U.P.B. Sci. Bull., Series C, Vol. 87, Iss. 2, 2025 ISSN 2286-3540

ANALYSIS OF VERSION CONTROL IN CONTINUOUS INTEGRATION AND DELIVERY

Oana-Anastasia MINCIU[1*], Beatrice-Nicoleta CHIRIAC[2], Florin Daniel ANTON[3], Anca Daniela IONITA[4]

The adoption of version control has been essential to software development dynamics, which are shifting considerably towards increasing delivery quality and speed. Continuous integration and continuous deployment / delivery require further improvements in this regard. This aim of this paper is to analyse the characteristics of version control and continuous practices, and to identify the key elements of interaction in the process of code integration and delivery. Considering commits, branches, artefacts, triggers, pipeline and other automations, the connecting points were extracted to portray differences between practices and their intricate collaboration and dependency with version control.

Keywords: software configuration management, version control, continuous integration, continuous deployment/delivery

1. Introduction

Software artefacts are subject to a multitude of changes during development, and programmers end up having multiple versions of the same file because of constant changes. Saving intermediate versions of a file with different names, or in different paths, is repetitive work and a very poor way of managing intermediate versions relying mostly on human memory. Thus, this versioning strategy is error prone and, moreover, only addresses individual work management. Naturally, for managing work originating from multiple people who contribute to the same files, has led to the creation of better and objective strategies, which take user memory and self-defined logic out of the equation to obtain structured and strictly defined mechanisms known as version control systems. A typical approach is to have a central server embodying a single source of truth of file versions, which

[1] Corresponding author

[1] PhD student, Automation and Industrial Informatics Department, National University of Science and Technology POLITEHNICA Bucharest, Romania, e-mail: oana.anastasia.minciu@gmail.com

[2] PhD student, Automation and Industrial Informatics Department, National University of Science and Technology POLITEHNICA Bucharest, Romania, e-mail: chiriacbeatrice96@gmail.com

[3] Reader, Automation and Industrial Informatics Department, National University of Science and Technology POLITEHNICA Bucharest, Romania, e-mail: florin.anton@upb.ro

[4] Prof., Automation and Industrial Informatics Department, National University of Science and Technology POLITEHNICA Bucharest, Romania, e-mail: anca.ionita@upb.ro

resulted in central version control systems. Another approach is to have multiple sites with copies in a peer-to-peer setup which encompass distributed version control systems. Central and distributed version control systems are compared to show that central ones are suited for projects which allow contributions from few users from a single site, while distributed ones can accommodate multiple users from small or big teams, located in multiple sites [1].

Version control as part of Software Configuration Management (SCM) has been treated as a development support discipline [2] with the role to help in coordinating software product changes. Conradi and Westfechtel provide an overview of how version models were implemented in the 90^s [3] and mention that many existing systems were file-based, like ClearCase, a versioning system with a long history, which still uses file-based versioning. Version control systems are essential over the lifetime of any software project for allowing tracking of simultaneous work from multiple developers, therefore they have become a necessity in software development with their applications and use cases increasing. The purpose of this paper is to analyse how version control expands beyond regular source code management. The first generation of version control systems was focused on allowing collaboration with separate tracking, only for one developer at once, whereas the next generation has used the concept of a central repository with remote access, where multiple developers could contribute at the same time [4]. Moreover, the present generation is focused on a central repository and multiple local copies owned by developers. Due to the necessity to accelerate the software development life cycle, Continuous Integration (CI) and Continuous Delivery / Deployment (CD) have become more and more used [5]. Using CI tools accelerates the software release process and helps avoiding introduction of faulty code helping with detection and prevention [6].

This paper is divided in five sections. After this introduction, the second section describes background information regarding version control. The third section exposes the aspects of security control and different types of version control regarding both code and artefacts. Then, it presents the interdependency between continuous practices and version control with a comparative analysis. The fourth section starts from the four criteria identified in the previous analysis, which show differences between CI and CD, and it presents a practical demonstration of them. The experiments follow a GitLab pipeline developed by installing and configuring a GitLab server and runner, defining the requirements for a project, and then designing and writing the implementation. Afterwards, the process was executed in multiple scenarios meant to showcase and discuss the results for the chosen criteria. The fifth section contains the conclusions.

2. Background

In comparison with traditional software development, a faster integration, realized through DevOps introduces a series of challenges, which also concern versioning [7]. One must maintain control over various elements, from projects, tests and integration code to documentation and build artefacts. Whereas version control is sometimes interchanged with source code management, DevOps practices have accelerated the extension of version control beyond just code. Continuous practices have introduced the importance of managing build artefacts, but moreover, version control actions have become triggers for these practices. Paez also covers the importance of defining versioning strategies for all artefacts used in DevOps [8], whether they are configuration text files, or source code files. Practices like infrastructure as code and continuous delivery lead to creating new artefacts that are to be versioned, therefore artefact versioning has become an essential part of managing DevOps practices and will continue to do so.

2.1. Version control in source code management

Source code versioning is the traditional use case in development, mainly to address coordinating user changes and multiple versions. Currently, most open-source projects adopt distributed versioning systems, because central ones come with the risk that, if the server is unavailable, developers cannot fetch the history and work on the correct code versions [1]. Nevertheless, both types of systems are in use, with central systems like Concurrent Versions System (CVS), Perforce, Subversion, or ClearCase, and distributed ones like Git, Mercurial, or Bazaar. Regardless of their type, they organize code in databases named repositories and portray version trees where individual nodes are distinct versions, which represent the submitted changes to the repository at one point in time. Each commit is a version, but versions meant for release are usually marked with a tag. Version trees can have multiple branches for separated lines of development, but at least one central branch is meant to always store stable code, and it is usually the target of CI practices, and subject to additional protections. All branches must be subject to the integration process, but the final goal is to keep stable code in the same state, therefore protected branches can have further restrictions.

2.2. Artefact version control

During software development, one project produces a large variety of artefacts beyond source code. In [8], Paez mentions that within the development process there are both source code artefacts and documentation artefacts, the former refers to application or infrastructure code, scripts, configuration files, binaries, while the latter is represented by requirements or diagrams. In CI/CD an artefact is any file generated during the process with some being intermediary with a short lifespan and others meant for long-term storage. For the first category, no version control is needed, as the data can simply be discarded afterwards, but for the second one, some kind of version control is required. A few examples of files meant for storage in version control are metadata, version specific identifier elements, container images, executables.

Code in its essence is merely text, regardless of language specific syntax, while other elements like metadata can also refer to text files, but most artefacts do not share the same format. Jones et al. emphasize the difference in complexity between text-based code files and other types, like for example CAD (Computer Aided Design) files, containing commands trees for three-dimensional models [9]. Thus, tracking artefact changes is different than tracking code changes; for code, small text changes known as deltas can be monitored, while for artefacts, any change results in an entirely new instance. Version control meant for code can handle binary files as well, but they are rather handled as singular units. Thus, two important challenges of artefact versioning are that binaries can be significantly bigger than code and that some kinds of artefacts can require different types of storing regarding structure. Regardless of size, any modifications in a binary lead to that file being completely replaced, as opposed to just a few lines of code.

Considering version control systems work with code snapshots, whenever a new change is transferred, the bigger the files are, the slower the data transfer will be between users and servers. For example, the JFrog product Artifactory, a tool for artefact storage, uses a file store and database system to pair each binary with its identifying checksum along with generic metadata like artefact names, size, creation dates, but also with specific package metadata [10]. This is because each type of artefact has different elements, generic files are single units, but other packages may be collections of different types of files, while container images are stored in layers**.** Consequently, although version control systems are great options for storing code, configuration, small files, more complex artefacts such as binaries should be stored separately, using tools tailored for larger artefacts with essential extensions that allow classifying and storing accordingly based on type.

3. Analysis of specific aspects in version control

3.1. Security for version control

While control version systems save change history of different information entities, the security component is indispensable for this process. Thus, cybersecurity models should be applied over the stored information, for maintaining the integrity, confidentiality and the availability of the data. These security measures are taken based on the specific of the versioning tool as well as based on the importance of the information kept inside the repository. Complex repositories face with multiple users and different methods of authentication based on the type of the version tool [11].

Centralized version control tool uses a client-server configuration. The central server hosts the repository and manages the users' access to it. Files and their versions that are stored in this central database can be accessed using a user ID and a set of access rights that are given by an administrator of the system. For example, IBM Rational ClearCase has a remote repository named Versioned Object Base (VOB) and each user owns a workspace that is associated with a view. Inside VOB, each file has its own version tree, and the users can have a dynamic view which permits them to visualize the modifications made into repository in real-time or a snapshot view which implies a local copy of the VOB. As a superior level of security, the VOB and VOB objects are using Access Control Lists (ACL). For each file hosted on the central server granular rules of access are given by the administrator depending on each view [12]. These kinds of ACLs contain rules applying to resources and a relevant example is that every user that owns a view in a ClearCase repo can visualize the modifications made by other users on their private views, but they are not able to make changes on another user's workspace.

On the contrary, distributed systems apply the advantages of the secured communication protocols like SSH or HTTPS used for the remote connections and the model of copy-modify-merge for preventing conflicts between multiple users. All transferred data is encrypted, and the commit process maintains its transactional characteristic. The control access to the remote repository via SSH is based on the allocation of private/public key pair to each user for authentication. The public key is shared and used for handshaking, while the private one is never shared and unique for each user. If the encryption-decryption algorithm is not working properly, the connection is stopped [13]. HTTPS approaches this aspect differently by using a password-based authentication. During HTTPS handshaking the server provides a list of digital certificates together with the public key to the communication partner. The client verifies the validity of the certificate and if it is valid, the client exchanges its key. After this step, the connection is established. For a higher level of security this kind of traffic can be monitored and analysed using security tools like firewall, IDS [14].

3.2. Version control in continuous practices

Continuous Integration is a concept part of the DevOps chain of tools and practices, meant to accelerate code integration between developers, while ensuring high software quality within several code validation and verification steps. Software development methodologies focus on the evolvement of procedures adapting to modern times [15]. Continuous Delivery extends this to ensure rapid package delivery, either to a central artefact server, or directly to a client. A simple description of both would be that Continuous Integration consists of the steps sequence beginning from a published code change of a project and ending with the deployment of the built artefact to an artefact server, while Continuous Delivery is the same process, but executed on code ready to be officially delivered to clients. Their implementations vary, but the process is called a pipeline modelled upon project requirements. New code is submitted through commits to a version control system, starting continuous integration, which consists of the following steps: build – to obtain build artefacts such as executables, analysis - static code analysis and test execution, integration result - gather the analysis results and provide a decision of whether the code fulfils the requirements to be integrated in a stable branch in version control.

CI/CD is considered a feedback loop, starting with submitted code changes and ending with the acceptance or rejection of these changes. Upon acceptance, the code is successfully integrated into the main branch in development, and executables are allowed to be uploaded to an artefact server, to be delivered either to functional testing teams or clients, if the process is for integration or delivery. The flow of CI/CD flow implies that the version control system is part of the starting point, the end of continuous integration and a part of continuous delivery. The commit in version control is the trigger of the process, so the start is associated with a version control action, while the end of continuous integration is the registration of the result in version control. For the delivery part, some additional steps are considered, from introducing release commits back to the version control system to simply deploying a suite of artefacts to an artefact server. The delivery pipeline extends the integration one to provide a final report to the code version control server to notify the process success or failure.

3.3. Comparison of version control in CI vs. CD

Version control and CI/CD are linked by actions that may be either automated, or manual operations. Triggers, result publishing and artefact deployment are automated, but actual merging of code includes manual tasks. This is completely normal as besides code and test analysis, projects can also have a manual review in place, after a successful pipeline to assess errors unrelated to code compilation, execution and testing. A continuous integration process can only do so much as to assert whether the code is ready to be integrated, but it cannot establish whether the code respects functionality requirements. Decisions to perform code merging rely on developers in the end, but CI/CD interacts closely with version control to ensure the completion of code merging and release.

For this analysis, version control has been split between source code management also known as Version Control System (VCS) and Artefact Management (AM). Considering how CI/CD connects with version control, the following criteria have been chosen to compare them: interaction with VCS elements, with artefacts, automation in relation to version control. The VCS elements studied include commits, pull/merge requests, branches and communication elements for notification. For AM, the relation to both temporary and long-lived artefacts were analysed. Automation was also a subject of interest as it is at the core of DevOps practices, especially for CI/CD pipelines. Table 1 presents the comparative analysis between Continuous Integration and Continuous Delivery regarding how they interact directly with but also depend on version control for code and artefacts. No specific automation server, version control system or artefact storage solution were considered in building the analysis.

Table 1

Comparative Analysis of Version control in CI vs CD

Criterion	CI	CD	Explanation
C1: VCS trigger	Yes	Yes	Code commits and merge or pull requests are triggers for the process.
C2: VCS commit and tags	No	Yes	CI pipelines cannot push commits or tags to VCS, because this is not part of their functions. CD pipelines can push one or more commits during the process of a release, either release commits or preparation for the next development version commits.
C3: VCS notification	Yes	Yes	A build result is returned to the VCS server to alert developers on the success or failure of the process.
C4: VCS branch specific operations	No	Yes	CI pipelines do not depend on the type of branch as all branches rely on the same base process for code integration. CD pipelines depend on branch type: release branches produce official versions; some branches produce intermediate versions; others cannot deliver any versions.
C5: Temporary build artefacts	Yes	Yes	Intermediate artefacts are generated, but are meant to be discarded upon the process completion and they include intermediate analysis results and package dependencies
C6: Deployment artefacts	No	Yes	CI produces artefacts (executables or packaged elements) that can be deployed, but it does not upload artefacts to a storage server because it only deals with code. CD produces and uploads artefacts to a storage server: both intermediate versions and official release artefacts.
C7: Test reports artefacts	Yes	Yes	Both produce test reports or logs; these artefacts are also temporary but have a longer lifespan than temporary build artefacts. Their contents are displayed in automation servers for a time, but not uploaded to an artefact server
C8: Complete automation	No	Yes	The CI pipeline process is automated, but it relies on a final peer review step in SM. CD is completely automated as it runs on code already reviewed.

Version control has also expanded beyond usage for software development code, with terms like GitOps being introduced. The combination between Infrastructure as Code and Continuous principles lead to the concept of a GitOps pipeline with Git as the single source of truth for DevOps operations [16]. However, regardless of the technology name, they all share common or close definitions for elements, like commits, branches, repositories, tags, or merge requests and pull requests to ensure seamless integration with software development. Thus, CI/CD pipelines are modelled on these generic elements and only their implementation is technology specific.

4. Experiments of version control in CI/CD

4.1. Method

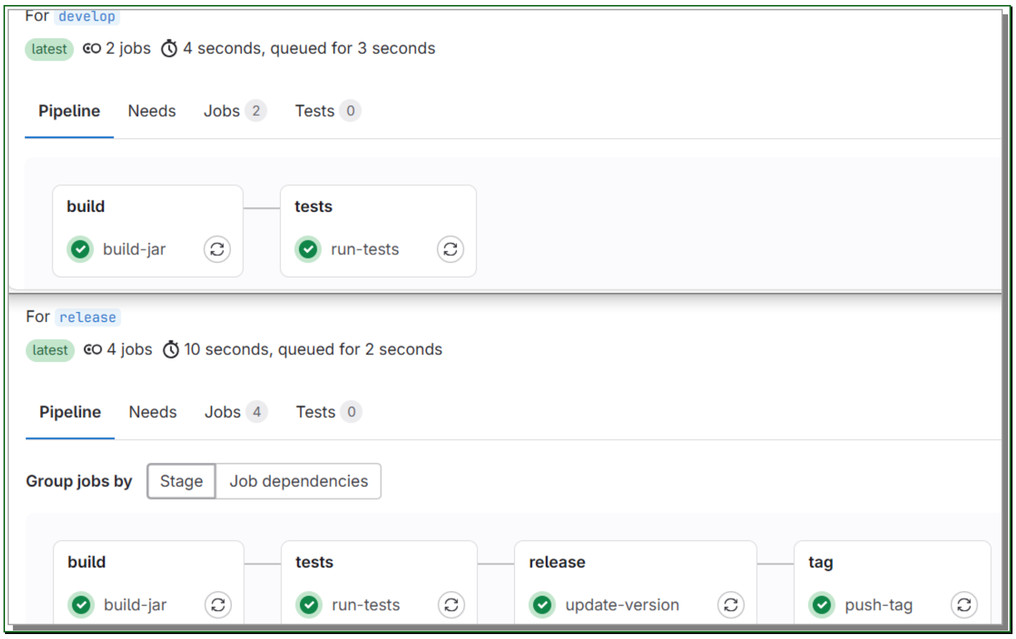
A combination of separate free tier git, automation and artefact servers was considered with GitLab or Bitbucket, and Jenkins plus Nexus, but no combination offers the seamless integration GitLab provides between the three essential version control pillars in CI/CD in one server: code, automation and artefact storage. GitLab has the major advantage of delivering all three and mitigating network traffic delay and deployment complexity. Thus, the GitLab platform was installed and configured using a Docker container with the 17.3.6-ee.0 version on a virtual machine with Ubuntu 22.04 and a similar machine for the GitLab runner. The target project is a personal Java application using Java 17 and Maven 3.6.3. The pipeline was developed with five stages to build, test, deploy the code and ensure release and tag pushing. Conditions are applied to control how and when the stages are executed based on the project requirements: on the experiment the main and develop branches are protected and require merge requests for changes, the main branch is allowed to deploy artefacts, while the develop branch runs only the integration steps. These actions conclude the environment installation and the continuous processes implementation.

4.2. Results and discussion

Inside the deployed environment the processes were executed to further study the differences of behaviours and results between CI and CD. Thus, from the previous chapter the following five criteria were explored: C2, C4, C6 and C8. The integration process includes the build and test stages, while the delivery process includes the update-versions, push-tag and deploy-jar stages. The delivery is split between the release and main branches because the project only allows artifact deployment from branch main and version changes from branch release. The automated triggers are also implemented by the authors with specific conditions depending on branches, pipeline parameters and already existing tags.

The release process execution shows how *criterion C2* can be observed within the pipeline behaviour. The release branch is configured to allow upgrading the project to a stable version inside the project configuration. Also, this change is committed and tagged in version control only from this branch. Fig. 1 shows the implemented pipeline execution and the results for the release branch which has the specific operations of updating the version and committing and pushing the tag to the repository, while another branch, like develop, has its own set of stages.

Therefore, the *criterion C4* can also be observed here, but also in the experiment from Fig. 2, which shows the pipeline execution for a tag.


For develop
latest 2 jobs 4 seconds, queued for 3 seconds
Pipeline Needs Jobs 2 Tests 0
build: build-jar
tests: run-tests

For release
latest 4 jobs 10 seconds, queued for 2 seconds
Pipeline Needs Jobs 4 Tests 0
Group jobs by Stage Job dependencies
build: build-jar
tests: run-tests
release: update-version
tag: push-tag

Fig. 1. Commit actions and branch specific operations in CI vs CD

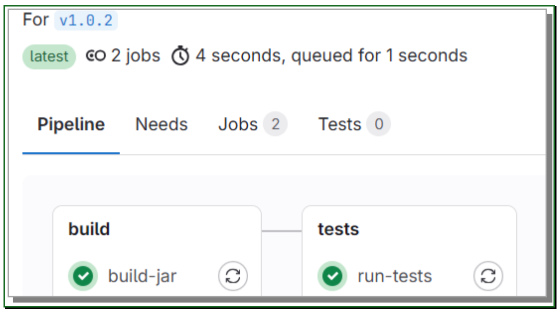
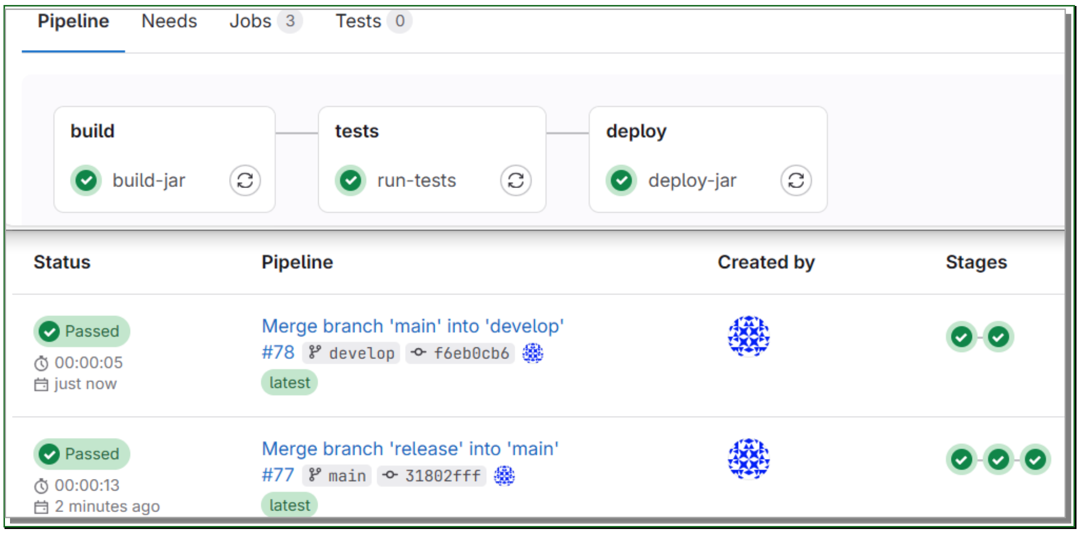

For v1.0.2
latest 2 jobs 4 seconds, queued for 1 seconds
Pipeline Needs Jobs 2 Tests 0
build: build-jar
tests: run-tests

Fig. 2. Branch and tag specific operations

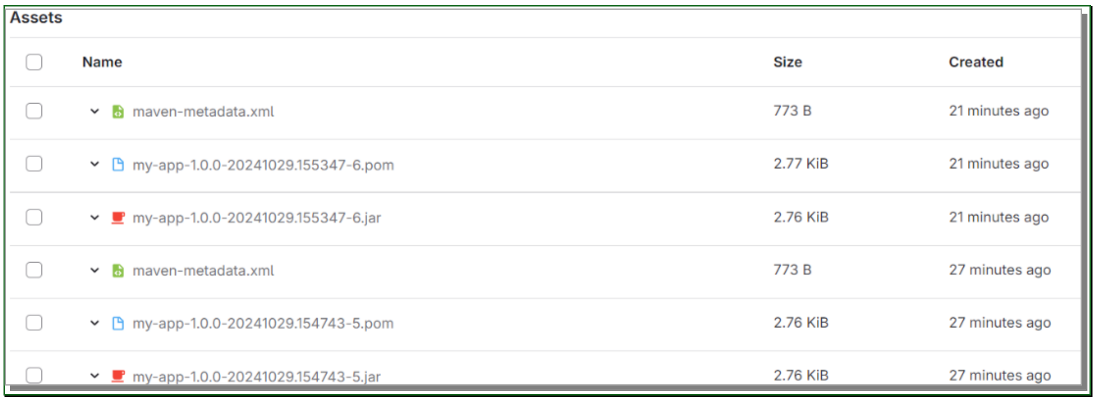
For the next evaluation the delivery behaviour was observed when new code was merged to the main branch, thus triggering the pipeline for that branch, which is the delivery pipeline responsible for deploying artefacts. Fig. 3 displays the pipeline execution reports as part of the implementation results, and the difference between CI and CD is shown in the absence of the deploy stage for the develop branch compared to the main branch. The develop branch process was executed earlier as part of the integration process. Thus, Fig. 3 displays the two *criteria C4 and C6*.



Status	Pipeline	Created by	Stages
Passed 00:00:05 just now	Merge branch 'main' into 'develop' #78 develop f6eb0cb6 latest		✓ ✓
Passed 00:00:13 2 minutes ago	Merge branch 'release' into 'main' #77 main 31802fff latest		✓ ✓ ✓

Fig. 3. Deployment of artifacts in CI vs CD

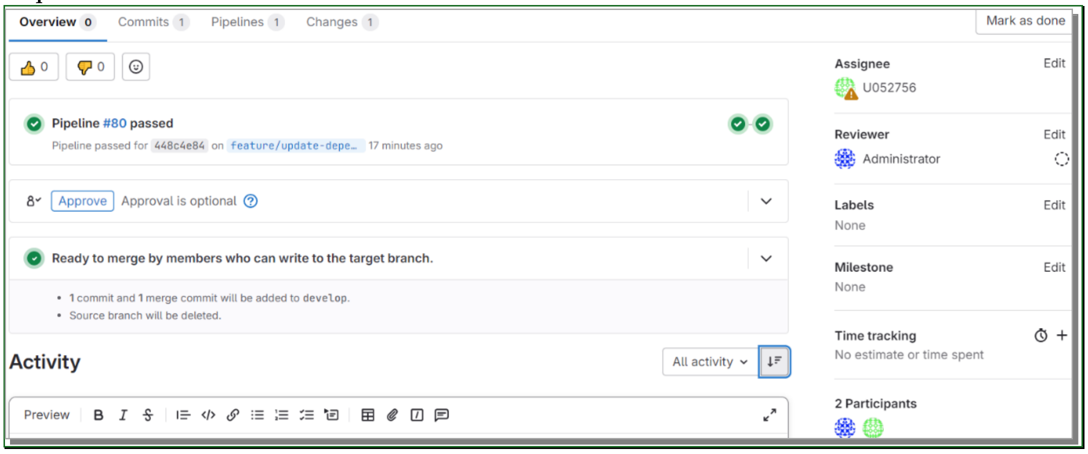
The uploaded artefacts belonging to the delivery section are displayed in Fig. 4. The pipeline generates and uploads automatically a .pom file and a .jar file for each version, but only from the main branch of the repository as this branch is meant for the delivery process, thus displaying the observation of *criterion C6*.


Assets

Name	Size	Created
maven-metadata.xml	773 B	21 minutes ago
my-app-1.0.0-20241029.155347-6.pom	2.77 KiB	21 minutes ago
my-app-1.0.0-20241029.155347-6.jar	2.76 KiB	21 minutes ago
maven-metadata.xml	773 B	27 minutes ago
my-app-1.0.0-20241029.154743-5.pom	2.76 KiB	27 minutes ago
my-app-1.0.0-20241029.154743-5.jar	2.76 KiB	27 minutes ago

Fig. 4. Delivered artefacts by the CD pipeline

For the study of *criterion C8* the aspect of merge requests and code review were employed by starting a merge request for a code change meant to be included in a delivered artefact. All the stable branches of the repository are marked as protected to prevent the introduction of unverified code. While the delivery process is shown to be fully automated from the pipeline, the acceptance of code into a stable branch requires a final manual verification before allowing the merge request.


Overview 0 Commits 1 Pipelines 1 Changes 1 Mark as done

Pipeline #80 passed
Pipeline passed for `448c4e84` on `feature/update-depe…` 17 minutes ago

Approve Approval is optional

Ready to merge by members who can write to the target branch.
- 1 commit and 1 merge commit will be added to `develop`.
- Source branch will be deleted.

Activity

Assignee Edit
U052756

Reviewer Edit
Administrator

Labels Edit
None

Milestone Edit
None

Time tracking
No estimate or time spent

2 Participants

Fig. 5. Merge request review process developer in CI

Fig. 5 offers an important overview of the manual review process: the assignee of this change is a user with no permission to merge, while the other user has permissions to merge to the protected branch. Thus, a manual review is required since the reviewer will be the one to merge the code and is responsible for it. The challenges on a real or bigger application are not different, this aspect being proven by the code review process showing a multi-user project behaviour.

5. Conclusions

This paper focused on the link between CI/CD and version control in all its aspects, from source code to build artefact management. The contributions of the authors are highlighted by the evaluation of security importance for maintaining repository integration, by identifying specific points in eight criteria of comparison and performing a parallel analysis and by the practical experiments to show different behaviours between CI and CD. The experiments include a pipeline developed in GitLab consisting of five stages that embody the software lifecycle of a personal application. The analysis exposed several aspects like individual commit behaviour, branch specific operations, artefact delivery and code review. The analysis proved that CI/CD processes are heavily linked to code version control, addressing not only source code fetching, but also the necessity of process reproduction at any time. Moreover, the start triggers rely on version control, with integration starting after a new code change is published, and delivery starting upon merging code to certain branches.

R E F E R E N C E S

[1] *N.N. Zolkifli, A. Ngah, A. Deraman*, Version control system: A review, Procedia Computer Science, 135, 2018, pp. 408-415.

[2] *P. Müller*, Configuration Management – A Core Competence for Successful through-life Systems Engineering and Engineering Services, Procedia CIRP, vol. 11, 2013, pp. 187-192.

[3] *R. Conradi, B. Westfechtel*, Version Models for Software Configuration Management, ACM Computing Surveys, vol. 30, no. 2, June 1998.

[4] *L. Bulteau, P.Y. David, F. Horn*, The Problem of Discovery in Version Control Systems, Procedia Computer Science, 223, 2023, pp. 209-216.

[5] *M. Shahin, M. Ali Babar, L. Zhu*, "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices," in *IEEE Access*, vol. 5, 2017, pp. 3909-3943.

[6] *T.A. Nitescu, A.I Concea-Prisacaru, V. Sgarciu,* Test Automation for Continuous Integration in Software Development, U.P.B. Sci. Bull., Series C, vol. 84, iss. 4, 2022, pp. 95-106.

[7] *A.V. Jha, R. Teri, S. Verma, S. Tarafder, W. Bhowmik, S. Kumar Mishra, B. Appasani, A. Srinivasulu, N. Philibert*, From theory to practice: Understanding DevOps culture and mindset, Cogent Engineering, 10:1, 2251758, 2023.

[8] *N. Paez,* Versioning Strategy for DevOps Implementations, 2018 Congreso Argentino de Ciencias de la Informática y Desarrollos de Investigación (CACIDI), Buenos Aires, Argentina, 2018, pp. 1-6, doi: 10.1109/CACIDI.2018.8584362.

[9] *D. Jones, A. Nassehi, C. Snider, J. Gopsill, P. Rosso, R. Real, M. Goudswaard, B. Hicks*, Towards integrated version control of virtual and physical artefacts in new product development: inspirations from software engineering and the digital twin paradigm, 31st CIRP Design Conference 2021, Procedia CIRP 100 (2021) 283–288.

[10] *Y. Chaysinh*, Best Practices for Managing Your Artifactory Filestore, Available at https://jfrog.com/whitepaper/best-practices-for-managing-your-artifactory-filestore-2/, 2023

[11] *R. Oberhauser*, VR-Git: Git Repository Visualization and Immersion in Virtual Reality, Proceedings of the the Seventeenth International Conference on Software Engineering Advances, 2022, pp. 9-14.

[12] *M. Girod, T. Shpichko, F. Izquierdo, T. Rydiander.* IBM Rational ClearCase 7.0: Master the Tools that Monitor, Analyze, and Manage Software Configurations Packt Publishing, 2011

[13] *P. Späth*, Git and Subversion, Pro Jakarta EE 10: Open Source Enterprise Java-based Cloud-native Applications Development, Berkeley, CA: Apress, 2023, pp. 27-42.

[14] *B.N. Chiriac, F.D. Anton, A.D. Ionita*, A hybrid IDS Architecture, U.P.B. Sci. Bull., Series C, vol. 85, iss. 1, 2023, pp. 77-90.

[15] *I.I. Anghel, R.S. Calin, M.L Nedelea, I.C. Stanica, C. Tudose. C.A Boiangiu, Software* Development Methodologies: A Comparative Analysis, U.P.B. Sci. Bull., Series C, vol. 84, iss. 3, 2022, pp. 45-58.

[16] *F. Beetz, S. Harrer*, GitOps: The Evolution of DevOps?, IEEE Software, vol. 39, no. 4, July-Aug 2022, pp. 70-75, doi: 10.1109/MS.2021.3119106.