these three numbers, projects can quickly determine all of the compatibility information a client requires and determine whether or not a client should update to the most recent revision and how much effort it will entail. Authors can also send highly essential information to their software's users using these three numbers. [58] [37].
Pre-releases are also possible with SemVer as illustrated in Figure 2.12. Developers can add certain optional labels after the patch, such as a pre-release label or a build number. To mark a package as a pre-release, developers must include a hyphen before the pre-release label, which can be a dash-separated identification. A pre-release indicates an unstable package and poses a high risk if utilized [150].

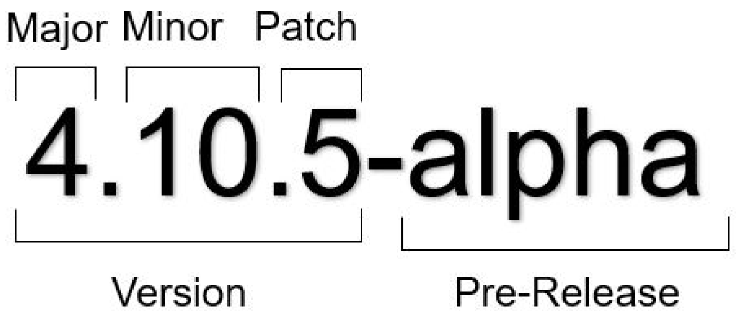


Figure 2.12: Semantic Versioning with pre-release

2.3.2 Other Versioning Strategies

There are multiple versioning strategies besides Semantic Versioning.

Calendar Versioning

Calendar Versioning, also called CalVer is a versioning technique that employs numbers that correlate to the day that a particular version will be released. There are a few segments of numbers and modifiers that can make up Calendar Versioning. The first segment often contains year information, which might be short year (YY), zero-padded year (0Y, relative to year 2000) or full year (YYYY). The second segment might be different calendar information or a year's incremental release number. Calendar Versioning is usually utilized by projects that want to establish a consistent cadence and user expectation for future version release times. An example of CalVer is shown in the following Figure 2.13 [37] [150].

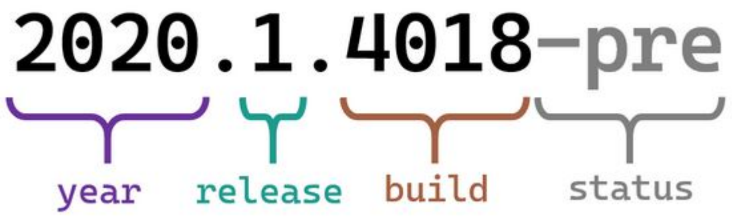


Figure 2.13: Calendar Versioning
Source: `https://nehckl0.medium.com/semver-and-calver-2-popular-software-versioning-schemes-96be80efe36`

2.4 Dependency/Package Management

When one piece of software relies on another to build or execute, it is called a dependency [54]. There is a distinction between components and libraries when discussing dependencies.
Libraries are software packages provided by third parties that are not controlled by an organization's own team other than the choice of which to utilize. Libraries are often updated only infrequently.
Components, on the other hand, are parts of software on which an organization's program depends but which are also built by the same organization. Components are often updated on a regular basis. This distinction is crucial since there are more things to consider when developing a build process when working with components than libraries, such as how to eliminate circular dependencies between components [87].
When talking about dependencies there are two types to distinguish between, direct dependencies and transitive dependencies. Direct dependencies are libraries that are directly accessed by an application, whereas transitive dependencies are libraries that other dependencies are using, they are dependencies of a dependencies as shown in Figure 2.14 [55].

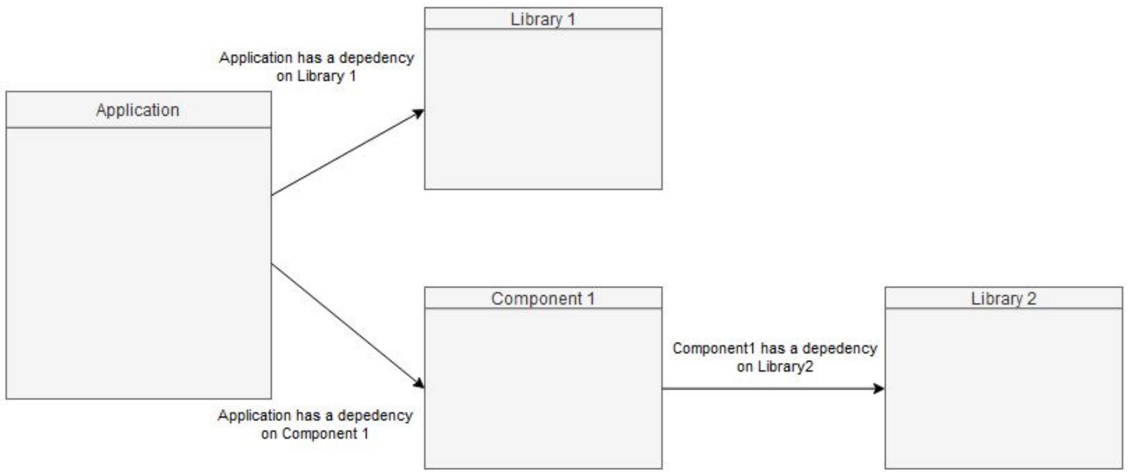

Library 1
Application has a depedency on Library 1
Application
Component 1
Library 2
Component1 has a depedency on Library2
Application has a depedency on Component 1

Figure 2.14: Transitive Dependencies

As organizations grow, managing dependencies manually across several projects quickly becomes tedious and difficult to maintain, which is why proper dependency management

is essential.
Dependency management enables to keep applications stable and reproducible, despite being under constant change, additionally it enables to setup the right dependencies for applications at all times.
A critical feature of dependency management is to consider that the versions of dependencies that an application needs, whether third-party libraries or organization owned components, are likely to be updated and modified, which might break applications or introduce undesired behavior. Which may raise security risks since some of the needed dependencies may get obsolete. In that case it is also more difficult to figure out what is breaking the application or the cause of performance issues, since this issues can occur without the main application changing [54][12]. Because it is crucial for the long-term maintenance of a product, a solid dependency management approach also includes versioning dependencies, including libraries and components. Reproducibility will be unattainable if dependencies do not have versions. That implies that if an application breaks as a result of a dependency update, developers will not be able to track down the modification that caused the problem or discover the last working version of the library [83] [108]. Additionally, when a software project is checked out from version control and an automated build is run, the same binaries will be built no matter who and when it builds, which means that the exact same binaries will be reproduced in the future when running that same version of the software application, hence versioning is indispensable [87].
In software projects, there are two viable options for managing libraries. One option is to put them in version control. That way developers may easily assess whether or not the application is up to date with the newest versions of the libraries by knowing which versions the program is utilizing. The advantage of this technique is that everything clients need to use the application is in the same repository as the source code, once the project is checked out of the repository, everyone is guaranteed to build the same packages [87].
However, there is issues with checking libraries and components into VCSs. Checked-in library repository may get enormous in size over time, making it difficult to identify which of these libraries are still being used by the application, and growing artifacts might be difficult to manage depending on which VCS is used.
Another option is to utilize a dependency management tool to get libraries from online repositories or, the organization's own artifact management server (See 2.5) [87] [3]. As part of a project's setup, a dependency management tool allows declaring exactly which versions of libraries and components are required. The dependency management tools

then downloads the necessary versions of the libraries that the project requires, resolving dependencies on other projects and guaranteeing that the project's lifecycle is free of inconsistencies. Additionally, using an own artifact repository, such as Artifactory (See Artifactory) or Nexus (See Nexus), guarantees that builds are reproducible and furthermore it also makes auditing libraries much easier, by managing which versions of each library are available to projects within an organization [87] [11].

2.5 Artifact Management Server

Packages, containers, binaries and libraries are examples of artifacts produced by software development processes [13]. Furthermore, artifacts may are dependencies (See 2.4) that an application requires to run or deploy, such as open-source software. Working with artifacts can be difficult because they come from a variety of places both inside and outside an organization. Each system with which an organization interacts carries a risk of failure owing to outages or other problems. By centralizing artifacts in a single location, an artifact management system overcomes these challenges of complexity and reliability. Organizations therefore gain greater control over their artifacts and how they are utilized [14][13]. When it comes to dependency management and CI/CD integration for artifacts and dependencies, an artifact management system can serve as a single source of truth. This cuts down on the time and risk associated with downloading dependencies from public repositories. Universal artifact management helps development teams avoid inconsistencies by making it simple to locate the correct version of an artifact [87] [12]. It also provides many other features and advantages such as:

1. Traceability: Versions are kept track of, which comes in handy when standardizing software libraries and auditing third-party licenses [13].
2. Reproducibility: Artifacts and metadata remain stable, ensuring predictable and repeatable builds [13].

Artifactory

JFrog Artifactory [15] is a cloud based repository management tool for software development teams and is used for managing artifacts and binary repositories [96] [15]. Software distribution, encrypted data storage, access control and automatic server backups are all important characteristics. Admins may obtain insight and manage application packages and open-source libraries using the Artifactory platform. To expedite application

delivery, administrators may link the solution with continuous delivery tools such as Ansible [79] and Saltstack [124]. JFrog Artifactory, being a universal Artifact Repository Manager, completely supports software packages written in any language or technology. Artifactory has a REST API that DevOps teams can use to automate releases and minimize downtime. Artifactory provides users with access to metadata related to environment maps and application files. It also supports integration with version control systems such as Git, TFS, and Bitbucket [98] [89]. Artifactory also allows to deploy with CI servers like TeamCity and Jenkins and additionally gives access to other advanced features like, open-source library organization, license control, a REST API, security and access control, monitoring and maintenance of binaries [96] [15] [95].
Key functionalities of Artifactory are:

1. Artifactory is easy to set-up and to configure. There is no need for dedicated hardware to buy [96].
2. Minimal costs to maintain [96]
3. JFrog Artifactory is supports periodic backups of private and protected repositories [96]
4. It also has access control, which allows administrators to manage the access rights of users and teams to repositories [96]

Nexus

Nexus is a binary repository manager, which enables open-source artifacts from public repositories to be cached, and also the hosting of software components produced inside of an organization, thanks to its proxy features. Nexus has functionalities like build promotion, staging, and comprehensive authentication and authorization. Nexus additionally includes meta data management, dispersed team support, and high availability capability [128].
Key functionalities of Nexus are [89]:

1. Universal repository support like Java, Maven, npm, NuGet, PyPI and RubyGems
2. Compatible with IDEs and CI tools like Eclipse, IntelliJ, Visual Studio, Jenkins
3. Role Based access control

2.6 Build Automation

Build automation is a way to handle builds within a CI/CD (See 3) pipeline. When a developer commits code changes to a repository, a continuous integration server detects the commit and starts a build on the CI system, then verifies the changes by running available software tests and sends a report to the developer which made the change or the code administrator [33]. Build automation is the process of creating and building software without the manual involvement of humans. Tasks that were formerly the responsibility of a developer are standardized with build automation, resulting in programmed, repeatable, automated stages for bringing new software forward to its ultimate form [161].
Software development follows a single track, step by step path from architecture through deployment without automation [161]. This, however, results in a time consuming, manual process that delays the lifespan and introduces problems. Automation, on the other hand, allows a company to become more flexible, agile, and responsive to changing business demands. Many software development processes, like unit- and integration testing, are ideal candidates for automation since they are repetitive [63][32].
Automated builds can be triggered in a variety of ways:

1. Manually: Developers start a build of a project manually, without the necessity of a particular event [32].
2. Schedule: A schedule trigger allows defining a time schedule for automatically running builds for a given project [48].
3. VCS: A build is triggered when changes are detected in the version control system
4. Post-process: A build is triggered after a different build is finished [47].

When implemented correctly, build automation can have enormous benefits for organizations, such as:

1. Improved Quality: Build automation to helps companies to move more quickly. That means they are be able to detect and fix errors faster, improving the overall quality of products and avoid faulty builds [63].
2. Faster deliveries: Build automation aids in delivery speed. This is because it minimizes repetitive work and guarantees that faults are discovered quickly, allowing to release faster [33].

3. Increase in productivity: Build automation allows for quick feedback. This means that developers will be more productive, therefore devoting less time on tools and processes and more to delivering value [32].

4. Scalable: Once implemented, automated builds are more scalable than manual processes. As projects grow in size and complexity, build automation can support and enable that growth without requiring large investments in additional resources [161].

Build Automation Workflow

The workflow of build automation is illustrated in Figure 2.15 and can be described as the following:

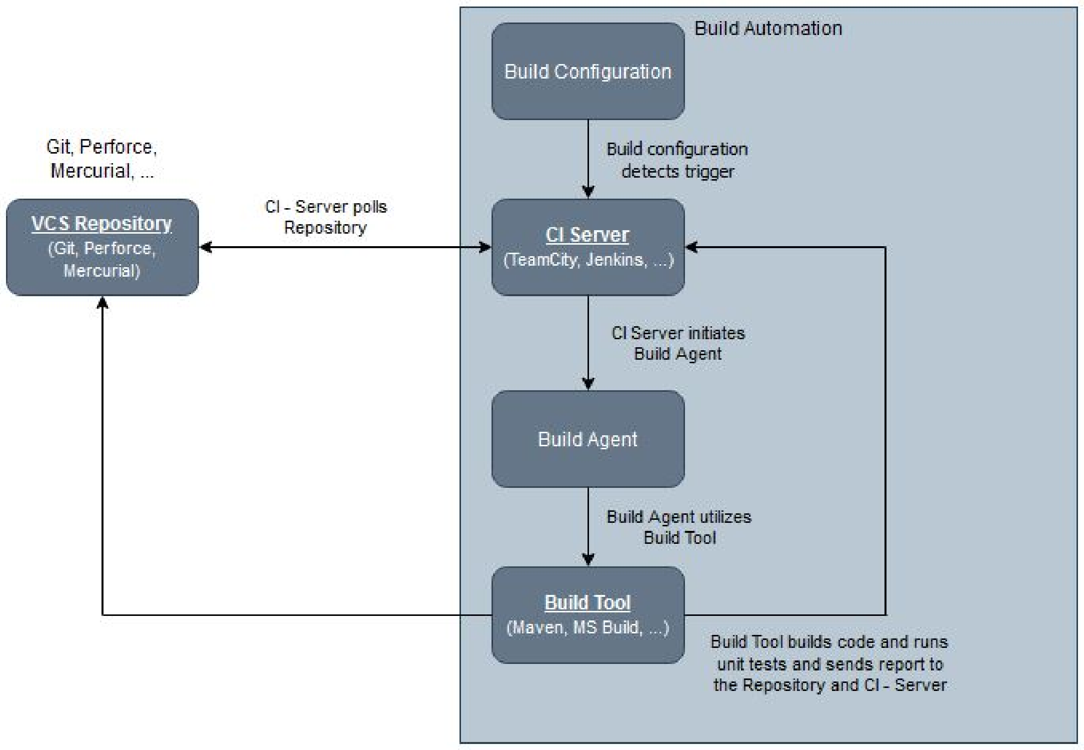
1. Based on the build trigger configuration, a CI server registers that a build has to start for a given VCS repository.
   A build configuration is a set of settings which determines which, how, what and when a build has to be started for a specific project, including the different variety of build triggers which were mentioned above. Build configurations may also include when to run tests and when and where to publish artifacts which resulted from building a software [33][126].
   A continuous integration server is a system that orchestrates a continuous integration pipeline, allowing developers to start builds numerous times per day and build automatically on different target servers outside of their local computers [33].

2. Depending on the build automation tool the CI server polls the VCS repository and forwards it to a build agent or the CI server commands a build agent to poll the repository itself [33].
   The build agent is the software that follows the instructions from the server to complete a build. It is usually installed and configured on a system other than the CI server [126].

3. When the build agent has the needed sources and files it utilizes the according build tool, based on the build configuration, which compiles and runs tests [126][33].
   A build tool is a script, framework, or other piece of software written specifically for a programming language to compile code, run tests on it, and perform other build related tasks, for example MSBuild for C++ or Maven for Java [87][33].

4. After the finished build, be it a successful build or a failed one, a report is sent to the CI server and depending on the outcome of the build artifacts may be send to

the server [126][33].



Build Automation
Build Configuration
Build configuration detects trigger
Git, Perforce, Mercurial, ...
VCS Repository (Git, Perforce, Mercurial)
CI - Server polls Repository
CI Server (TeamCity, Jenkins, ...)
CI Server initiates Build Agent
Build Agent
Build Agent utilizes Build Tool
Build Tool (Maven, MS Build, ...)
Build Tool builds code and runs unit tests and sends report to the Repository and CI - Server

Figure 2.15: Build Automation Workflow

2.6.1 TeamCity

TeamCity is a Continuous Integration and Deployment server that comes with built-in continuous software testing, early build problem reporting, and code quality analysis. TeamCity can be easily deployed and used to improve release management techniques thanks to a straightforward installation process [141][50]. TeamCity is compatible with most IDEs, version control systems and supports Java,.NET, and Ruby development [140]. TeamCity automates the development and testing of software products. It gives immediate feedback on every code change, decreases code integration issues. Builds, changes, and failures can all be saved in TeamCity. Cloud integration, continuous integration, build history, customization, extensibility and access control are some of the features available [50][126]. Furthermore, TeamCity includes a number of build tools and integration middleware that operate natively with build tools like Gradle, Maven, NAnt, MSBuild, and others [33].

TeamCity Architecture

The TeamCity architecture consists of two major components [126][64]:

1. TeamCity Server: TeamCity Server provides the UI for managing build configurations. Build Information and result. It is the component which maintains all the objects settings, keeps track of the status of running builds and manages build queues [82].

2. TeamCity Build Agents: Is the program that carries out a build process in accordance with the directions from the server. It is typically installed and set up on a system other than the TeamCity server. Multiple operating systems, various platforms, and pre-configured environments are all possible for TeamCity build agents [82].

Advantages of TeamCity:

1. Simple integration with popular IDEs [50].

2. Continuous integration and code reuse [141].

3. Support of various version control systems like Perforce, Mercurial, Git and Subversion [49]

4. User and build history management [50]

5. Cross-platform [50]

6. Provision of statistics on build agents and the usage of build machines [140]

Drawbacks of TeamCity:

1. It is not open-source [50].

2. Steep learning curve since there are many possibilities and configurations available [50].

2.6.2 Jenkins

Jenkins is a open-source automation platform designed for Continuous Integration. It is a self contained automation server that can automate a variety of operations connected to software development, testing, and delivery or deployment [74]. It is used to continually test and compile software applications, which makes it effortless for developers to commit changes and to automatically start a new build. It also enables continuous software delivery by interacting with a wide range of testing and deployment tools and has extensibility and configuration simplicity. It is compatible with all main systems and can quickly test and deploy on numerous platforms [140][74]. It is also possible for organizations, which utilize Jenkins to automate and increase the speed of software development processes. Jenkins incorporates many different development life-cycle operations, such as test, build, package and deploy [33].
Advantages of Jenkins:

1. It is open-source [94]
2. Free of cost [168]

Drawbacks of Jenkins:

1. A lot of functionalities are depending on plug-ins [8]
2. A lot of plug-ins have issues with the updating [8].

3 Continuous Integration and Continuous Delivery/ Deployment

The first step toward a modern software engineering infrastructure starts with the introduction of Continuous Integration and Continuous Delivery / Deployment.
Continuous integration (CI) is a method in which programmers make minor modifications and tests to their code on a regular basis. This process is automated due to the size of the requirements and the number of stages involved, ensuring that teams can build, test, and deploy applications in a consistent and repeatable manner. CI streamlines code updates, giving developers more time to make changes and contribute to better products [23] [38].

Continuous delivery and deployment (CD) are the packaging equivalent of continuous integration (CI). CD enables building, configuring, packaging, and deploying software in a matter that it may be delivered and released to production in an automated and consistent process at any time [38][164].

A correctly structured CI/CD pipeline enables businesses to quickly respond to shifting consumer demands and technology advancements. Since CI/CD pipelines are significantly more flexible, approaches like DevOps with CI/CD overcome problems like always changing requirements and needs of clients [164].

Figure 3.1 displays how an iterative CI/CD Workflow looks like. With every code change automated builds and tests are run. Depending on needs, releases are automatically created on every single code change, implemented feature or on a time basis. The releases are then deployed and used by other parties.

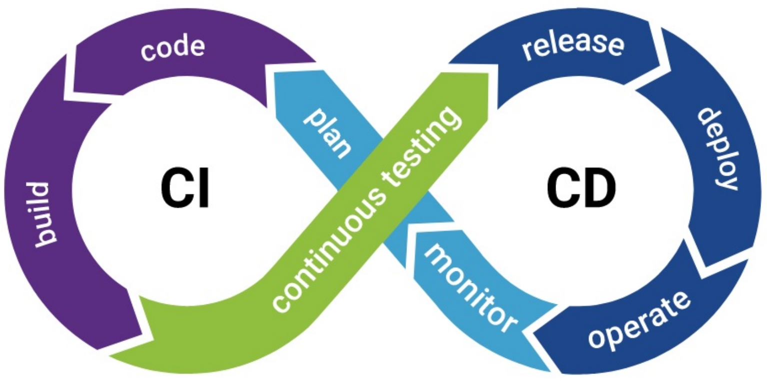


Figure 3.1: CI/CD Workflow
Source: `https://www.synopsys.com/glossary/what-is-cicd.html`

3.1 Continuous Integration (CI)

The majority of software builds by wide-reaching teams spend a substantial amount of time in an inoperative condition. The cause for this is that most of the time no one wants to attempt to run the entire application before it is completed, since it can be time consuming and tedious and no developer is trying to compile the entire software project and utilize it in an environment similar to production [87]. This is especially on point in software projects that use long-living branching strategies or wait until a project is finished to perform quality assurance. Many of these projects arrange extensive integration phases near the end of the software development phase to give the teams enough time to integrate the features and branches and have the software application operating before acceptance and integration testing. Too often, some projects discover that by the time they reach this stage, their software or feature is no longer suited for purpose. These integration phases can take an exceptionally long time, and the durations are not predictable [87][38].

With CI in use, every time someone makes a change, continuous integration demands that the complete application is created and a wide-ranging set of automated tests be run against it. Importantly, whenever these automated builds or tests fail, the development team must quickly stop what they are doing and solve the problem. Continuous integration aims to keep the software in a functioning state at all times [87][18]. Software is broken until it is shown differently, usually during the testing or integration stages, if continuous integration is not used. Software is proved to work with continuous integration due to a robust set of automated tests that run on every new update. Companies that efficiently utilize continuous integration can release software faster and with fewer errors than those who do not. Bugs are identified far earlier in the delivery process, when they are less expensive to fix, saving both money and time [87][38].

3.1.1 Continuous Integration Workflow

Development checks out code from the repository and works on it locally. For the feature to add, a new branch in the version control system will be created. Development conducts tests locally once the feature branch is ready to be merged [60] [145]. The modifications will be committed to the VCS once all tests pass. A CI server checks out changes to the repository and conducts a "build and test" whenever changes occur. The CI system compiles the complete system on the particular branch and performs every available unit and integration tests. Development gets notified of the integration result via the CI server. There are four possible outcomes: failed builds, successful builds, failed tests, and successful tests [145].

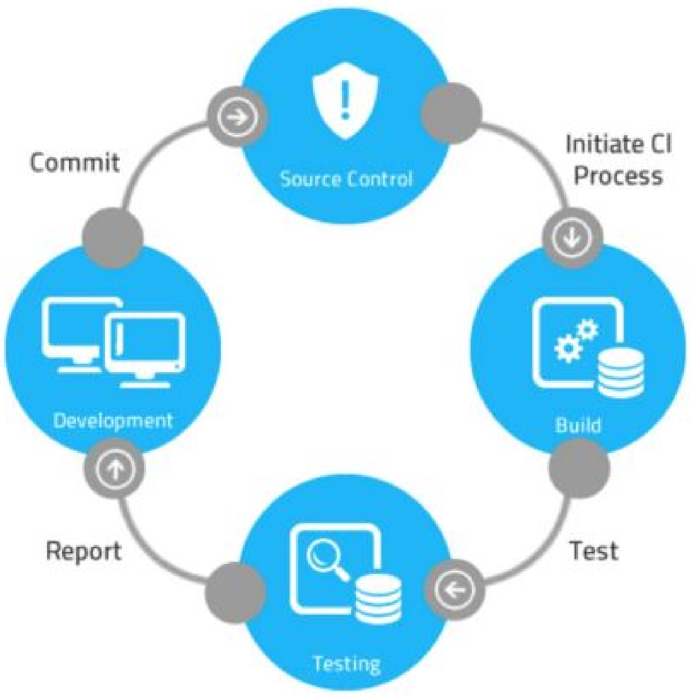


Figure 3.2: CI Workflow
Source:https://speakerdeck.com/jadavchirag/continuous-delivery-with-flutter

3.1.2 CI Practices and Patterns

The goal of a CI system is to ensure that software is working, in essence, all of the time. In order to ensure that this is the case, there are patterns and practices.

Build on every commit

One of Continuous Integrations principal goals is to make application administration easier in order to speed up production and maintenance. The frequency of deployment is one of the most essential parameters in this sort of procedure. This metric is critical for a successful DevOps transition, thus it must be clearly monitored. A CI/CD pipeline gives immediate feedback to developers by generating the solution and executing a set of automated tests each time a change is submitted [51] [52]. The goal is to avoid laying weak foundations and maintain a consistently releasable code base. Every pull request, merge or commit to the version control system should therefore be built in order to catch unwanted errors and bugs. Developers who commit code changes, need to be able to monitor the mainline build so they can fix it if the code changes break the application [87] [51].

VCS Conventions

Defining norms has significant implications for individuals, teams, and automated systems. Even if their importance is sometimes overlooked, they play a major part in the continuous integration chain and may help to organize development [52].

1. Commit messages: Making commit messages more human and machine readable, which can help developers who have to review the commits to understand the code changes [52], e.g. if a commit is fixing a particular problem on a particular method, this may be mentioned in the commit message.

2. Branch names: A convention for branch names in version control system can help other teams and developers to understand what changes are done in a particular branch [52], e.g. if a branch is created for a new feature, the prefix of the branch name may be "feature".

Only Check-In Working Builds and Commits

The primary branch should always be in a stable state. Therefore a restriction in order to prevent developers to commit directly to the primary branch is favourable. Code changes to the primary branch should only be allowed via verified pull requests by other developers [51].

Cleaned-Up Environments

Multiple source codes can be supported by a build automation tool at the same time. Taking the effort to clean up pre-production settings between deployments can help to get the most out of testing processes. It is difficult to keep track of all the configuration changes and upgrades that have been performed to each environment when they have been operating for a long period. Environments diverge from the initial configuration and from one another over time, thus results of software tests may vary from environment to environment and may not have identical outcomes [52][28].

3.1.3 CI Antipatterns

Antipatterns, like their design pattern counterparts, establish an industry vocabulary for prevalent flaws in organizational procedures and implementations. An Antipattern is a type of pattern that indicates a popular solution to a problem that has definitely negative outcomes. The Antipattern could be the result of a developer who did not know any better, did not have the expertise or experience to solve a specific problem, or implemented a perfectly fine pattern in the inappropriate environment [10] [56].

Infrequent Check-Ins

In this anti pattern, the code remains uncommitted and unchecked in for a long time on local laptops. The most common explanation is the enormous amount of changes required to complete the feature. The integration is delayed since check-ins are not frequent. Other developers check-in their code while the feature is being developed. Others will check in more code as this feature takes longer. Because of the large amount of modifications made, this causes several merge conflicts [9].

Spam Notification

Sending little feedback that is either non-actionable or provides no insight into the build failure. Excessive input, including to team members who were not involved in the build. This is eventually classified as spam, causing developers to disregard messages. People become overwhelmed by such notifications and begin to ignore them. This renders the notifications worthless, as one may miss a build failure notification. To address this, the CI server should be set up to not send out notifications for every check-in. It should only be notified when a fresh check-in fails, this is when developers should concentrate on fixing the build [9].

Build Time

Continuous integration is all about getting quick feedback. A long-running build lengthens the time developers must wait after checking in because they must wait for the build to finish before proceeding. This may lead to infrequent check-ins in order to avoid the long wait time. Running several checks and tests as part of the build is one reason for the lengthy build time. To solve this problem, configure the build to only build the code and run fast unit tests [9].

3.2 Continuous Delivery / Deployment

Continuous delivery and deployment (CD) starts where Continuous Integration (CI) stops. CD simplifies the process of getting the new code changes on different environments. This could be, for example, a production server.
Most teams deal with many environments, each of which has its own configuration, such as [87][110]:

1. Development: Where developers build the code
2. Integration: This is where new code and features are combined and validated that it works with existing code
3. Test: This is when the merged code is put through both functional and non-functional tests to ensure it fits the organizations and customers needs.
4. Staging: Staging is used to confirm that the program is ready for usage by testing it with actual data.
5. Production: This is when the program is made accessible to users.

Continuous Delivery and Continuous Deployment ensures that code modifications to these many environments is done in an automatic manner. Continuous delivery starts where continuous integration ends since it sends the software code to a production and testing environment after the build process [18]. This means an automatic release process in addition to automated testing, and deployment of applications at any time. In theory, continuous delivery allows to distribute as often as the business needs it. However, if all the benefits of continuous delivery are wanted, a deployment to production as soon as possible to ensure that tiny batches of code are released that are easier to troubleshoot in the event of a problem, is desirable [104] [86].
According to a peer-reviewed study [59], the use of CD in businesses has an immense positive impact on software delivery performance issues like change fail rates and IT performance [86].

With the correct usage of continuous delivery companies may gain important benefits such as:

1. More reliable releases: The number of code changes in each release reduces as the frequency of releases increases. This makes locating and resolving any issues that

do arise easier, as well as lowering the amount of time they have an effect [86] [104]. It is simple to make zero downtime installations that are unnoticeable to users by using different patterns (See 3.2.1)[86].

2. Lower costs: In the long-term, any successful software or service will change vastly. It lowers the cost of generating and delivering incremental software changes by eliminating many of the fixed expenses involved with the release process by investing in deployment, and environment automation [154] [86].

3. Reduced complexity: The difficulty of software deployment will be reduced[154].

4. Faster development: There is no need to interrupt production for releases[154].

5. Improved Customer Satisfaction: Customers notice a steady stream of improvements, and quality improves on a daily basis rather than monthly, quarterly, or annually [36].

Continuous Delivery/ Deployment Workflow

A CD workflow, also called a CD pipeline, is an implementation of the continuous paradigm in which automated builds, tests, and deploys are organized as a single release procedure. A CD pipeline, to put it another way, is a series of processes that a code changes will go through on their way to production. A CD pipeline automates the delivery of high-quality goods from test to staging to production, according to business requirements [17]. As seen in Figure 3.3, after the testing phase in the CI pipeline is done, additional reviews, in the form of tests for example, are done before publishing to a staging or production environment.

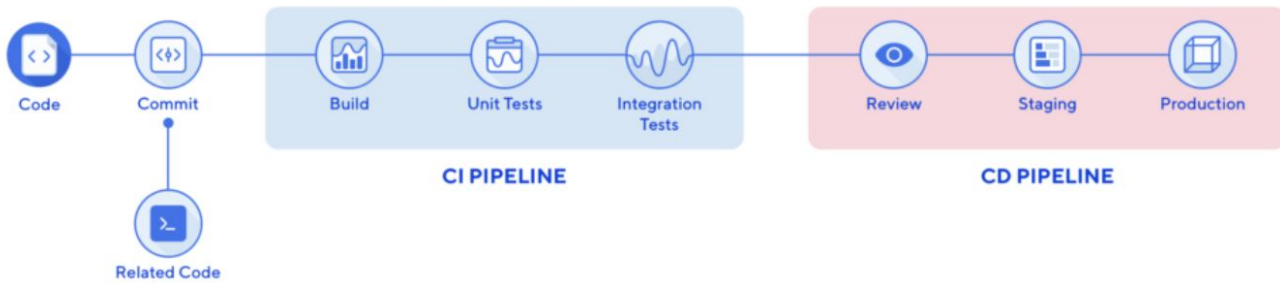


Figure 3.3: CD Workflow
Source: `https://www.elasticweb.nl/kennisbank/continuous-integration-en-continuous-delivery-verder-uitgelegd`

Continuous Delivery vs Continuous Deployment

Continuous delivery is a semi-manual technique in which developers transmit changes to clients by just manually enabling them, whereas continuous deployment focuses on automating the entire process. [165] [18]

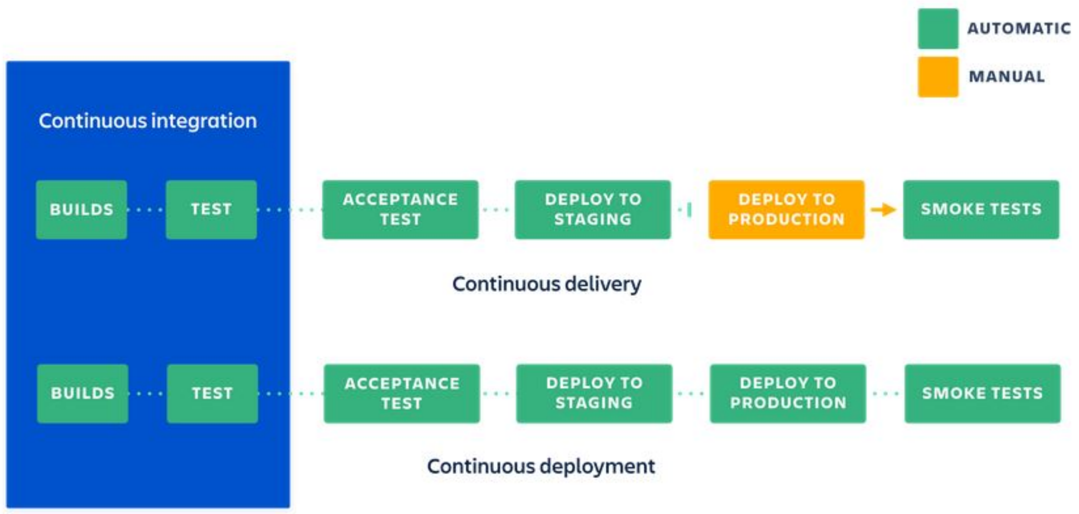


Figure 3.4: CD vs CD
Source: `https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment`

3.2.1 CD Practices and Patterns

This section focuses on some of the important patterns that are prevalent while implementing CD.

Blue-Green Deployments

Getting software from the last level of testing to actual production is one of the issues with automating deployment. It is critical to complete this promptly to avoid downtime. The blue-green deployment strategy accomplishes this by having two production environments that are as similar as feasible. The previous version is referred to as the blue environment, whilst the current version is referred to as the green environment. Once all production traffic has been migrated to green, blue may either be put on standby in case of a rollback, or it can be taken from production and upgraded to become the template for the next update. If something goes wrong, teams may switch the router back to the blue environment using the blue-green deployment method [87][160].

Build Binaries Only Once

The binaries that are put into production should be identical to the ones that passed the acceptance test. If developers rebuild binaries, there is a chance that something may change between the time they are created and when they are released, and the released binary will be different from the tested one. [122].

Always Deploy The Same Way

To guarantee that the build and deployment process is properly tested, utilize the same approach to deploy to every environment, whether the workstation of a developer, a testing environment, or production [122].

3.2.2 CD Antipatterns

This section focuses on some of the important anti-patterns that are prevalent while implementing CD.

Manual Deploying

Most modern apps, regardless of size, are difficult to deploy due to the numerous moving elements. Many companies release software by hand. This means that the actions required to install such an application are viewed as discrete and atomic tasks that are completed by a single person or group. These steps require judgement, making them vulnerable to human mistake. Even if this is not the case, variations in the order and timing of these processes can result in various outcomes. These distinctions are rarely beneficial [87].

Deploying only to a Production-like Environment after Development

In this design, most of the development work is completed before the software is deployed to a production-like environment, for example, staging. It is very common to find that inaccurate assumptions about the production environment have been built into the design of the system when using this method of deployment. New issues are frequently discovered after the application is delivered to staging [87].

3.3 Release Managment

A release is the publication of a software application's final or most recent version. It usually refers to the introduction of a new or improved version of an application [158]. A release makes services and features available to users. Release management guarantees that release teams deliver the apps and upgrades that the business requires while maintaining the integrity of the production environment. Release management is more often than not a business duty rather than a technical one. This is because, from a revenue or portfolio management standpoint, release timing decisions can be linked to business strategy [110]. A business can choose to release features in accordance with a marketing strategy or stagger them to avoid cannibalism of existing goods or to counter competition activity. Features can also be distributed to different clients based on the company's product offerings, such as advanced functionality for premium customers [170]. The effort necessary to oversee, administer, and make that process successful requires a variety of disciplines [101][170]:

1. Planning and scheduling releases
2. Standardizing release processes when possible
3. Defining the roles and duties within the release process
4. Automating the release procedures
5. Reporting on release performance

Release Practices and Patterns

One of the main goals of continuous deployment is to lower the risk of software release [61]. In order to ensure for low risk releases there are the following patterns.

Canary Release

Software canary release is a deployment approach that combines the best features of different deployment methods to create an optimal current strategy. To reduce the risks associated with releasing new software, it uses a step-by-step rollout procedure with excellent monitoring and rollback alternatives. Small subsets of people are used in a canary release, who are exposed to new updates before everyone else. They usually have a limited amount of time to evaluate the rollout and assess whether the software is ready for distribution to a larger number of users or all chevaliers [152][105][87].

Dark Launch

Development teams may utilize dark launching to test the effectiveness of new, production-ready features without exposing them to the full user base. Using feature flags or feature toggles, which activate or deactivate functionality without publishing code, is the easiest approach to dark launch [137]. Feature flags are if-then logic statements in a code that separate code deployments from feature releases. The development team may make a feature visible to a limited set of users once it has been delivered to production, and then scale up as needed. During the dark launch, software developers may monitor user feedback to determine whether the product should be sent out to a larger audience. They are also keeping an eye on how the new features effect system performance. If the feature fails to gain traction, users provide unfavorable feedback, or system difficulties appear, the feature flag may be removed with a single click, without restarting the program, while the function is refined. Software teams do not have to rely on potentially dangerous, all-or-nothing feature releases with dark launches [137][29].

Deployment vs Release: The key difference

The difference between release and deployment is, that deployment does not always mean that clients and users have access to the features and functionalities, but only in the production environment as mentioned in 3.2, whereas releasing always indicates the publishing to users [110].

4 Software Testing

Testing is the process of examining a software product to evaluate whether or not it meets the required standards. The testing process comprises assessing the characteristics of the software product for missing requirements, errors , or flaws, performance, and reliability. It entails the use of tools to analyze one or more attributes of interest while running software/system components. Testing enables to find and fix mistakes and flaws in the software before the software product is released and delivered. A wide test coverage for a software solution offers reliability, security and great performance, which saves money, time, and increases client satisfaction. Software flaws may be costly, thus testing is essential [27] [76] [172].

Benefits of software testing:

1. Cost-Effectiveness: In the long run, having tests for software projects can save money. There are numerous stages to software development, and catching defects early on saves a lot of money in the long run [88] [114] [4].
2. Increase in quality: Testing and quality are inextricably linked. The number of faults discovered during testing can be used to evaluate quality, and those problems will be corrected as part of the software development lifecycle. This will be done throughout the development period, and by testing frequently, the software's quality can be enhanced [114].
3. Reduce risks: Defects must be found and either rectified or excluded from the final product to ensure that the system operates as intended during live operation. Because the impact and severity of a serious fault discovered in a live environment are both high due to the end user's involvement, testing is required to reduce risks and ensure that the program is ready for live operation [114].
4. Customer satisfaction: Clients expect system stability. Testing during the development lifecycle provides insight into how well and reliably the application was created, boosting customer trust once it is deployed in a live environment. [88] [114].

When it comes to software testing it can be classified into two categories: functional testing and non-functional testing.

Functional Testing

Functional testing is the testing of the functional features of a software application. This type of software testing compares the system to its functional specifications. Functional testing guarantees that the application meets all of the criteria or standards. This form of testing is focused on the end product of the processing. It does not make any structural assumptions and instead concentrates on mimicking actual system utilization. It is a sort of testing that verifies each software program function operates in compliance with the specifications and technical requirements. Each software application's functionality is tested by handing appropriate test input and comparing the actual result to the anticipated result [57][73][143].
Following are several types of functional tests:

1. Unit tests: Unit testing is a sort of software testing that examines the corrections of a single unit or component. Unit testing is often performed by the developer during the application development phase. A method, function, procedure, or object can be considered a unit in unit testing. Programming skills is required to execute this type of testing [5].

2. Integration tests: Integration tests are critical if an application communicates with a number of external systems using a range of protocols, or if an application is made up of a succession of loosely linked modules with complicated interactions between them [87]. Integration testing is a sort of software testing that testing multiple modules of an application as a whole. This sort of testing focuses on finding defects in the interface, communication, and data flow between modules. These functional tests ensure that all of the software test's various components are functioning properly as a whole [5][142].

3. End-to-end tests: End-to-end testing requires placing a complete application system through its paces in a situation that closely approaches actual use, such as utilizing network communications, connecting to a database or interfacing with other hardware or systems.[143][153].

4. Regression tests: Regression testing involves testing the application's unmodified features to ensure that changing, deleting, or adding new features won't have an adverse effect on other project components that have not changed. [142].

5. Smoke tests: Smoke testing is used to ensure that the system under test's basic and important functions are operating properly at a high level. When the development team provides a new build, the Software Testing team evaluates it and guarantees that no severe issues exist. Before beginning a more thorough level of testing, the testing team will ensure that the build is stable. [143].

Non-Functional Testing

Non-functional testing is a sort of testing that examines a software application's non-functional features such as usability, dependability and performance. It is specifically designed to assess a system's readiness using non-functional factors that are never addressed during functional testing.
Following are some examples of non-functional testing types:

1. Load tests: Load testing allows engineers to better understand how a system behaves when subjected to a given load. The load testing procedure involves simulating the expected number of concurrent users and transactions over a period of time in order to verify predicted response times and identify bottlenecks. This type of test allows developers to figure out how many users an app or system can manage before it goes live [139].

2. Scalability tests: A non-functional testing technique called scalability testing measures how well a system or network performs when the volume of user requests rises or falls. Scalability testing ensures that the system can manage anticipated increases in user traffic, data volume, transaction counts frequency, and so on. It assesses the system's ability to fulfill escalating demands [171][75].

3. Stress tests: Stress testing exposes a system to higher than expected traffic volumes, allowing developers to assess how well it performs beyond its capacity limitations. Software teams can use stress tests to determine the scalability of a workload. Stress tests put a demand on hardware resources in order to figure out when an application can break due to resource utilization. CPUs, RAM, and hard disks are possible resources [139] [169].

Automated Software Testing

When it comes to software testing there are two kinds of testing: manual testing and automated testing [132].

Without the use of test scripts, manual testing is the process of running tests by hand. Manual tests are routinely performed during development cycles for source code modifications as well as for additional scenarios like various operating systems and hardware configurations.

In automated testing, test automation frameworks like Selenium, Cypress and Robotium, are used to automate the execution of tests. Automated testing can carry out pre-recorded and scheduled tasks, evaluate results against expected behavior, and inform a test engineer whether manual tests were successful or unsuccessful. Once automated tests are created, they may easily be extended and repeated. [153][174].
Automated tests can be conducted multiple times throughout the day. This approach is in line with continuous testing, continuous integration, and continuous delivery (CD) techniques to software development, all of which aim to automatically deploy code changes to production [159].

Benefits of Automated Testing

1. Maintaining a constant cost of testing: As new features are added, the software gets more complicated, and as the program grows more advanced, adding new features becomes increasingly difficult. This is especially true when there is a push to provide new versions quickly while not investing enough effort in planning and improving code quality. This eventually slows the delivery of new features [27][133]. The majority complexity of a software is due to features being added hastily without good design, a lack of communication within the team, or a lack of expertise, either about the underlying technology or about the business demands. As a result of when a software becomes more complicated, its maintenance cost rises since it takes more time to test everything, as well as more time to patch and retest the flaws that are discovered. Accidental complexity, in particular, makes software more vulnerable and difficult to maintain. [27][174]. Automated software testing can help with reducing the risks of neglecting testing particular section of a software, since with automated software tests it can be configured to run tests on every change, without the need of an engineer triggering it manually [27].

2. Reduced Business Expenses: To assure quality, software tests must be done frequently during development cycles. Software tests should be run on every source code change and additionally every software release needs to be tested on any of the

available operating systems and hardware configurations. Manually repeating such tests is time-consuming and expensive. Automated tests can be run repetitively at no additional expense after they have been generated. The time it takes to execute repeated tests can be reduced immense using automated software testing [159][174].

3. Better reporting: Automated tests repeat the same processes every time they are run and keep thorough records of the outcomes. The results can then be compared to other reports to see how the software performs in comparison to expectations and needs [174].

4. Increases test coverage: Automated software testing may assist to enhance software quality by increasing the detail and range of tests. Unsupervised automated testing can be utilized to execute lengthy tests that are often ignored during manual testing. They can also be built in a variety of configurations. To asses if a software product is operating as desired, automated software testing can glance inside an application and inspect memory, file contents, data tables and internal program states. With each test run, automated tests are able to execute many complicated test cases, offering extensive coverage [133][174][81].

Scheduling Automated Tests

If an application's entire testing process is short and developers can wait, and the tests are stable, it is worth retaining the testing phase as a single consolidated process and executing it in CI mode, which means the build is started automatically upon each check-in or commit. However, if it takes longer, the test data sets are too big or if the tests are unstable, the development team's productivity will suffer. It is typical to split the construction process in these situations [87][27].

Nightly Builds

A nightly build takes place every night when no one is working and no commits made to the source code [87]. Nightly builds are triggered automatically every night and all source code that has been committed into source control during the day is compiled. In larger software projects, a complete reassembling of the whole project and the running of tests with all test data may take too long for an individual to do as part of a work day, therefore nightly builds can be introduced. Nightly builds, which are executed periodically regardless of whether any source code has changed, also make sure that the build tools have not broken as a result of system updates [27][135][117]. This method necessitates that someone from the development team investigates the results every morning, fixing the tests as needed and reporting any relevant bugs to the developers [27][135].

Running Tests as Part of the CI

The tests must be executed as part of the CI build itself if companies are to truly benefit from test automation and the quick feedback loop it can provide. This can only happen if the tests are very reliable and quick, otherwise developers will not be able to check in valid modifications or they will just ignore them and the system would be left ineffective [27]. An example would be to build the whole application including every test on the creation of a release.

Continuous Testing

Continuous Integration and Continuous Delivery are focused on how applications are developed and brought to customers which are certainly important aspects for a software development life cycle. Instead of being a benefit, the increased frequency and pace of Continuous Integration and Continuous Delivery may become a liability if the automated delivery process is unable to immediately assess how changes will affect business risk or negatively affect end-user experience. [167]. This is where the Continuous Testing approach comes into play. Continuous Testing (CT) is a software development process that involves automatically testing applications continually throughout the software development life cycle (SDLC) and whenever changes come in. CT's purpose is to assess software quality across the SDLC, providing vital feedback earlier and allowing for higher quality, faster delivery [166].
When done correctly, it conducts automated tests as part of the software delivery pipeline to offer feedback as quickly as feasible. It promotes testing at all stages of the SDLC, from development, e.g. on every code change, to deployment, e.g. every time a new release is created, which helps to enhance the CI/CD process. Continuous testing makes certain that development is error-free and that only stable software is released. At a higher level, it eliminates the stumbling barriers that come with conducting testing in a single phase like in the Waterfall model. As soon as code is integrated with continuous testing, it is automatically tested. This immediately supports CI/CD and the goal of producing high-quality software in a shorter amount of time [166][167].
CT also saves time and effort for developers because they no longer have to wait for software testing teams to finish testing before they can fix their code. Instead, testing is done on a continuous basis, allowing for proactive changes to code quality and secu-

rity issues in real time. It has a broader benefit in that it minimizes risk. Instead of being reviewed or checked once at a certain phase of the SDLC, software is reviewed or checked many more times and in many more ways with CT. This gives more visibility into weaknesses and more chances to find them [166].

5 Implementation

In this chapter, as a proof-of-concept a through strategy will be presented in order to shift from legacy systems and applications to a postmodern automated software infrastructure, while using methods and technologies described in the previous chapters of this thesis.
The suggested concept should show software engineering practitioners, which do not make use of any modern software development practices, a possibility to upgrade their infrastructure to use Continuous Integration and Deployment, automated building processes, continuous testing with automated software tests and techniques for software developer teams to make use of best practices.

5.1 The Legacy System's Current State

The considered legacy system consists of two C++ applications, Application-A and Application-B. As seen in Figure 5.1, Application-A has dependencies to two open-source libraries and Application-B is depending on the binaries from Application-A. The open-source libraries are not hosted by the company and have to be downloaded locally and the Visual Studio[151] project solutions have to be configured manually to be able to compile. Both applications and the test data are not under any version control and not part of any automated infrastructure, in order to build them every component has to be downloaded on the developers' local machines and configured manually. Because of that, whenever developers want to merge their changes together, they have to do it hand-operated and test the applications manually.
From the current state of the application and the non-existing infrastructure it can be assumed that introducing changes to the software applications is cumbersome, needs a lot of effort and communications between the developers. Because of lacking versioning of source code and test data, changes need to be implemented especially carefully and since there is also no automation testing processes, developers need to manually execute and test every single part of the code in order to be sure that changes do not break the

systems. Ultimately causing difficulties in assured releases to customers.

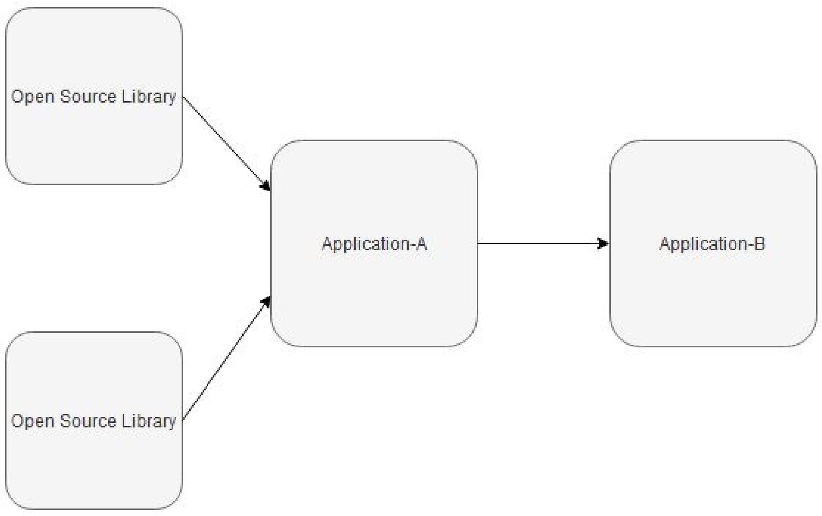


Figure 5.1: Legacy System Overview

5.2 Desired State

In order to modernize the software development process and implement an automated infrastructure for building and testing, it is needed to introduce a version control system, an automation build tool, software engineering best practices like versioning- and branching strategies and a binary repository management system. Additionally, it will allow to version and audit every developed application and every created and used library or binary.
The modernized system will include:

1. Git (See 2.1.2) as VCS to version control the source code and configure files.
2. TeamCity (See 2.6.1) as an automation build tool in order to build and test code changes checked in to the VCS.
3. JFrog Artifactory (See 2.5) as a binary repository management system for binaries and test data.
4. Semantic Versioning (See 2.3.1), as a unified versioning strategy for developers, in order to consistently version software releases and test data for reproducibility.

5. GitFlow (See 2.2.5) as a branching strategy, so that all developers and teams from an organization, create unified branching names in the VCS and therefore understand what other developers and teams are doing.

6. Automated testing strategies to increase test coverage, to ensure that only working code can be checked in to the VCS and to be able to release always stable applications.

As illustrated in Figure 5.2, the desired state of the whole infrastructure will enable an agile workflow for software developers. It will allow developers commit changes to the VCS without the fear of breaking anything. Whenever it is wished for, the build automation tool will be able to build and monitor the code base, to ensure stable and compiling code. The results of automated tests will be reported to the developers responsible of the according repository of the build. The automation build tool and developers will have access to Artifactory in order to download needed binaries and test data, and also update and upload these data back on.

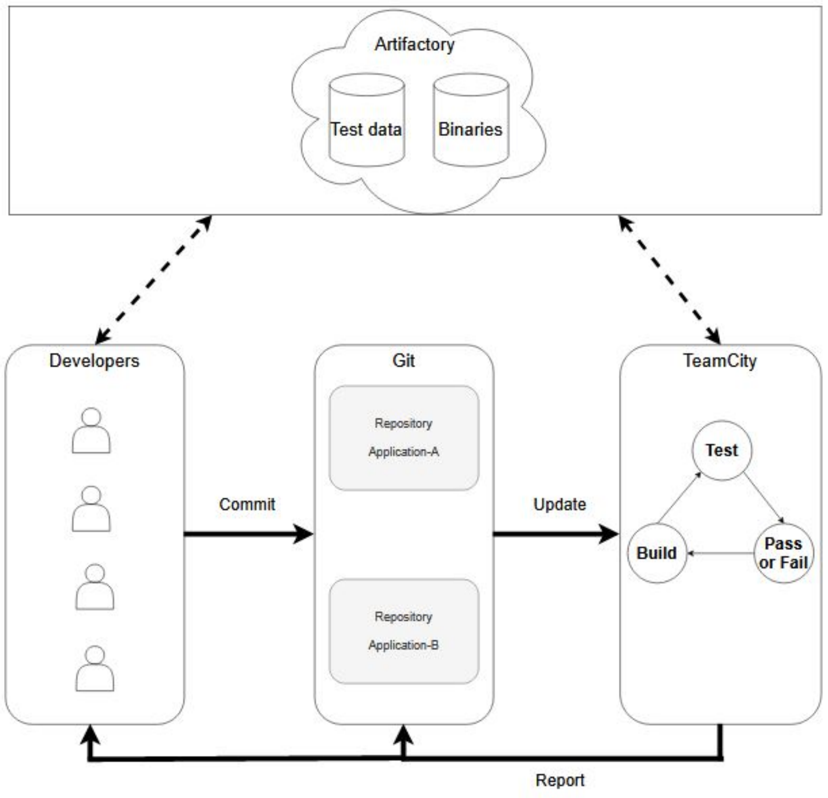


Figure 5.2: Desired Workflow

5.3 Git as Version Control System

As mentioned above the strategy in this thesis suggests Git as the version control system for the source code of the applications.

Why Git?

Considering all the advantages and disadvantages of the mentioned version control systems in section 2.1, this thesis suggests the use of Git. Taking into account that Git is one of the most popular VCS [149][25] and the out of the box integrability with TeamCity without the need of any additional configuration [67], Git is an reasonable choice. Additionally, Data Version Control (DVC) (See Data Version Control), the suggested tool to version binaries and test data, is based on Git and Git repositories which makes integrating it simple [53].

GitHub

There are multiple platforms to host git repositories, such as GitLab, GitHub and Bit-Bucket. The implementation for this thesis will use GitHub, since it is the most popular one among all the other platforms [129][71].

5.3.1 Re-platforming to Git

In order to use Git as VCS it is needed to create an online repository on GitHub and to install Git on a local machine which has the source code of the application.
As shown in Figure 5.3, creating repositories in GitHub is done by pressing the "New" button.

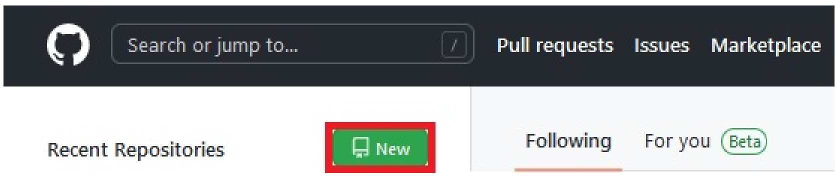


Figure 5.3: Creating a repository in GitHub

After creating the online repository, it is then possible to connect the local source code with the online repository and to upload the source code to that online repository. Firstly, it is needed to navigate via a command-line tool like Cmd, Powershell or Git Bash to the root folder of the source code. Then the following Git commands needed to be executed step by step [66][24]:

1. `git init` : This command creates an empty local Git repository and adds a ".git" directory.
2. `git add .` : The git add command adds a modification to the staging area from the working directory. It informs Git to include changes to a specific file or all changed files indicated by the period, in the command. However, `git add` has little effect on the repository, changes are not truly recorded until `git commit` is performed.
3. `git commit -m "<Commit Message>"`: The git commit command saves a snapshot of the current staged changes, done with the previous `add` command, in the project.

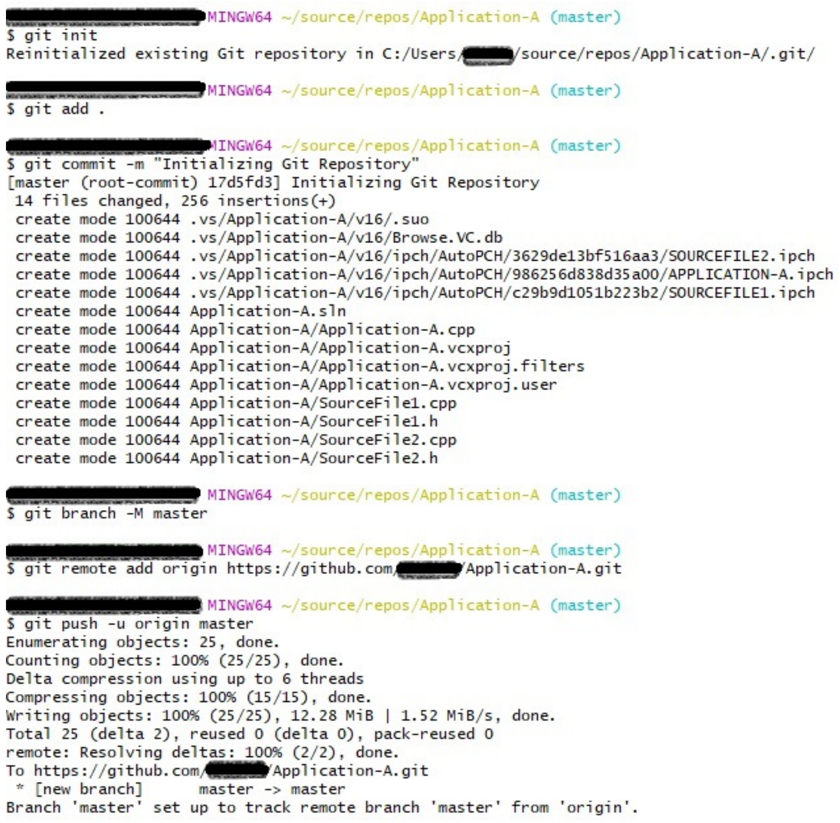
Git Snapshots are committed to the local repository. With the `-m` option, a message can be added to the commit, to describe what the changes and the current commit is about.

4. `git branch -M master`: It creates, or deletes branches. With the `-M` option the command will rename the current branch.

5. `git remote add origin <URL of the Repository>`: This command links the local project folder to an online repository.

6. `git push -u origin master`: The `git push` command is applied to transfer local repository content to the set remote repository. Pushing transfers commits from a local repository to the a linked remote repository. This allows other team members to view a collection of stored modifications.

All these command sequence looks like as shown in Figure 5.4. After that the software project is under version control and everyone who wants to contribute to that project can download the repository with `git clone <URL of the Repository>`

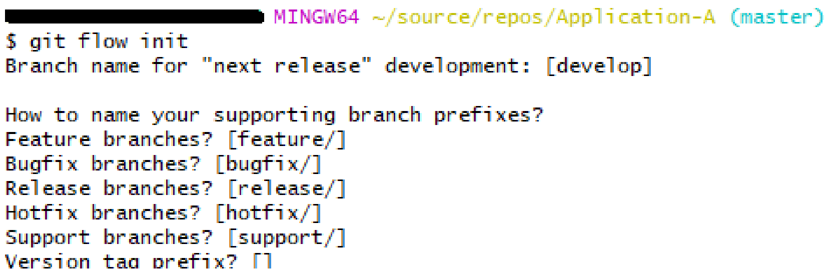


```
[redacted]MINGW64 ~/source/repos/Application-A (master)
$ git init
Reinitialized existing Git repository in C:/Users/[redacted]/source/repos/Application-A/.git/

[redacted]MINGW64 ~/source/repos/Application-A (master)
$ git add .

[redacted]INGW64 ~/source/repos/Application-A (master)
$ git commit -m "Initializing Git Repository"
[master (root-commit) 17d5fd3] Initializing Git Repository
 14 files changed, 256 insertions(+)
 create mode 100644 .vs/Application-A/v16/.suo
 create mode 100644 .vs/Application-A/v16/Browse.VC.db
 create mode 100644 .vs/Application-A/v16/ipch/AutoPCH/3629de13bf516aa3/SOURCEFILE2.ipch
 create mode 100644 .vs/Application-A/v16/ipch/AutoPCH/986256d838d35a00/APPLICATION-A.ipch
 create mode 100644 .vs/Application-A/v16/ipch/AutoPCH/c29b9d1051b223b2/SOURCEFILE1.ipch
 create mode 100644 Application-A.sln
 create mode 100644 Application-A/Application-A.cpp
 create mode 100644 Application-A/Application-A.vcxproj
 create mode 100644 Application-A/Application-A.vcxproj.filters
 create mode 100644 Application-A/Application-A.vcxproj.user
 create mode 100644 Application-A/SourceFile1.cpp
 create mode 100644 Application-A/SourceFile1.h
 create mode 100644 Application-A/SourceFile2.cpp
 create mode 100644 Application-A/SourceFile2.h

[redacted] MINGW64 ~/source/repos/Application-A (master)
$ git branch -M master

[redacted] MINGW64 ~/source/repos/Application-A (master)
$ git remote add origin https://github.com/[redacted]/Application-A.git

[redacted] MINGW64 ~/source/repos/Application-A (master)
$ git push -u origin master
Enumerating objects: 25, done.
Counting objects: 100% (25/25), done.
Delta compression using up to 6 threads
Compressing objects: 100% (15/15), done.
Writing objects: 100% (25/25), 12.28 MiB | 1.52 MiB/s, done.
Total 25 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), done.
To https://github.com/[redacted]/Application-A.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

Figure 5.4: VCS setup of Application-A

5.3.2 GitFlow as Branching Strategy

As it was mentioned in section 2.2 Branching strategies are concerned with how branches are used, formed, and named during the development process. While working with a version control system for generating and maintaining code, a software development team uses a branching technique to structure and keep the VCS clean.
The strategy in this thesis suggests the utilizing of GitFlow, since it includes complete rules with definite responsibilities of branches, suggests very thorough and detailed version control and is very versatile if any adjustments for business needs are wanted as mentioned in 2.2.5.
To apply git flow on an existing repository the command `git flow init` may be applied [20]. This initializes the git flow tool on a computer in order to define how the different branches of the Gitflow (See 2.2.5) have to be named when creating them. The output of this command looks like as shown in Figure 5.5.



```
[redacted] MINGW64 ~/source/repos/Application-A (master)
$ git flow init
Branch name for "next release" development: [develop]

How to name your supporting branch prefixes?
Feature branches? [feature/]
Bugfix branches? [bugfix/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? []
```

Figure 5.5: Initialising Gitflow for Application-A

This command needs to be executed by every developers which wants to utilize the git flow tool on their local machine.
After the initialization it is then possible to use this tool to create branches with appropriate names. If e.g. a release branch is wanted to be created then the following command may be executed: `git flow release start <Version Number>`. Since with the initializing command it was configured what the prefix of a release branch is, this command creates a branch with the name `release/1.0.0` if the given version number to the command is `1.0.0`.

Branch Protection

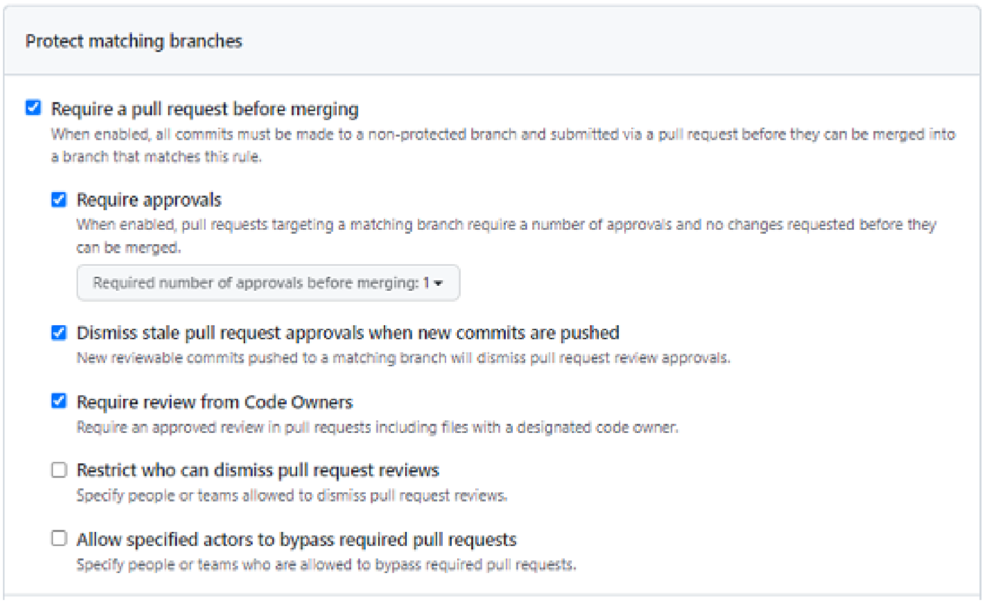
Git branch protection rules are an effective configuration choice that gives repository administrators control over the application of security restrictions. This helps guard against unauthorized users or user groups of deleting or committing code unexpectedly on the git branches. Additionally, the code that was committed might have passwords, API keys, or other programming secrets hardcoded in. These code secrets might wind up in repositories, despite typically being placed there for testing and then forgotten. It thus has the potential to result in a data leak. Also, developers who have write access may push incompatible changes, delete significant branches, or erase the commit history, which needs to be prevented. Additionally, code that is pushed could be untested, contain coding secrets, and ignite a security nightmare [84].
Therefore, it is suggested to use branch protection rules to safeguard the branches to prevent all of this. In GitHub this can be achieved by pressing `Settings` on a repository and then navigating to `Branches` tab and then specify rules for specific branches such as the primary branch `master` (See [84]). As shown in Figure5.6 it is then possible to have different kind of configurations, such as:

1. All commits must be made to a non-protected branch and submitted via a pull

request before they can be merged into a branch that matches this rule and that no direct commits to that protected branch are possible.

2. Pull requests must be approved by atleast a required amount of people, in order to be able to merge the pull request

3. Pull requests must be approved by code owners (See below).


Protect matching branches

☑ Require a pull request before merging
When enabled, all commits must be made to a non-protected branch and submitted via a pull request before they can be merged into a branch that matches this rule.

☑ Require approvals
When enabled, pull requests targeting a matching branch require a number of approvals and no changes requested before they can be merged.
Required number of approvals before merging: 1

☑ Dismiss stale pull request approvals when new commits are pushed
New reviewable commits pushed to a matching branch will dismiss pull request review approvals.

☑ Require review from Code Owners
Require an approved review in pull requests including files with a designated code owner.

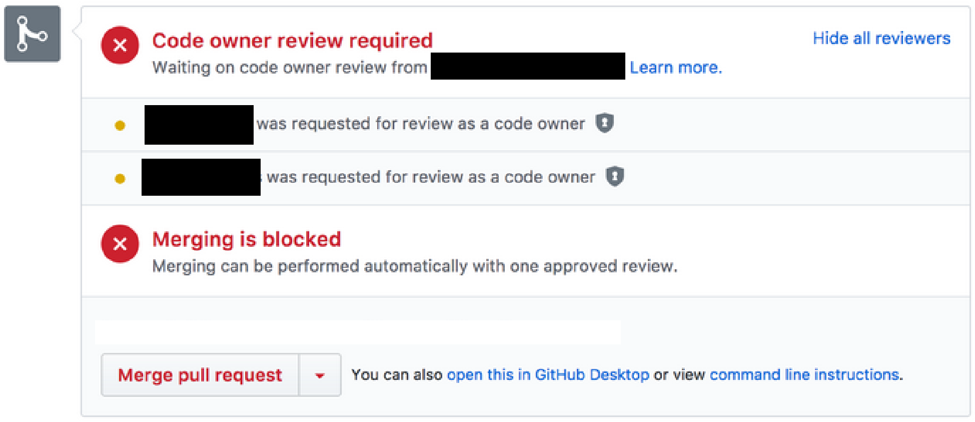
☐ Restrict who can dismiss pull request reviews
Specify people or teams allowed to dismiss pull request reviews.

☐ Allow specified actors to bypass required pull requests
Specify people or teams who are allowed to bypass required pull requests.

Figure 5.6: Branch Protection

Code Owners

While thorough code review is critical to the success of any project, it is not always evident who should review changes. Using code owners, repository administrators may specify which individuals and teams must evaluate projects. When a pull request modifies any owned files, this functionality automatically asks reviews from the code owners. Additionally, when protected branches are enabled, a code owner must leave a review for each owned file before anybody may merge a pull request to that branch. To specify code owners, create a file named `CODEOWNERS` in the repository's root directory (See [7]). Once this is implemented and a pull request is opened GitHub demands a review of code owners, as seen in Figure 5.7.


Code owner review required

Waiting on code owner review from [redacted] Learn more.

Hide all reviewers

[redacted] was requested for review as a code owner

[redacted] was requested for review as a code owner

Merging is blocked

Merging can be performed automatically with one approved review.

Merge pull request

You can also open this in GitHub Desktop or view command line instructions.

Figure 5.7: Pull Requests with Code Owners

5.4 Artifactory as Artifact Management Server

With GitHub source and configuration files of software applications are under version control, yet often this is not everything what projects need in order to compile, build and run. As stated in section 2.5, software applications often have a dependency to other binaries, open-source software, test data or libraries. Since these artifacts are updated irregularly and are consumed by not one single software application, but many others, it is recommended to have these data versioned in a different place as source code in order to avoid downloading arbitrarily large files and directories only to get access to source code [87] and this different place may be an artifact management server.
For this purpose this thesis recommends the use of Artifactory (See 2.5), because Artifactory supports the C++ package manager Conan (See 5.5) out of the box [42], which is useful for the software applications in this thesis and the installation is straightforward (See [93]) and since Artifactory supports wast amount of other binary and package types [15].

Data Version Control

With Artifactory it is clear where to upload and download binaries and test data, but not how to version the data. Certainly it is possible to do it manually with the web user-interface of Artifactory [16], yet if it is wanted to process, track, save and switch between versions of data files or datasets the same way as code within Git, a tool which

can be executed as part of automation scripts is preferable.
For this reason it is suggested to utilize Data Version Control(DVC)[53]. DVC is compatible with any common Git server or provider and runs on top of any Git repository (GitHub, GitLab, etc). DVC supports lock-free local branching, versioning, and all the other benefits of a distributed version control system [53].
To setup everything properly the following steps have to be applied, after the installation of DVC (See [90]):

1. It is needed to initialize DVC inside a Git repository, which needs access to data from Artifactory. The initialization is done with the command `dvc init`. This creates a `.dvc` folder and internal files.

2. Next it is necessary to tell dvc where the needed test data are with the command `dvc remote add artifactory <URL to test data> -default`. This alternates the `config` file inside the `.dvc` folder with the url of the remote artifact management server.

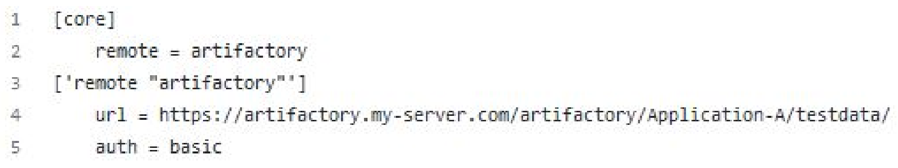
3. Supplementary, if authentication is needed, so that dvc can access Artifactory, it is necessary to add the line `auth = basic` under the `remote` tag, which indicates that whenever access is requested by DVC, username and password needs to be entered for Artifactory.

4. Following that the `config` file has to look like Figure 5.8.

5. To utilize all these commands for everyone who accesses the Git repository, all these changes need to be pushed on to the Git repository with `git push`.

6. To setup the computers of new developers to use DVC, they need to execute the following two commands once: `dvc remote modify -local artifactory user <USERNAME>` and `dvc remote modify -local artifactory user <PASSWORD>`. These commands set the credentials for Artifactory and the `-local` option is to makes sure the credentials are stored locally and not commited to the repository.

7. After that, whenever it is wanted to download the test data, the command `dvc pull` will be sufficient.

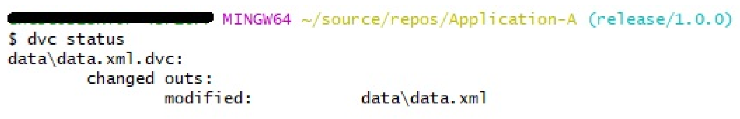


```
[core]
    remote = artifactory
['remote "artifactory"']
    url = https://artifactory.my-server.com/artifactory/Application-A/testdata/
    auth = basic
```

Figure 5.8: DVC Remote Settings

As mentioned above DVC is build on top of Git. That is why the versioning of data is the same as the versioning of source code with branches. Whenever test data needs to be versioned for a specific branch or release, it is done parallel to the source code, which will be explained with the following example. The test data of Application-A consists of many xml files. These files may change from release to release. In order to keep track of the states of the test data for each release, it is necessary to version them, which will be done parallel to the long living release branches within GitFlow. In this example, the following steps will version the test data `data.xml` inside the data folder, for the release version 1.0.0 of Application-A:

1. With `dvc status` it is displayed that the data.xml file has been changed for the `release/1.0.0` branch as shown in Figure5.9.
2. The changed test data file needs to be uploaded to the DVC repository in Artifactory with `dvc add data\data.xml` followed by `dvc push`.
3. The command `dvc add data\data.xml` in the previous step also modified the `data\data.xml.dvc` file, which needs to be pushed to the Git repository on the release/1.0.0 branch with `git push`.
4. With these steps the current state of the test data is associated with the according release and whenever the state of the source code needs to be run with the according test data, developers only have to checkout that particular branch and use the `dvc pull` command.



```
[redacted] MINGW64 ~/source/repos/Application-A (release/1.0.0)
$ dvc status
data\data.xml.dvc:
        changed outs:
                modified:           data\data.xml
```

Figure 5.9: Application-A DVC Status

5.5 Conan as Package Manager

In the beginning of this chapter it was stated that in the current state of the applications, dependencies like C++ open-source libraries needed to be download manually and furthermore that the projects configuration and include directories needed to be manually configured in order to use these libraries. These cumbersome processes can be passed over to a package manager tool.
A package manager maintains track of what software is installed on a computer and makes it simple to install new software, update software to newer versions, or delete previously installed software and offers capabilities to publish own packages [157][119]. As for this thesis, Conan (See [34]), will be the package manager of choice, since it is the preferred C++ package manager in Artifactory and supports Conan repositories out of the box without any additional configuration needed [97].

Managing Build Dependencies

When a Conan repository is created on Artifactory and the needed open-source libraries are uploaded as conan packages (See [42]) it is then possible to utilize it as a single source of truth for all the needed dependencies development wide. As mentioned in section 2.5 this cuts down on the time and risk associated with downloading dependencies from public repositories. A universal artifact and depedency location helps development teams avoid inconsistencies by making it simple to locate the correct version of an artifact and dependency [87] [12].
After the installation of Conan is done (See [91]), it is then possible to utilize Conan for software projects. The following steps will describe how to set up Conan package dependencies for a Visual Studio C++ software application in the Git repository:

1. By default conan searches for packages on `conan.io`, therefore it is needed to change the remote settings to the own conan repository in Artifactory. This is done by the command `conan remote add remote-name remote-url` (See [41]). This configuration is local, which means that it needs to be done on every computer which wants to utilize conan for this purpose.

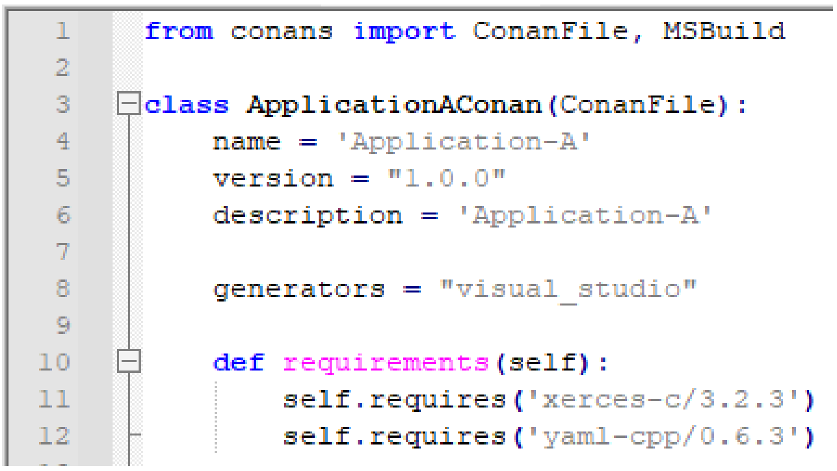
2. Then it is necessary to create a `conanfile.py` file (See [43]) in the root folder of the project. This file is an instruction for installing and creating conan packages. In this case it will be used in order to manage dependencies for `Application-A`, which are the open-source libraries `xerces` and `yaml-cpp`.

3. In Figure 5.10 it is illustrated how the `conanfile.py` has to look like in order to get these two dependencies. `generators = "visual studio"` describes for which kind of a project the dependencies are needed, which in our case is a visual studio project and the `requirements` function contains the name of the libraries which are needed.

4. Afterwards the command `conan install .` has to be executed in the root directory where the Visual Studio solution, with the `.sln` extension, and the `conanfile.py` file is. This downloads all the required dependencies and generates a `conanbuildinfo.props` file, which needs to be added to the Visual Studio project via the Property Manager of Visual Studio (See [115]). Doing so, adds a reference to the location where the Conan package are downloaded and stored with the `$(ConanIncludeDirectories)` variable as seen in Figure 5.11.

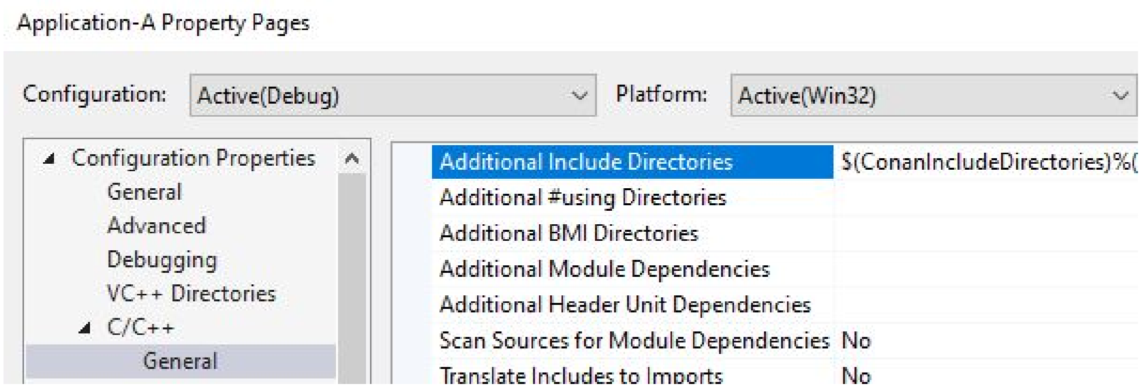
5. Thereafter a project file with the extension `.vcxproj` will be created, this file includes the changes to the dependency directories which were done in the previous step and needs to be pushed to the Git repository.

6. Once this file is in the repository, everyone who uses this software application can simply download all the C++ dependencies with the command `conan install .` in the root project folder and the whole application is able to compile.



```python
from conans import ConanFile, MSBuild

class ApplicationAConan(ConanFile):
    name = 'Application-A'
    version = "1.0.0"
    description = 'Application-A'

    generators = "visual_studio"

    def requirements(self):
        self.requires('xerces-c/3.2.3')
        self.requires('yaml-cpp/0.6.3')
```

Figure 5.10: Application-A conanfile.py

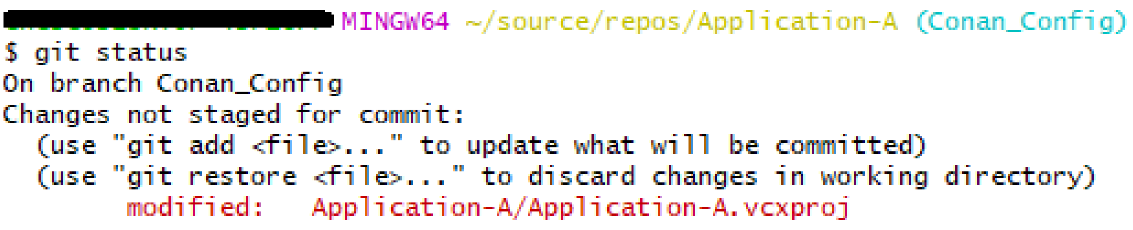


Application-A Property Pages

Configuration: Active(Debug)   Platform: Active(Win32)

Additional Include Directories	$(ConanIncludeDirectories)%(
Additional #using Directories	
Additional BMI Directories	
Additional Module Dependencies	
Additional Header Unit Dependencies	
Scan Sources for Module Dependencies	No
Translate Includes to Imports	No

Figure 5.11: Application-A Project Properties



```
[redacted] MINGW64 ~/source/repos/Application-A (Conan_Config)
$ git status
On branch Conan_Config
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   Application-A/Application-A.vcxproj
```

Figure 5.12: Application-A Project Property File

5.6 TeamCity as Automation Build Tool

Up until this point the updated infrastructure has a unified place for automatically auditing and continuously integrating source code, a single source where all binaries and libraries can be stored, downloaded and continuously deployed, yet to complete the CI/CD workflow it is necessary to have a build automation system (See 2.6) which continuously compiles and builds source code, continuously tests the implemented changes and the over-all integration of all components and continuously deploys when it is demanded.

Why TeamCity?

When looking into both tools which were mentioned in section 2.6, the fact that TeamCity audits user and build history without the need of any configuration and the fact that most features and functionalities are usable out of the box, unlike the open-source approach of Jenkins, where a lot of functionalities are based on third-party open-source plugins like GitHub integration or the a customizable dashboard view, were decisive reasons for TeamCity.

TeamCity Terminology

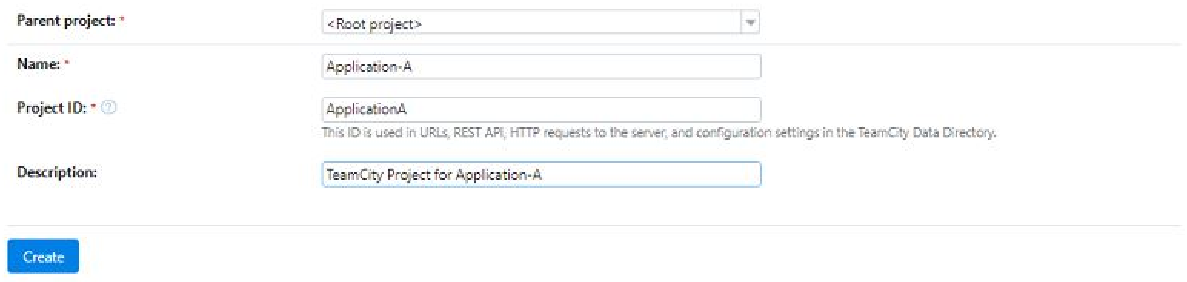
Here will be the different terms for TeamCity explained, in order to understand the following topics and sections [72][82]:

1. Build: A process that completes a certain CI/CD task. The majority of builds consist of numerous sequential build steps that each carry out a specific action. A build is executed according to the settings specified in its build configuration.
2. Build Chain: A series of builds coupled by snapshot dependencies that form a pipeline.
3. Build Configuration: A build configuration in TeamCity is a set of settings utilized to start a build or a sequence of builds as part of a build chain. A list of all recent builds is displayed on a build configuration's home page. Checking out source code, running integration tests, building artifacts, preparing release deployments and "nightly" builds are a few examples of build configuration.
4. Build Parameter: A name-value pair, which can be defined by users and can be used in builds. They enable to flexibly share settings and values to pass them to build steps, which can be accessed by two percentage signs inside build configurations.
5. Build Runner: A TeamCity module that enables integration with a particular tool, such as Gradle, the Command Line,.NET, Kotlin Script. Each build step specifies the execution runner that will be used.
6. Build Step: A job that a build runner will carry out. Multiple build steps can be included in a single build configuration.
7. Environment Parameter: A type of build parameter that is passed into a build process. Defined by the `env.` prefix.
8. Project: Is a collection of build configurations. A project may be a software project, a particular version or release of a project or any other logical set of build configurations. For each of its build configurations, a project sets common settings.

9. VCS Root: TeamCity must establish a connection to the VCS and whenever it needs to obtain the source code, VCS Root has to be configured. With it TeamCity keeps track of VCS changes and obtains the sources needed for a particular build configuration.
10. Root project: A default project at the top of the complete project hierarchy. Its settings are available to all the other projects. It is created by default and cannot be deleted.
11. Snapshot dependency: A connection between build settings that enables the assignment of numerous builds to the same source revision (commit), ensuring that the same project files are used across all building phases.

Project Configuration

After the installation and setup of TeamCity (See [92]), it is then feasible to create a project configurations. As displayed in Figure 5.13 a project for Application-A is created.

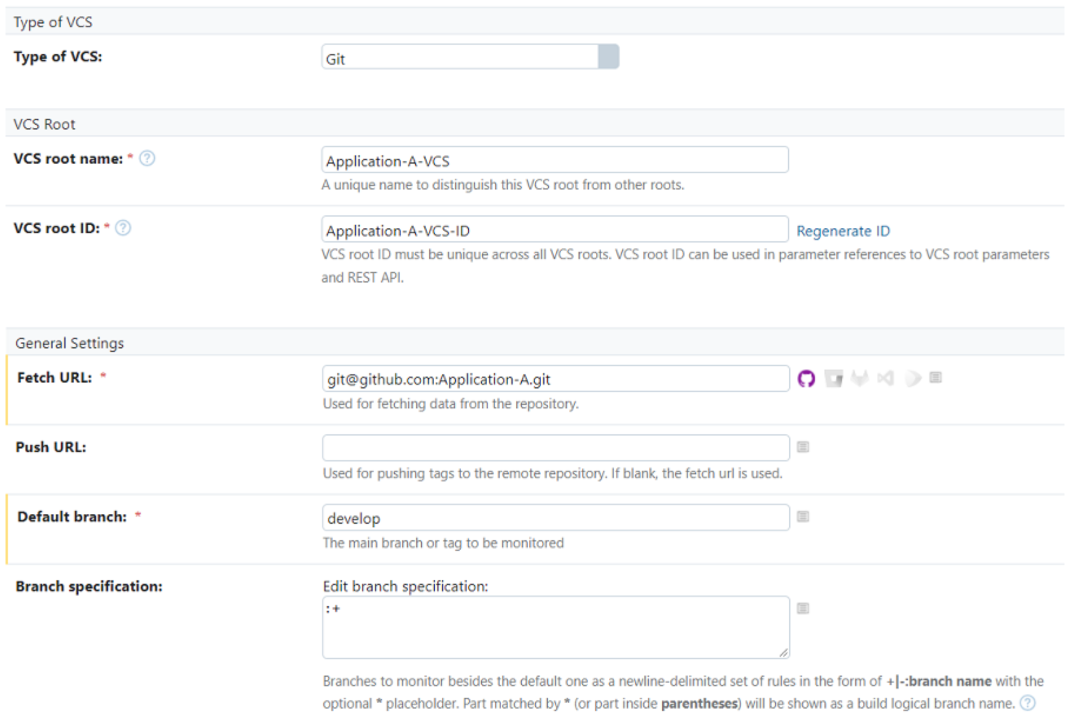


Parent project: *	<Root project>
Name: *	Application-A
Project ID: *	ApplicationA This ID is used in URLs, REST API, HTTP requests to the server, and configuration settings in the TeamCity Data Directory.
Description:	TeamCity Project for Application-A

Figure 5.13: Application-A TeamCity Project Creation

5.7 Build Configuration Strategy

Inside that project it is possible to create different build configurations. As mentioned above, many build configurations can be created for many different reasons and serve different purposes even for one repository. For example it is possible to have a dedicated build configuration which only compiles and executes the unit tests of a source code, another build configuration may be dedicated to test the application with different test data for the same source code. Since a project is created in the previous step it is then possible to create all the different build configuration for a specific project or repository in there. In order to do that, it is necessary to set up a VCS Root for the particular repository, which can be reused by every build configuration afterwards. Like shown in Figure 5.14 in order to create a VCS Root it is needed to set the URL to the Git repository of the project and since GitFlow is used for the Application-A repository the default branch, which should be monitored is the `develop`. An important thing to notice is also the `:*` in the `Branch specification` setting, which indicates that every other branch should also be monitored(for a more detailed explanation See [147] [49]).



Type of VCS	
Type of VCS:	Git
VCS Root	
VCS root name: *	Application-A-VCS A unique name to distinguish this VCS root from other roots.
VCS root ID: *	Application-A-VCS-ID — Regenerate ID VCS root ID must be unique across all VCS roots. VCS root ID can be used in parameter references to VCS root parameters and REST API.
General Settings	
Fetch URL: *	git@github.com:Application-A.git Used for fetching data from the repository.
Push URL:	 Used for pushing tags to the remote repository. If blank, the fetch url is used.
Default branch: *	develop The main branch or tag to be monitored
Branch specification:	Edit branch specification: :+ Branches to monitor besides the default one as a newline-delimited set of rules in the form of **+|-:branch name** with the optional * placeholder. Part matched by * (or part inside **parentheses**) will be shown as a build logical branch name.

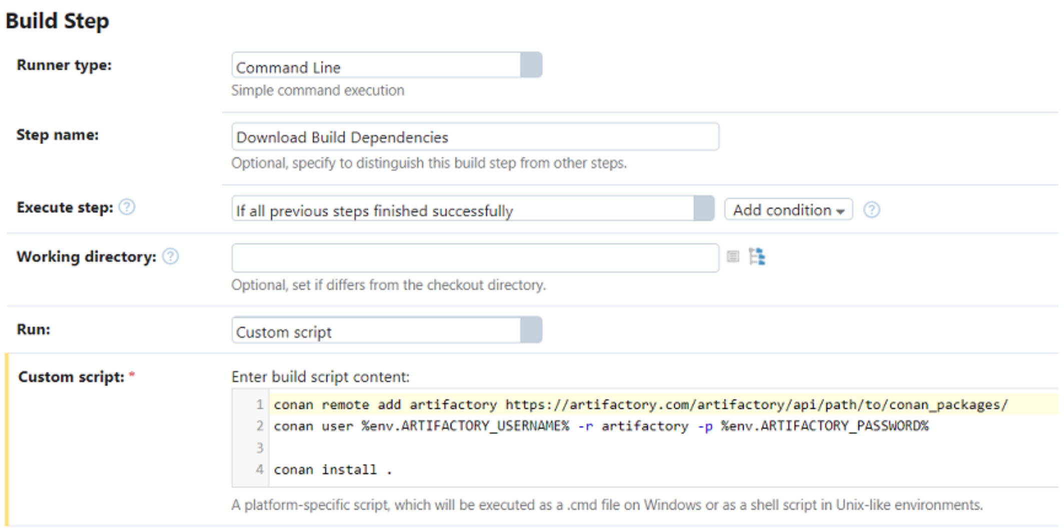
Figure 5.14: Application-A VCS Root

After that it is possible to configure and add build steps to a build configuration.
The following subsections will describe and show build configurations which will cover general use cases.

Build Configuration: Build and Test

The first build configuration is responsible to build and run unit test whenever changes are made in Git repository.
This is done with the following build steps and configurations:

Since Application-A has build dependencies which need to be present in order to compile the application it is necessary to download these dependencies. However, since previously Conan is configured for Application-A to automatically handle the build dependencies it is simple for TeamCity to use it as it is for developers. As stated in subsection Managing Build Dependencies in order to download the dependencies with conan the remote conan repository, which is this case Artifactory, and the credentials for it has to be configured and then the command `conan install .` has to be executed from the command line. Therefore the first build step has the have a `Command Line` Build Runner which uses a common CLI. For this reasons the first build steps looks like the following Figure 5.15. As it can be seen, user name and password for Artifactory are configured in environment parameters in order to not have them in written in clear text inside build configurations (See [46] [72]).



Build Step

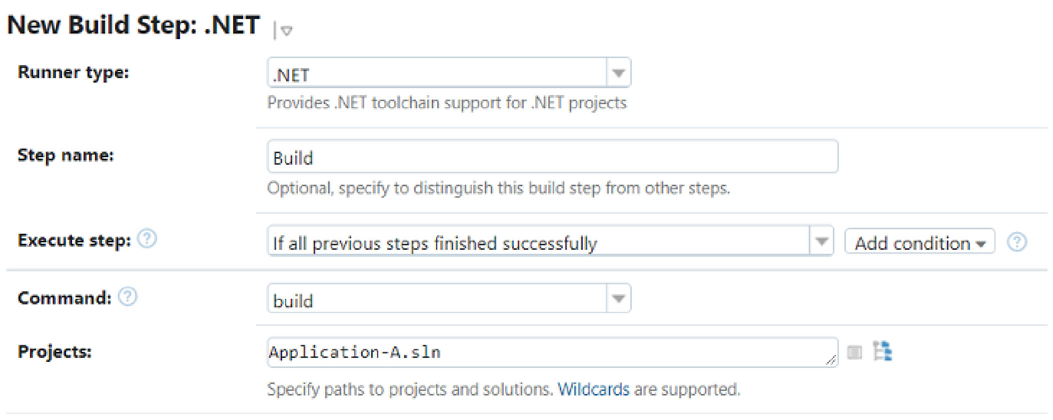
Runner type:	Command Line Simple command execution
Step name:	Download Build Dependencies Optional, specify to distinguish this build step from other steps.
Execute step:	If all previous steps finished successfully — Add condition
Working directory:	Optional, set if differs from the checkout directory.
Run:	Custom script
Custom script: *	Enter build script content:

```
conan remote add artifactory https://artifactory.com/artifactory/api/path/to/conan_packages/
conan user %env.ARTIFACTORY_USERNAME% -r artifactory -p %env.ARTIFACTORY_PASSWORD%

conan install .
```

A platform-specific script, which will be executed as a .cmd file on Windows or as a shell script in Unix-like environments.

Figure 5.15: Conan Build Configuration Setup

With the previous build step getting all the necessary build dependencies it is then possible to build and compile Application-A. Since Application-A is a Visual Studio project, TeamCity allows to use the `.NET` Build Runner (See [1]), which enables to build .NET projects. It is important to notice that the `Projects` option in the build step includes the name of project solution file of Application-A as shown in Figure 5.16.

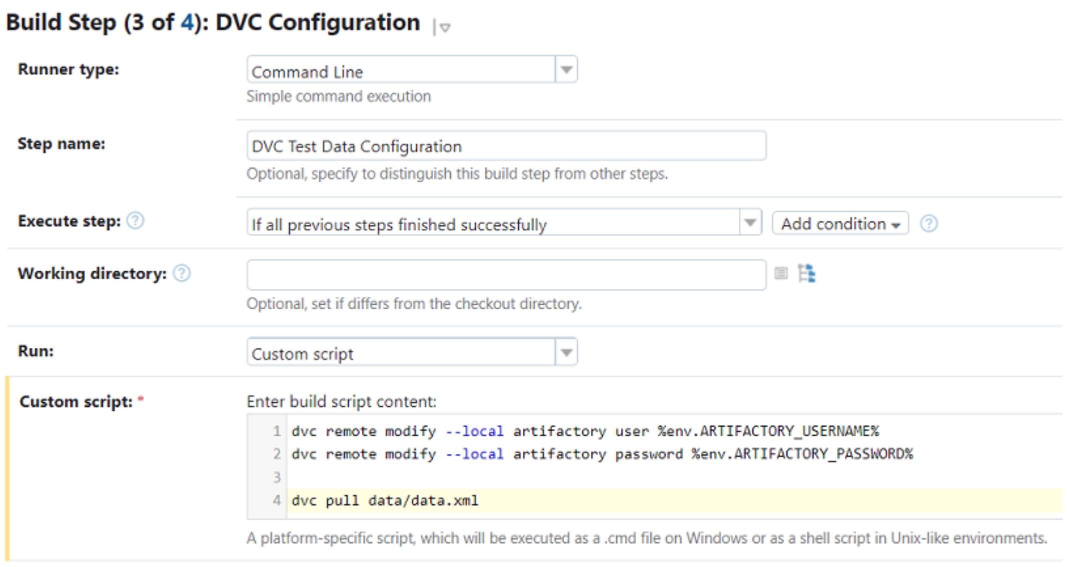


New Build Step: .NET

Runner type:	.NET Provides .NET toolchain support for .NET projects
Step name:	Build Optional, specify to distinguish this build step from other steps.
Execute step:	If all previous steps finished successfully · Add condition
Command:	build
Projects:	`Application-A.sln` Specify paths to projects and solutions. Wildcards are supported.

Figure 5.16: .NET Compile Build Configuration

After building the software application successfully, indicating that there are no compile errors in the code, it is then wanted to run software tests in order to be sure that changes do not introduce unwanted behaviour or bugs. Since the test data is versioned and managed with DVC it is necessary to also configure DVC for the build configuration, in order to run the test with the test data. Therefore, according to the DVC setup in section Data Version Control, the next build steps looks like the following Figure 5.17:



Build Step (3 of 4): DVC Configuration

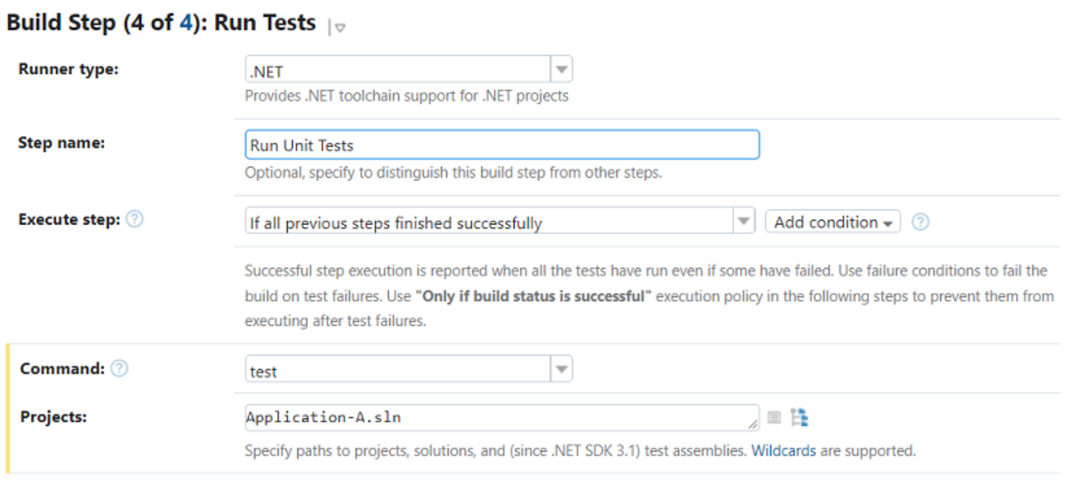
Runner type:	Command Line Simple command execution
Step name:	DVC Test Data Configuration Optional, specify to distinguish this build step from other steps.
Execute step:	If all previous steps finished successfully
Working directory:	Optional, set if differs from the checkout directory.
Run:	Custom script
Custom script: *	Enter build script content:

```
dvc remote modify --local artifactory user %env.ARTIFACTORY_USERNAME%
dvc remote modify --local artifactory password %env.ARTIFACTORY_PASSWORD%

dvc pull data/data.xml
```

A platform-specific script, which will be executed as a .cmd file on Windows or as a shell script in Unix-like environments.

Figure 5.17: DVC Build Configuration

To run software unit tests it is again needed to use the .NET Build Runner with the `test` command, as seen in Figure 5.18.

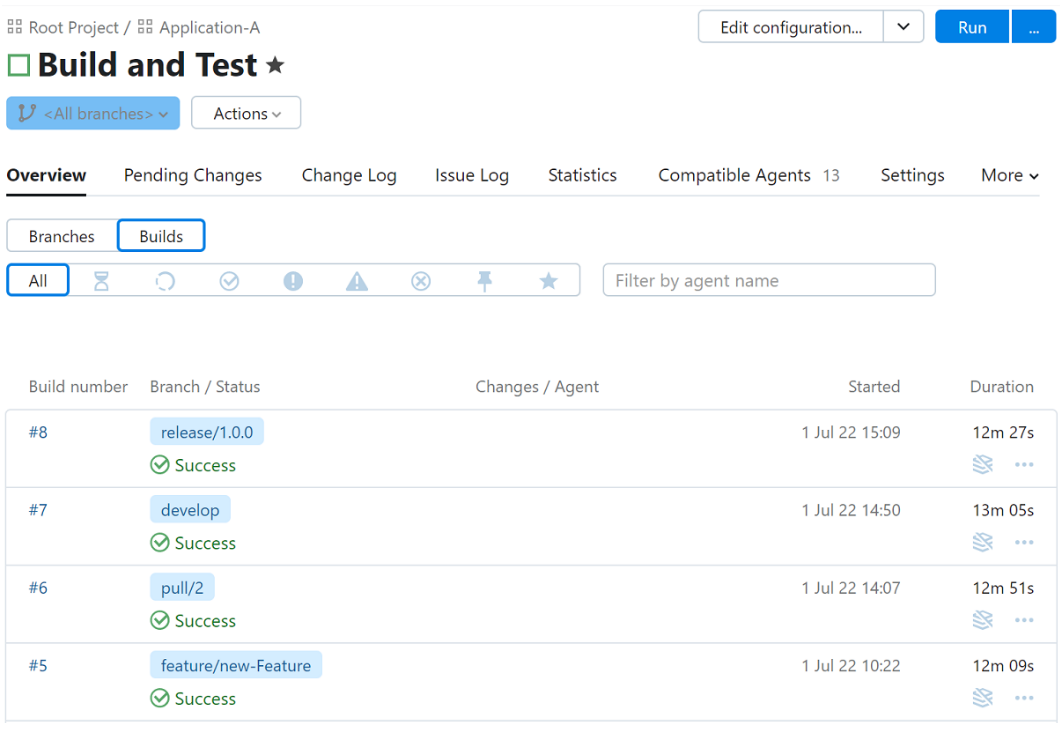


Build Step (4 of 4): Run Tests

Runner type:	.NET Provides .NET toolchain support for .NET projects
Step name:	Run Unit Tests Optional, specify to distinguish this build step from other steps.
Execute step:	If all previous steps finished successfully Successful step execution is reported when all the tests have run even if some have failed. Use failure conditions to fail the build on test failures. Use **"Only if build status is successful"** execution policy in the following steps to prevent them from executing after test failures.
Command:	test
Projects:	Application-A.sln Specify paths to projects, solutions, and (since .NET SDK 3.1) test assemblies. Wildcards are supported.

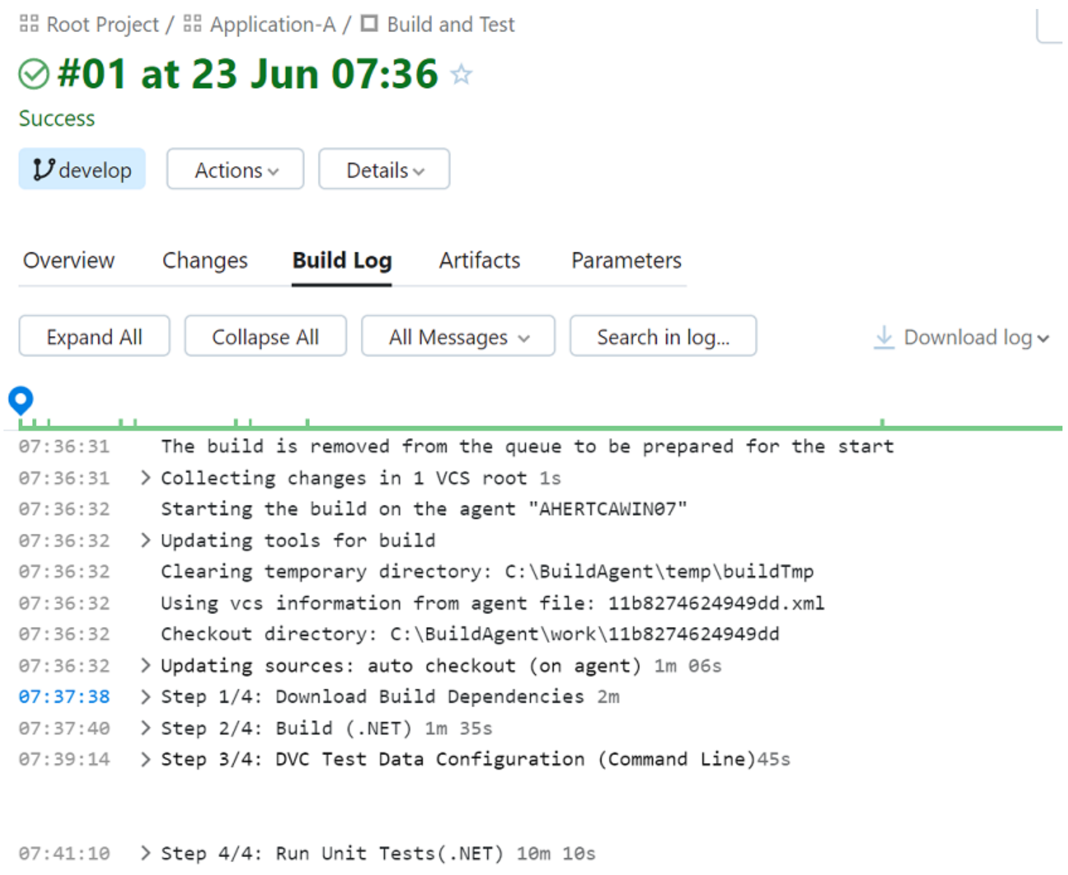
Figure 5.18: .NET Test Build Configuration

With all this configurations, this build configuration is then triggered to build whenever a change has been committed to a branch in the Git repository, as shown in Figure 5.19 and the code will be compiled and the unit tests will be run. The result of this configuration is demonstrated in Figure 5.20.



Build number	Branch / Status	Changes / Agent	Started	Duration
#8	release/1.0.0 Success		1 Jul 22 15:09	12m 27s
#7	develop Success		1 Jul 22 14:50	13m 05s
#6	pull/2 Success		1 Jul 22 14:07	12m 51s
#5	feature/new-Feature Success		1 Jul 22 10:22	12m 09s

Figure 5.19: Application-A Build and Test Overview



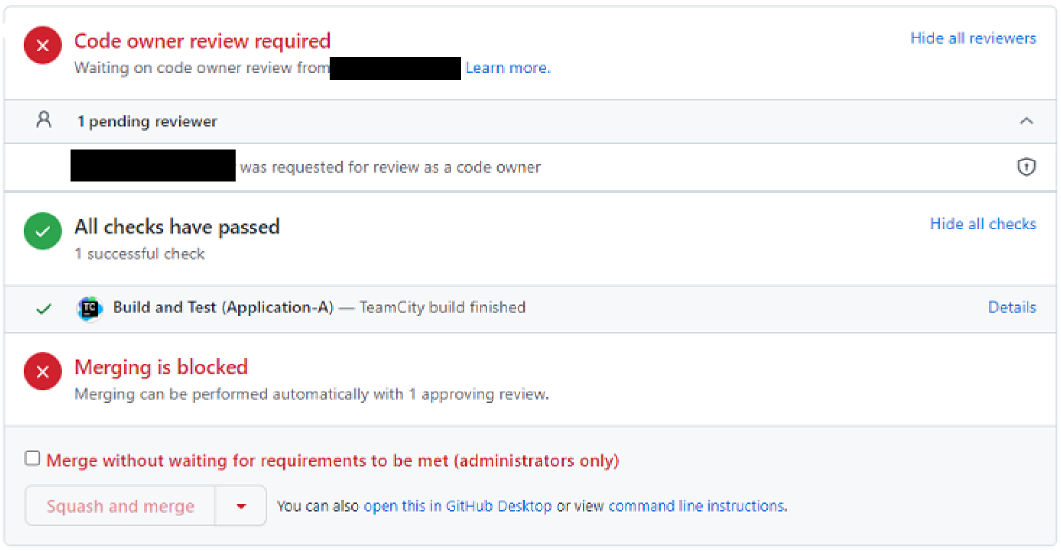
```
07:36:31     The build is removed from the queue to be prepared for the start
07:36:31   > Collecting changes in 1 VCS root 1s
07:36:32     Starting the build on the agent "AHERTCAWIN07"
07:36:32   > Updating tools for build
07:36:32     Clearing temporary directory: C:\BuildAgent\temp\buildTmp
07:36:32     Using vcs information from agent file: 11b8274624949dd.xml
07:36:32     Checkout directory: C:\BuildAgent\work\11b8274624949dd
07:36:32   > Updating sources: auto checkout (on agent) 1m 06s
07:37:38   > Step 1/4: Download Build Dependencies 2m
07:37:40   > Step 2/4: Build (.NET) 1m 35s
07:39:14   > Step 3/4: DVC Test Data Configuration (Command Line)45s


07:41:10   > Step 4/4: Run Unit Tests(.NET) 10m 10s
```

Figure 5.20: Application-A Build and Test

Build Configuration: On Pull Request

With the preceding build configuration every change done to Application-A is checked automatically, yet to prevent, that unstable and bug contained code is getting merged to the primary branch, GitHub and TeamCity introduced the Pull Request feature. The Pull Requests build feature enables to automatically load pull request data and source code and run builds on the pull request branches in GitHub on specific build configurations (See [123]). Additionally, in order to receive the outcome of the triggered pull request build, there is the Commit Status Publisher feature (See [40]).
When both features are configured, TeamCity instantly starts a build whenever a pull request is created and the result is getting forward to the pull request on the VCS repository. As shown in Figure 5.21 a pull request was opened and the `Build and Test` build configuration of Application-A is run successfully, indicating that the changes compiled and the software unit tests run successfully, but since the approval of a code

owner is still missing, GitHub does not allow the pull request to be merged.


Code owner review required
Waiting on code owner review from [redacted] Learn more.

Hide all reviewers

1 pending reviewer

[redacted] was requested for review as a code owner

All checks have passed
1 successful check

Hide all checks

✓ Build and Test (Application-A) — TeamCity build finished

Details

Merging is blocked
Merging can be performed automatically with 1 approving review.

☐ Merge without waiting for requirements to be met (administrators only)

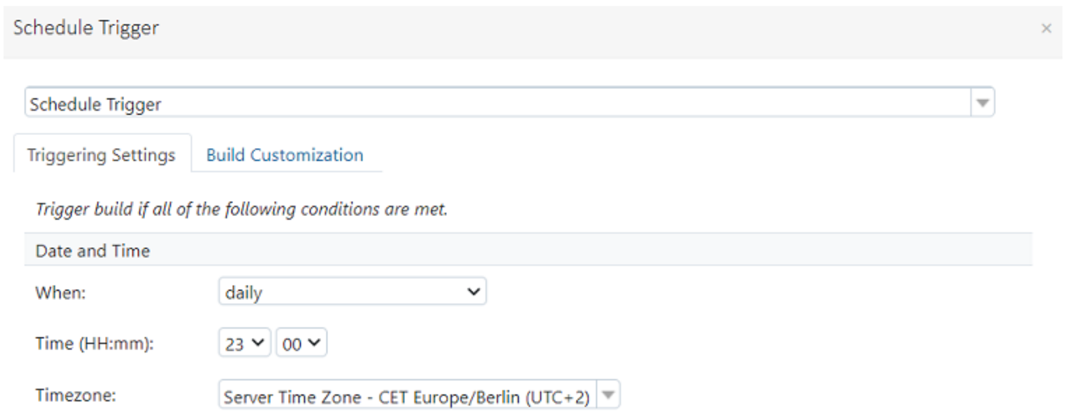
Squash and merge

You can also open this in GitHub Desktop or view command line instructions.

Figure 5.21: Application-A Pull Request Status

Build Configuration: Nightly Build

As mentioned in Nightly Builds completely building and testing larger projects with a big amount of test data can be time consuming, which is a problem when the time consuming task blocks an entire build agent for multiple hours during the day, meaning no other builds can be run on that agent. Therefore, following the recommendation of [27], a possibility is to run the complete builds with all the test data during the night. For this purpose TeamCity has the feature `Schedule Trigger`, which allows to schedule builds based on time [48]. Utilizing this feature enables to create a new build configuration which is dedicated to build every night at a specific time, for example every day at 11pm as shown in Figure 5.22.


Schedule Trigger

Schedule Trigger

Triggering Settings | Build Customization

Trigger build if all of the following conditions are met.

Date and Time

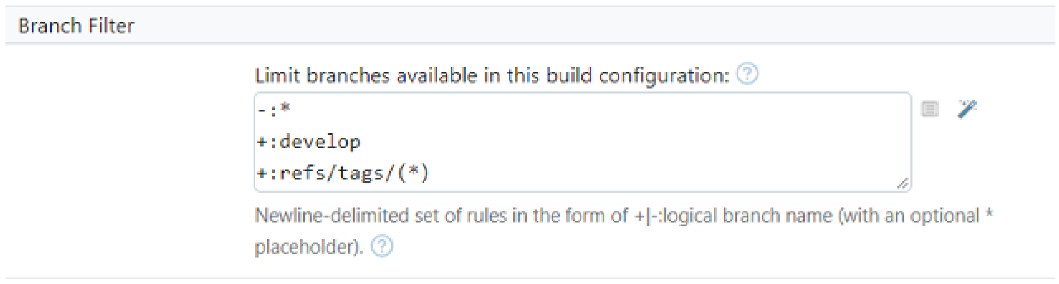
When:	daily
Time (HH:mm):	23 00
Timezone:	Server Time Zone - CET Europe/Berlin (UTC+2)

Figure 5.22: Application-A Schedule Trigger

Build Configuration: Publish and Deploy

After implementing and introducing features and the software application is in a deployable state, so that it can be used in the test or production environment, it is possible to have a dedicated build configuration to deploy and deliver the current state of software applications. However, before doing that it is recommended to figure out when the software application will be used in a test environment and when in production, since according to [87] the build, deployment and test process that is applied to the software application change is the validation whether the current state of a project is a release candidate or not.
Therefore, the following build configuration will build the whole application when there are changes introduced to the primary development branch `develop` and with the configuration which were done until now, e.g. code owner and pull request checks, the branch should be always stable. After the building and compiling of the application the build configuration will then deploy that state of the project as a test release to Artifactory, so that the applications and builds which depend on that will be able to consume and test it, in order to determine if that state is a release candidate or not. Additionally, whenever a release is created on GitHub (See [107]), the build configuration will build and deploy the release version with the given version number to Artifactory, so it can be used in the test and production environment.
For the new build configuration, it must be defined which branches it should monitor. In this case it will the `develop` branch and whenever releases are created. Since this build configuration is using the same VCS Root as the one at the beginning of this section, which monitors all the branches, the already monitored branches have to be filtered.

This is done by excluding all the branches which are monitored by the VCS Root and including `develop` and `refs/tags/(*)` (See [49]), as shown in Figure 5.23.


Branch Filter

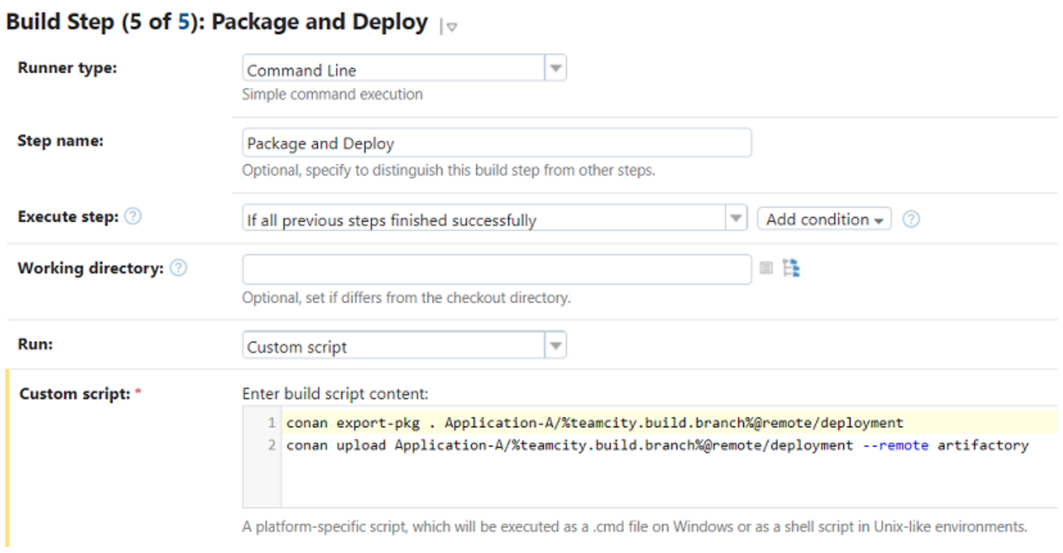
Limit branches available in this build configuration:

```
-:*
+:develop
+:refs/tags/(*)
```

Newline-delimited set of rules in the form of +|-:logical branch name (with an optional * placeholder).

Figure 5.23: Application-A Branch Filter for Deployment

The first couple of build steps are the same as the "Build and Test" build configuration, since it is wanted to build and test the application before deploying and making it public. The last build step however, will create a conan package out of the application and then deploy it to Artifactory, which has to look like the following Figure 5.24. Since the remote conan repository is set in the first build step under the alias `artifactory`, it can also be used in this build step as the target remote repository for the deployment. Additionally, the package will be versioned the branch name,


Build Step (5 of 5): Package and Deploy

Runner type: Command Line
Simple command execution

Step name: Package and Deploy
Optional, specify to distinguish this build step from other steps.

Execute step: If all previous steps finished successfully

Working directory:
Optional, set if differs from the checkout directory.

Run: Custom script

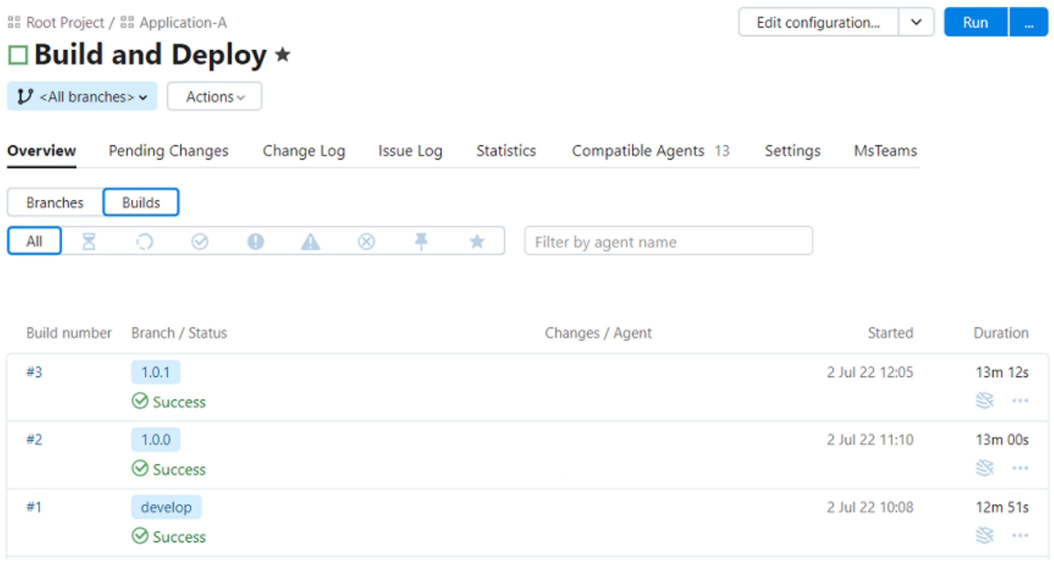
Custom script: *
Enter build script content:

```
conan export-pkg . Application-A/%teamcity.build.branch%@remote/deployment
conan upload Application-A/%teamcity.build.branch%@remote/deployment --remote artifactory
```

A platform-specific script, which will be executed as a .cmd file on Windows or as a shell script in Unix-like environments.

Figure 5.24: Application-A Package and Deploy

With this configuration in place, the "Build and Deploy" build configuration, will always run whenever a commit is done on the `develop` branch or a release is created, as shown in Figure 5.25, which ultimately creates and deploys the package according to the `TeamCity Build Branch` name.



Build number	Branch / Status	Changes / Agent	Started	Duration
#3	1.0.1 Success		2 Jul 22 12:05	13m 12s
#2	1.0.0 Success		2 Jul 22 11:10	13m 00s
#1	develop Success		2 Jul 22 10:08	12m 51s

Figure 5.25: Application-A Build And Deploy Builds

The result of these builds is that Artifactory then contains the correct versioned conana packages, as displayed in Figure 5.26, which are ready to be consumed by other applications and builds. It is important to notice that the package names match accordingly to the versions of the releases and whereas the `develop` package get overridden whenever a new change is done to the `develop` branch, which means that this package always represents the most current state of the application.

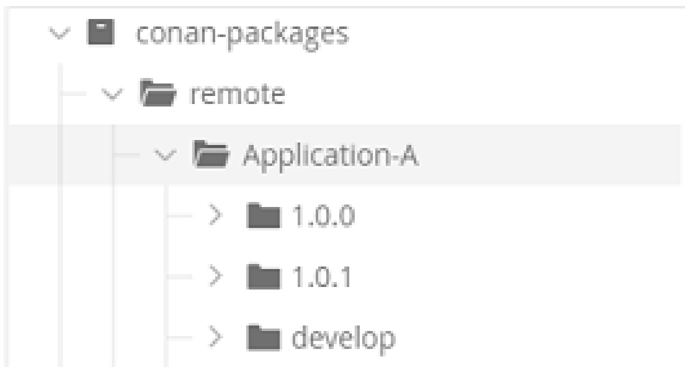
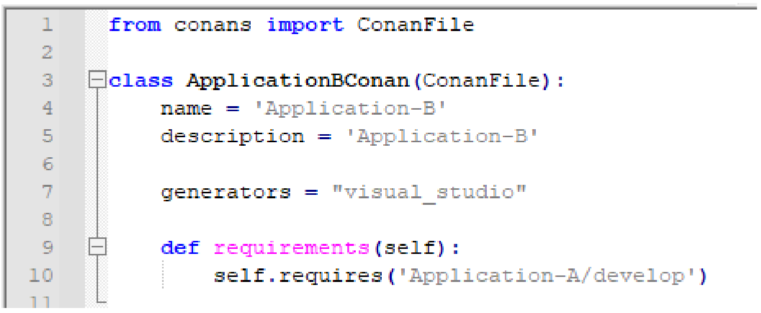


Figure 5.26: Conan Packages in Artifactory

5.8 Integration-Testing

As mentioned at the beginning of section 5.1 Application-B has a dependency on Application-A in order to build and compile properly. Which means that it must be continuously ensured that whenever a change is done to the primary branch of Application-A, that Application-B is still able to compile and use Application-A. Therefore a build configuration will be created, as part of a build-chain, which will always be built, whenever the "Build and Deploy" configuration is triggered. After the "Build and Deploy" has successfully deployed a new `develop` conan package, the new Application-B build configuration will then pull the new package and compile Application-B.
The first thing which needs to be done is to configure the conan.py file of Application-B, so that it uses the `develop` conan package of Application-A, as illustrated in Figure 5.27. Secondly, the build steps of the new build configuration for Application-B are the same as the "Build and Test" of Application-A, but with one addition. A snapshot dependency has to be configured to the "Build and Deploy" configuration of Application-A, in order to create build-chain, so that the new configuration is build always after the "Build and Deploy". Within TeamCity then it is possible to illustrate the build chain, as shown in Figure.



```
from conans import ConanFile

class ApplicationBConan(ConanFile):
    name = 'Application-B'
    description = 'Application-B'

    generators = "visual_studio"

    def requirements(self):
        self.requires('Application-A/develop')
```

Figure 5.27: Application-B ConanFile.py

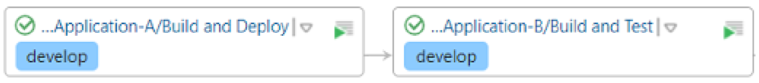

...Application-A/Build and Deploy
develop
...Application-B/Build and Test
develop

Figure 5.28: TC Build Chain

6 Conclusion

This thesis examined recent literature in the field of software engineering that is focused on modern software practices and infrastructures for software development. The research covered numerous sources and summarized essential information about Continuous Integration and Delivery, Software Configuration Management, different branching and versioning strategies, dependency management and build and test automation. After the comprehensive literature study a thorough implementation strategy for transitioning from old systems and applications to a postmodern software infrastructure was presented, by including most of the technologies, strategies and practices which were concluded by the thorough research.

The suggested strategy incorporated the four major components of software configuration management, by introducing different practices and technologies to cover all four of them.
Firstly with Configuration Identification, by enabling to identify and divide projects into smaller, easier to manage subsystems. Which was done with Git repositories for source code and configuration files and with Artifactory for binaries and dependencies.
Then with Configuration Status Accounting by enabling to keep track of when, why and who made changes to the different components of a software application. Which was done with Git for source code changes, with DVC for test data and by utilizing GitFlow which provided meaningful branching names in order to have an overview of changes and their purpose.

Following that with Configuration Auditing, by permitting to track of advancement with the change history log of GitHub and build history log of TeamCity.

Then with Configuration Change Control by allowing to organize the access of team members to project components so unwanted and unauthorized changes are not performed, which was done with Git code owners and branch protection.

Furthermore, the proof of concept includes a strategy for dependency management of C++ libraries, by using Conan as a package manager and Artifactory for hosting and administration. In addition the presented proof of concept includes a fully working Continuous Integration / Continuous Delivery infrastructure with Continuous Testing on top of it.

Yet, the infrastructure with the suggested combination of the technologies and workflows, is as mentioned a proof of concept, which worked for the presented situation and use case. However, when the implemented strategy might not be sufficient enough, depending on the demands and needs, an alternative technology stack may be more compatible. Alternatively, for example, when dealing with vast amount of binary files, such as in the video game industry, Perforce might be a more reasonable choice as a VCS, since it is capable of managing binary files better then Git, therefore getting rid of Artifactory. Additionally, Perforce allows to partially checkout repositories, which is more ideal when having extensive amount of data which do not have to be downloaded all the time.

Seeing that how many different technologies, best practices, strategies and aspects to take care of, throughout the research, the formulating of a functioning strategy and the implementation of these, elucidated the challenges and hesitation of companies to switch to more modern software development strategies. The investigation of which practice may assist for which stumbling block was a challenge in itself, since keeping an overview about the vast amount of information was an extraordinarily obstacle, over and above that comprehending and implementing all the theoretical aspects was indeed a difficult task. Since even seemingly small approaches and details such as branching strategies had a bigger impact after all.

Bibliography

[1] *.NET | TeamCity On-Premises*. en-US. URL: https://www.jetbrains.com/help/teamcity/net.html (visited on 07/02/2022).

[2] *10 Risks of Keeping a Legacy Software*. en. Apr. 2021. URL: https://relevant.software/blog/10-risks-of-keeping-a-legacy-software/ (visited on 05/30/2022).

[3] *7 Best Practices for Managing Open Source Components*. en-US. URL: https://www.altexsoft.com/blog/engineering/5-best-practices-for-managing-open-source-components/ (visited on 05/31/2022).

[4] *7 Reasons Why Software Testing is Important*. en-US. Apr. 2021. URL: https://www.indiumsoftware.com/blog/why-software-testing/ (visited on 05/31/2022).

[5] *9 Types Of Software Testing In Software Engineering*. en-US. Sept. 2019. URL: https://theqalead.com/topics/types-of-software-testing/ (visited on 05/31/2022).

[6] *A successful Git branching model*. en. URL: http://nvie.com/posts/a-successful-git-branching-model/ (visited on 05/31/2022).

[7] *About code owners*. en. URL: https://ghdocs-prod.azurewebsites.net/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-code-owners (visited on 06/30/2022).

[8] *Advantages and Disadvantages of Jenkins*. URL: https://www.tutorialandexample.com/advantages-and-disadvantages-of-jenkins (visited on 05/31/2022).

[9] *Anti Patterns of Continuous Integration | HackerNoon*. URL: https://hackernoon.com/anti-patterns-of-continuous-integration-e1cafd47556d (visited on 05/31/2022).

[10] *Anti-patterns You Should Avoid in Your Code*. en. Nov. 2020. URL: https://www.freecodecamp.org/news/antipatterns-to-avoid-in-code/ (visited on 05/31/2022).

[11] *Approaches to C++ Dependency Management, or Why We Built Buckaroo | HackerNoon.* en. URL: `https://hackernoon.com/approaches-to-c-dependency-management-or-why-we-built-buckaroo-26049d4646e7` (visited on 05/31/2022).

[12] *Artifact management overview.* en. URL: `https://cloud.google.com/artifact-management/docs/overview` (visited on 05/31/2022).

[13] *Artifact management overview.* en. URL: `https://docs.cloudbees.com/docs/cloudbees-cd/latest/automation-platform/artifactmgmt` (visited on 05/31/2022).

[14] *Artifact Management Overview | JFrog Platform.* en-US. URL: `https://jfrog.com/artifact-management/` (visited on 05/31/2022).

[15] *Artifactory - Universal Artifact Management.* en-US. URL: `https://jfrog.com/artifactory/` (visited on 05/31/2022).

[16] *Artifactory: How to upload a folder (with its content) to Artifactory.* en-US. URL: `https://jfrog.com/knowledge-base/artifactory-how-to-upload-a-folder-with-its-content-to-artifactory/` (visited on 06/25/2022).

[17] Atlassian. *Continuous Delivery Pipeline 101.* en. URL: `https://www.atlassian.com/continuous-delivery/principles/pipeline` (visited on 05/31/2022).

[18] Atlassian. *Continuous integration vs. delivery vs. deployment.* en. URL: `https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment` (visited on 04/24/2022).

[19] Atlassian. *Git Feature Branch Workflow | Atlassian Git Tutorial.* en. URL: `https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow` (visited on 05/31/2022).

[20] Atlassian. *Gitflow Workflow | Atlassian Git Tutorial.* en. URL: `https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow` (visited on 05/30/2022).

[21] Atlassian. *Perforce to Git - why to make the move | Atlassian Git Tutorial.* en. URL: `https://www.atlassian.com/git/tutorials/perforce-git` (visited on 05/08/2022).

[22] Atlassian. *Trunk-based Development.* en. URL: `https://www.atlassian.com/continuous-delivery/continuous-integration/trunk-based-development` (visited on 05/30/2022).

[23] Atlassian. *What is Continuous Integration.* en. URL: `https://www.atlassian.com/continuous-delivery/continuous-integration` (visited on 06/27/2022).

[24] Atlassian. *What is Git: become a pro at Git with this guide | Atlassian Git Tutorial.* en. URL: `https://www.atlassian.com/git/tutorials` (visited on 05/02/2022).

[25] Atlassian. *What is version control | Atlassian Git Tutorial.* en. URL: `https://www.atlassian.com/git/tutorials/what-is-version-control` (visited on 05/02/2022).

[26] *Audit Configuration - an overview | ScienceDirect Topics.* URL: `https://www.sciencedirect.com/topics/computer-science/audit-configuration` (visited on 04/24/2022).

[27] Arnon Axelrod. *Complete Guide to Test Automation: Techniques, Practices, and Patterns for Building and Maintaining Effective Software Projects.* en. Berkeley, CA: Apress, 2018. ISBN: 978-1-4842-3831-8 978-1-4842-3832-5. URL: `http://link.springer.com/10.1007/978-1-4842-3832-5` (visited on 06/12/2022).

[28] *Best Practices for Successful CI/CD | TeamCity CI/CD Guide.* en. URL: `https://www.jetbrains.com/teamcity/ci-cd-guide/ci-cd-best-practices/` (visited on 05/31/2022).

[29] *bliki: DarkLaunching.* URL: `https://martinfowler.com/bliki/DarkLaunching.html` (visited on 05/31/2022).

[30] *bliki: FeatureBranch.* URL: `https://martinfowler.com/bliki/FeatureBranch.html` (visited on 05/31/2022).

[31] *Branching Content Hub.* en. URL: `https://www.perforce.com/resources/vcs/version-control-branching` (visited on 05/03/2022).

[32] *Build Automation - Your Guide to an Automated Build Process | Buildd.co.* URL: `https://buildd.co/product/build-automation` (visited on 05/31/2022).

[33] *Build Automation: How it Works and Which Tools to Use.* en-US. URL: `https://www.altexsoft.com/blog/build-automation/` (visited on 05/31/2022).

[34] *C/C++ Open Source Package Manager.* URL: `https://conan.io/` (visited on 06/26/2022).

[35] *Centralized vs Distributed Version Control: Which One Should We Choose?* en-us. Aug. 2019. URL: `https://www.geeksforgeeks.org/centralized-vs-distributed-version-control-which-one-should-we-choose/` (visited on 05/30/2022).

[36] Lianping Chen. “Continuous Delivery: Huge Benefits, but Challenges Too”. en. In: *IEEE Softw.* 32.2 (Mar. 2015), pp. 50–54. ISSN: 0740-7459. DOI: 10.1109/MS.2015.27. URL: http://ieeexplore.ieee.org/document/7006384/ (visited on 04/24/2022).

[37] Luke Chen. *SemVer and CalVer — 2 popular software versioning schemes.* en. Nov. 2020. URL: https://nehckl0.medium.com/semver-and-calver-2-popular-software-versioning-schemes-96be80efe36 (visited on 05/31/2022).

[38] *CI/CD pipelines explained: Everything you need to know.* en. URL: https://www.techtarget.com/searchsoftwarequality/CI-CD-pipelines-explained-Everything-you-need-to-know (visited on 04/24/2022).

[39] *CM-03 Configuration Change Control.* URL: https://www.opensecurityarchitecture.org/cms/library/08_02_control-catalogue/154-08_02_CM-03 (visited on 04/24/2022).

[40] *Commit Status Publisher | TeamCity On-Premises.* en-US. URL: https://www.jetbrains.com/help/teamcity/commit-status-publisher.html (visited on 07/02/2022).

[41] *conan remote — conan 1.46.2 documentation.* URL: https://docs.conan.io/en/1.46/reference/commands/misc/remote.html (visited on 06/27/2022).

[42] *Conan Repositories - JFrog - JFrog Documentation.* URL: https://www.jfrog.com/confluence/display/JFROG/Conan+Repositories (visited on 06/21/2022).

[43] *conanfile.py — conan 1.49.0 documentation.* URL: https://docs.conan.io/en/latest/reference/conanfile.html (visited on 06/26/2022).

[44] *Configuration Identification - an overview | ScienceDirect Topics.* URL: https://www.sciencedirect.com/topics/engineering/configuration-identification (visited on 04/24/2022).

[45] *Configuration Status Accounting | Engineering360.* URL: https://www.globalspec.com/reference/71660/203279/5-4-configuration-status-accounting (visited on 04/24/2022).

[46] *Configuring Build Parameters | TeamCity On-Premises.* en-US. URL: https://www.jetbrains.com/help/teamcity/configuring-build-parameters.html (visited on 07/02/2022).

[47] *Configuring Build Triggers | TeamCity On-Premises.* en-US. URL: https://www.jetbrains.com/help/teamcity/configuring-build-triggers.html (visited on 05/31/2022).

[48] *Configuring Schedule Triggers | TeamCity On-Premises.* en-US. URL: https://www.jetbrains.com/help/teamcity/configuring-schedule-triggers.html (visited on 05/31/2022).

[49] *Configuring VCS Settings | TeamCity On-Premises.* en-US. URL: https://www.jetbrains.com/help/teamcity/configuring-vcs-settings.html (visited on 05/31/2022).

[50] Rajesh Kumar Mentor for DevOps-DevSecOps- SRE- Cloud- Container et al. *What is TeamCity and How it works? An Overview and Its Use Cases.* en-US. Apr. 2022. URL: https://www.devopsschool.com/blog/what-is-teamcity-and-how-it-works-an-overview-and-its-use-cases/ (visited on 05/31/2022).

[51] *Continuous Integration.* URL: https://martinfowler.com/articles/continuousIntegration.html (visited on 05/31/2022).

[52] *Continuous Integration Patterns and Anti-Patterns - DZone Refcardz.* en. URL: https://dzone.com/refcardz/continuous-integration (visited on 05/31/2022).

[53] *Data Version Control · DVC.* en. URL: https://dvc.org/ (visited on 06/22/2022).

[54] *Dependency Hell. What is Software Dependency Issues.* en. URL: https://www.boldare.com/blog/software-dependency-hell-what-is-it-and-how-to-avoid-it (visited on 05/31/2022).

[55] *Dependency Management: 3 Tips to Keep You Sane.* en-US. Mar. 2020. URL: https://www.mend.io/free-developer-tools/blog/dependency-management/ (visited on 05/31/2022).

[56] *Design Patterns and Refactoring.* en. URL: https://sourcemaking.com (visited on 05/31/2022).

[57] *Differences between Functional and Non-functional Testing.* en-us. May 2019. URL: https://www.geeksforgeeks.org/differences-between-functional-and-non-functional-testing/ (visited on 05/31/2022).

[58] Suemayah Eldursi. *A Guide to Semantic Versioning | Baeldung on Computer Science.* en-US. Mar. 2021. URL: https://www.baeldung.com/cs/semantic-versioning (visited on 05/31/2022).

[59] Nicole Forsgren and Jez Humble. *The Role of Continuous Delivery in IT and Organizational Performance.* en. SSRN Scholarly Paper 2681909. Rochester, NY: Social Science Research Network, Oct. 2015. DOI: 10.2139/ssrn.2681909. URL: https://papers.ssrn.com/abstract=2681909 (visited on 04/20/2022).

[60] C. D. Foundation. *CI/CD Patterns and Practices*. en-US. Sept. 2020. URL: https://cd.foundation/blog/2020/09/17/ci-cd-patterns-and-practices/ (visited on 04/24/2022).

[61] *Four Principles of Low-Risk Software Releases | Principle 1: Low-Risk Releases Are Incremental | InformIT*. URL: https://www.informit.com/articles/article.aspx?p=1833567 (visited on 05/31/2022).

[62] Robert T. Futrell, Donald F. Shafer, and Linda Isabell Shafer. *Quality Software Project Management, Two Volume Set*. 1st. Pearson., Jan. 2002. ISBN: 978-0-13-091297-8. URL: https://www.informit.com/store/quality-software-project-management-two-volume-set-9780130912978?w_ptgrevartcl=How+You+Can+Benefit+from+Software+Configuration+Management_26858 (visited on 07/03/2022).

[63] Jackie Garcia. *Build Automation 101: Your Guide to an Automated Build Process*. en. URL: https://www.perforce.com/blog/vcs/build-automation (visited on 05/31/2022).

[64] *Getting Started - TeamCity 6.5.x Documentation - Confluence*. URL: https://confluence.jetbrains.com/display/tcd65/getting+started#GettingStarted-TeamCityArchitecture (visited on 06/23/2022).

[65] *Getting Started With GIT - Studytonight*. en. URL: https://www.studytonight.com/git-guide/getting-started-with-git (visited on 05/30/2022).

[66] *Git - SCM*. URL: https://git-scm.com/book/en/v2 (visited on 05/02/2022).

[67] *Git | TeamCity On-Premises*. en-US. URL: https://www.jetbrains.com/help/teamcity/git.html (visited on 06/18/2022).

[68] *Git Branching Strategies vs. Trunk-Based Development - LaunchDarkly*. en. URL: https://launchdarkly.com/blog/git-branching-strategies-vs-trunk-based-development/ (visited on 05/31/2022).

[69] *Git Branching Strategies: GitFlow, Github Flow, Trunk Based...* en-US. Mar. 2022. URL: https://www.flagship.io/git-branching-strategies/ (visited on 05/30/2022).

[70] *Git vs. Perforce: How to Choose (and When to Use Both)*. en. URL: https://www.perforce.com/blog/vcs/git-vs-perforce-how-choose-and-when-use-both (visited on 05/08/2022).

[71] *GitLab vs GitHub: Explore Their Major Differences and Similarities*. en-US. Mar. 2021. URL: https://kinsta.com/blog/gitlab-vs-github/ (visited on 06/19/2022).

[72] *Glossary | TeamCity On-Premises*. en-US. URL: https://www.jetbrains.com/help/teamcity/glossary.html (visited on 07/02/2022).

[73] Thomas Hamilton. *Functional Testing Vs Non-Functional Testing: What's the Difference?* en-US. Mar. 2020. URL: https://www.guru99.com/functional-testing-vs-non-functional-testing.html (visited on 05/31/2022).

[74] Thomas Hamilton. *What is Jenkins? Why Use Continuous Integration (CI) Tool?* en-US. Jan. 2020. URL: https://www.guru99.com/jenkin-continuous-integration.html (visited on 05/31/2022).

[75] Thomas Hamilton. *What is Scalability Testing? Learn with Example*. en-US. Feb. 2020. URL: https://www.guru99.com/scalability-testing.html (visited on 05/31/2022).

[76] Thomas Hamilton. *What is Software Testing? Definition, Basics & Types in Software Engineering*. en-US. Jan. 2020. URL: https://www.guru99.com/software-testing-introduction-importance.html (visited on 05/31/2022).

[77] Brad Hart. *The Best Branching Strategies For High-Velocity Development*. en. URL: https://www.perforce.com/blog/vcs/best-branching-strategies-high-velocity-development (visited on 05/03/2022).

[78] Brad Hart. *Trunk Based Development or Feature Driven Development — What's Better For Your Team?* en. URL: https://www.perforce.com/blog/vcs/trunk-based-development-or-feature-driven-development (visited on 05/30/2022).

[79] Ansible Hat Red. *Ansible is Simple IT Automation*. en-us. URL: https://www.ansible.com (visited on 07/08/2022).

[80] *Helix Core Visual Client (P4V) Guide (2022.1)*. URL: https://www.perforce.com/manuals/p4v/Content/P4V/Home-p4v.html (visited on 05/08/2022).

[81] *How Can Enterprises Enhance Software Development by Test Automation Services?* en-US. Jan. 2022. URL: https://www.matellio.com/blog/how-can-enterprises-enhance-software-development-by-test-automation-services/ (visited on 07/01/2022).

[82] *How To Build CI/CD Pipeline With TeamCity For Selenium Test Automation*. en-US. June 2021. URL: https://www.lambdatest.com/blog/ci-cd-pipeline-with-teamcity-for-selenium-test-automation/ (visited on 06/23/2022).

[83] *How To Maintain an Open-Source Library*. en. URL: https://segment.com/blog/tips-for-maintaining-an-open-source-library/ (visited on 05/31/2022).

[84] *How to set up Git branch protection rules.* en-US. July 2021. URL: https://spectralops.io/blog/how-to-set-up-git-branch-protection-rules/ (visited on 06/30/2022).

[85] *How to Use Git Branches & Buddy to Organize Project Code - SitePoint.* en. July 2019. URL: https://www.sitepoint.com/use-git-branches-buddy/ (visited on 05/31/2022).

[86] Jez Humble. *What is Continuous Delivery? - Continuous Delivery.* URL: https://continuousdelivery.com/ (visited on 04/20/2022).

[87] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* 1st. Addison-Wesley Professional. Part of the Addison-Wesley Signature Series (Fowler) series., July 2010. ISBN: 978-0-321-60191-9. URL: https://www.informit.com/store/continuous-delivery-reliable-software-releases-through-9780321601919?ranMID=24808 (visited on 04/15/2022).

[88] ILZE. *4 Reasons Why Software Testing is Important.* en-US. July 2018. URL: https://www.testdevlab.com/blog/2018/07/03/importance-of-software-testing/ (visited on 05/31/2022).

[89] Sonatype Inc. *Software Supply Chain Security - DevSecOps Governance | Sonatype.* en-us. URL: https://www.sonatype.com (visited on 05/31/2022).

[90] *Install.* en. URL: https://dvc.org/doc/install (visited on 06/22/2022).

[91] *Install — conan 1.46.2 documentation.* URL: https://docs.conan.io/en/1.46/installation.html (visited on 06/26/2022).

[92] *Install and Start TeamCity Server | TeamCity On-Premises.* en-US. URL: https://www.jetbrains.com/help/teamcity/install-and-start-teamcity-server.html (visited on 06/23/2022).

[93] *Installing Artifactory - JFrog - JFrog Documentation.* URL: https://www.jfrog.com/confluence/display/JFROG/Installing+Artifactory (visited on 06/22/2022).

[94] *Jenkins.* en. URL: https://www.jenkins.io/ (visited on 05/31/2022).

[95] *JFrog Artifactory.* en-US. URL: https://jfrog.com/solution-sheet/jfrog-artifactory/ (visited on 05/31/2022).

[96] *JFrog Artifactory | ASERVO.* URL: https://www.aservo.com/en/tools/configuration-management/jfrog-artifactory-en (visited on 05/31/2022).

[97] *JFrog Artifactory CE | Your Conan C/C++ package manager repository.* en-US. Mar. 2018. URL: https://jfrog.com/blog/announcing-jfrog-artifactory-community-edition-c-c/ (visited on 06/26/2022).

[98] *JFrog Artifactory Vs. Sonatype Nexus.* en-US. Feb. 2021. URL: https://jfrog.com/blog/artifactory-vs-nexus-integration-matrix/ (visited on 05/31/2022).

[99] Eva Johnson. *CVCS & DVCS: The Needs That Version Control Systems Serve.* en-us. July 2014. URL: https://content.intland.com/blog/sdlc/the-needs-that-version-control-systems-serve (visited on 05/06/2022).

[100] Eva Johnson. *Pros and Cons: Is Git Better Than Mercurial?* en-us. Jan. 2015. URL: https://content.intland.com/blog/sdlc/why-is-git-better-than-mercurial (visited on 05/02/2022).

[101] Kayly Lange. *Release Management in DevOps.* en-US. URL: https://www.bmc.com/blogs/devops-release-management/ (visited on 05/31/2022).

[102] *Legacy System Modernization Approaches | A Rackspace Guide.* en. Dec. 2020. URL: https://www.rackspace.com/blog/brief-guide-legacy-system-modernization (visited on 07/07/2022).

[103] *Legacy System Modernization: How to Transform the Enterprise for Digital Future.* en-US. URL: https://www.altexsoft.com/whitepapers/legacy-system-modernization-how-to-transform-the-enterprise-for-digital-future/ (visited on 07/07/2022).

[104] Rafal Leszko. *Continuous delivery with Docker and Jenkins: delivering software at scale.* en. First published. Birmingham Mumbai: Packt Publishing, 2017. ISBN: 978-1-78712-523-0.

[105] Rafal Leszko. *Continuous delivery with Docker and Jenkins: delivering software at scale.* en. First published. Birmingham Mumbai: Packt Publishing, 2017. ISBN: 978-1-78712-523-0.

[106] Matt Mackall. "Towards a Better SCM: Revlog and Mercurial". en. In: (), p. 8.

[107] *Managing releases in a repository.* en. URL: https://ghdocs-prod.azurewebsites.net/en/repositories/releasing-projects-on-github/managing-releases-in-a-repository (visited on 07/03/2022).

[108] *Managing the Open Source Dependency.* URL: https://www.computer.org/csdl/magazine/co/2020/02/08996108/1hmvEWqZ8Vq (visited on 05/31/2022).

[109] Matthew Martin. *Software Configuration Management in Software Engineering.* en-US. Section: Business Analyst. Mar. 2020. URL: https://www.guru99.com/software-configuration-management-tutorial.html (visited on 04/24/2022).

[110] Joseph Mathenge. *Deploying vs Releasing Software: What's The Difference?* en-US. URL: https://www.bmc.com/blogs/software-deployment-vs-release/ (visited on 05/31/2022).

[111] *Mercurial SCM.* URL: https://www.mercurial-scm.org/ (visited on 05/02/2022).

[112] *Mercurial vs. Git: why Mercurial?* en-US. Feb. 2012. URL: https://www.atlassian.com/blog/software-teams/mercurial-vs-git-why-mercurial (visited on 05/06/2022).

[113] *Modernizing Legacy Systems: A complete guide.* en-US. URL: https://headspring.com/modernizing-legacy-systems/ (visited on 05/30/2022).

[114] Tomomi Uchida Molin. *Why is software testing so important?* en-gb. URL: https://www.theiceway.com/blog/why-is-software-testing-so-important (visited on 05/31/2022).

[115] *MSBuild (Visual Studio) — conan 1.49.0 documentation.* URL: https://docs.conan.io/en/latest/integrations/build_system/msbuild.html#msbuild-integration (visited on 06/27/2022).

[116] Quang Nguyen. *Git-Flow vs GitHub-Flow.* en. Sept. 2021. URL: https://quangnguyennd.medium.com/git-flow-vs-github-flow-620c922b2cbd (visited on 05/31/2022).

[117] *Nightly / Daily build.* en. Apr. 2022. URL: https://iq.opengenus.org/nightly-build/ (visited on 06/12/2022).

[118] Bryan O'Sullivan. "Distributed revision control with Mercurial". In: (), p. 201.

[119] *Package management basics - Learn web development | MDN.* en-US. URL: https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Understanding_client-side_tools/Package_management (visited on 06/26/2022).

[120] David Pearce. *5 Risks of Legacy Technology.* en-us. URL: https://www.rutter-net.com/blog/5-risks-of-legacy-technology (visited on 05/30/2022).

[121] *Perforce vs Git | Choosing the Right Version Control Systems.* en-US. June 2020. URL: https://www.educba.com/perforce-vs-git/ (visited on 05/08/2022).

[122] DevOps Professional. *Continuous Delivery Best Practices.* en-US. June 2019. URL: https://www.devonblog.com/continuous-delivery/continuous-delivery-best-practices/ (visited on 05/31/2022).

[123] *Pull Requests | TeamCity On-Premises.* en-US. URL: `https://www.jetbrains.com/help/teamcity/pull-requests.html` (visited on 07/02/2022).

[124] *Salt Project – Salt Open Source.* en-US. URL: `https://saltproject.io/` (visited on 07/08/2022).

[125] Brent Schiestl. *Code Branching Definition — What Is a Branch?* en. URL: `https://www.perforce.com/blog/vcs/branching-definition-what-branch` (visited on 05/03/2022).

[126] Nishant Sharma. *How to set up a build pipeline on JetBrains TeamCity?* en. Dec. 2021. URL: `https://medium.com/testvagrant/how-to-set-up-a-build-pipeline-on-jetbrains-teamcity-41a1b0a67d76` (visited on 05/31/2022).

[127] *Software Configuration Management - an overview | ScienceDirect Topics.* URL: `https://www.sciencedirect.com/topics/computer-science/software-configuration-management` (visited on 04/24/2022).

[128] *Sonatype Nexus Repository Managers | ASERVO.* URL: `https://www.aservo.com/en/tools/configuration-management/sonatype-nexus-repository-managers-en` (visited on 05/31/2022).

[129] *Stack Overflow Developer Survey 2020.* URL: `https://insights.stackoverflow.com/survey/2020/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2020` (visited on 06/19/2022).

[130] *SVN Tutorial - Javatpoint.* en. URL: `https://www.javatpoint.com/svn` (visited on 05/02/2022).

[131] *SVN vs Git - Javatpoint.* en. URL: `https://www.javatpoint.com/svn-vs-git` (visited on 05/02/2022).

[132] Apica Systems. *Automated vs Manual Testing: Which Should You Use, and When?* en-US. July 2017. URL: `https://www.apica.io/difference-between-automated-manual-testing/` (visited on 05/31/2022).

[133] *Test Automation Benefits: 12 Reasons to Automate in 2020.* en-US. Nov. 2019. URL: `https://www.testim.io/blog/test-automation-benefits/` (visited on 06/05/2022).

[134] *Testing Legacy Software without Documents | Brainbox.* en-US. May 2020. URL: `https://www.brainbox.consulting/blogs-news/software-testing-blog/testing-legacy-software-without-documents/` (visited on 05/30/2022).

[135] TestProject. *What Are The Benefits of Having Nightly Builds*. en-US. Oct. 2019. URL: https://blog.testproject.io/2019/10/14/what-are-the-benefits-of-having-nightly-builds/ (visited on 06/12/2022).

[136] *The benefits of a distributed version control system*. en. URL: https://about.gitlab.com/topics/version-control/benefits-distributed-version-control-system/ (visited on 05/02/2022).

[137] *The Only Guide to Dark Launching You'll Ever Need - LaunchDarkly*. en. URL: https://launchdarkly.com/blog/guide-to-dark-launching/ (visited on 05/31/2022).

[138] *The Software Configuration Management Process: 5 Steps*. en-US. May 2021. URL: https://theqalead.com/topics/software-configuration-management-process/ (visited on 05/02/2022).

[139] *The Ultimate Guide to Performance Testing and Software Testing: Testing Types, Performance Testing Steps, Best Practices, and More*. en-US. Apr. 2021. URL: https://stackify.com/ultimate-guide-performance-testing-and-software-testing/ (visited on 05/31/2022).

[140] *Top 10 BEST Build Automation Tools To Speed Up Deployment Process*. en-US. URL: https://www.softwaretestinghelp.com/best-build-automation-software-tools/ (visited on 05/31/2022).

[141] *Top 10 Build Automation Tools*. en-US. Apr. 2021. URL: https://lightrun.com/dev-tools/top-10-build-automation-tools/ (visited on 05/31/2022).

[142] *Types of Software Testing: Different Testing Types with Details*. en-US. Aug. 2007. URL: https://www.softwaretestinghelp.com/types-of-software-testing/ (visited on 05/31/2022).

[143] *Types of Software Testing: Different Testing Types with Details*. en. URL: https://hackr.io/blog/types-of-software-testing (visited on 05/31/2022).

[144] *Understanding Subversion (SVN) - Here's Why Your Hosting Choice Is Important*. en-US. URL: https://digital.com/best-web-hosting/subversion/ (visited on 05/05/2022).

[145] *Understanding the CI/CD Pipeline: What It Is, Why It Matters*. en-US. Mar. 2019. URL: https://www.plutora.com/blog/understanding-ci-cd-pipeline (visited on 04/24/2022).

[146] *UnderstandingMercurial - Mercurial*. URL: https://www.mercurial-scm.org/wiki/UnderstandingMercurial (visited on 05/06/2022).

[147] *VCS Root | TeamCity On-Premises*. en-US. URL: https://www.jetbrains.com/help/teamcity/vcs-root.html (visited on 07/01/2022).

[148] Ravi Verma. *Centralized vs Distributed Version Control Systems [CVCS vs DVCS]*. en-us. May 2014. URL: https://devopsbuzz.com/centralized-vs-distributed-version-control-systems/ (visited on 05/30/2022).

[149] *Version Control Systems Popularity in 2016*. URL: https://rhodecode.com/insights/version-control-systems-2016 (visited on 06/18/2022).

[150] *Versioning Strategy*. en-US. May 2018. URL: https://flowcanon.com/software/versioning-strategy/ (visited on 05/31/2022).

[151] *Visual Studio: IDE und Code-Editor für Softwareentwickler und -teams*. de-DE. URL: https://visualstudio.microsoft.com/ (visited on 06/26/2022).

[152] Stephen Watts. *What is the Canary Deployment & Release Process?* en-US. URL: https://www.bmc.com/blogs/canary-deployment-release/ (visited on 05/31/2022).

[153] *What Are Automation Testing Tools? 9 Types & Examples*. en-US. Apr. 2021. URL: https://theqalead.com/topics/what-are-automation-testing-tools/ (visited on 05/31/2022).

[154] *What Are Some of the Advantages of Continuous Delivery? What Are the Common Barriers?* en. URL: https://www.zend.com/blog/continuous-delivery-benefits-and-barriers (visited on 04/24/2022).

[155] *What is a centralized version control system*. en. URL: https://about.gitlab.com/topics/version-control/what-is-centralized-version-control-system/ (visited on 05/02/2022).

[156] *What is a Legacy System? - Talend*. en. URL: https://www.talend.com/resources/what-is-legacy-system/ (visited on 04/24/2022).

[157] *What is a package manager?* URL: https://www.debian.org/doc/manuals/aptitude/pr01s02.en.html (visited on 06/26/2022).

[158] *What is a software release?* en. URL: https://www.techtarget.com/searchsoftwarequality/definition/release (visited on 05/31/2022).

[159] *What is Automated Testing and How Does it Work?* en. URL: https://www.techtarget.com/searchsoftwarequality/definition/automated-software-testing (visited on 06/05/2022).

[160] *What is blue green deployment?* en. URL: https://www.redhat.com/en/topics/devops/what-is-blue-green-deployment (visited on 05/31/2022).

[161] *What is Build Automation?* en-US. URL: https://flexagon.com/what-is-build-automation/ (visited on 05/31/2022).

[162] *What Is CI/CD and How Does It Work? | Synopsys*. en. URL: https://www.synopsys.com/glossary/what-is-cicd.html (visited on 07/07/2022).

[163] *What is CI/CD Pipeline? | Katalon Platform*. en. URL: https://katalon.com/resources-center/blog/ci-cd-pipeline (visited on 07/07/2022).

[164] *What Is CI/CD? Continuous Integration & Continuous Delivery Explained*. en-US. URL: https://www.bmc.com/blogs/what-is-ci-cd/ (visited on 06/27/2022).

[165] *What is CI/CD/CD? The Differences & Benefits & DevOps Application*. en-CA. Jan. 2021. URL: https://www.indellient.com/blog/whats-the-difference-between-continuous-integration-continuous-delivery-and-continuous-deployment/ (visited on 04/24/2022).

[166] *What is Continuous Testing and How Does it Work? | Synopsys*. en. URL: https://www.synopsys.com/glossary/what-is-continuous-testing.html (visited on 06/09/2022).

[167] *What is Continuous Testing?* en-US. URL: https://www.tricentis.com/products/what-is-continuous-testing/ (visited on 06/09/2022).

[168] *What is Jenkins? | Jenkins For Continuous Integration*. en-US. Nov. 2016. URL: https://www.edureka.co/blog/what-is-jenkins/ (visited on 05/31/2022).

[169] *What is Performance Testing?* en. URL: https://www.techtarget.com/searchsoftwarequality/definition/performance-testing (visited on 05/31/2022).

[170] *What is Release Management? (All That You Need To Know)*. en-US. URL: https://www.plutora.com/software-release-management/what-is-release-management (visited on 05/31/2022).

[171] *What is Scalability Testing? How to Test the Scalability of an Application*. en-US. URL: https://www.softwaretestinghelp.com/what-is-scalability-testing/ (visited on 05/31/2022).

[172] *What is Software Testing and How Does it Work? | IBM*. en-us. URL: https://www.ibm.com/topics/software-testing (visited on 05/31/2022).

[173] *What Is SVN (Subversion)?* en. URL: https://www.perforce.com/blog/vcs/what-svn (visited on 05/02/2022).

[174] *What Is The Benefit of Test Automation and Why Should We Do It?* URL: https://smartbear.com/solutions/automated-testing/ (visited on 06/05/2022).

[175] *What is versioning and how does it work?* en. URL: https://www.techtarget.com/searchsoftwarequality/definition/versioning (visited on 05/31/2022).

[176] *Why Use Version Control?* en. URL: https://www.git-tower.com/learn/git/ebook/en/desktop-gui/basics/why-use-version-control (visited on 05/30/2022).

[177] Shanika Wickramasinghe. *DevOps Branching Strategies Explained.* en-US. URL: https://www.bmc.com/blogs/devops-branching-strategies/ (visited on 05/03/2022).