



BerlinUP
Journals

Electronic Communications of the EASST

Volume 85 Year 2025

**deRSE25 - Selected Contributions of the 5th Conference for
Research Software Engineering in Germany**

*Edited by: René Caspart, Florian Goth, Oliver Karras, Jan Linxweiler, Florian Thiery,
Joachim Wuttke*

Implementation of DevOps with Gitlab CI/CD for a Large Satellite Simulation Software

Suditi Chand, Mike Pfannenstiel, Stefanie Bremer

DOI: 10.14279/eceasst.v85.2713

License: © ⓘ This article is licensed under a CC-BY 4.0 License.

Electronic Communications of the EASST (<https://eceasst.org>).

Published by **Berlin Universities Publishing**
(<https://www.berlin-universities-publishing.de/>)

Implementation of DevOps with Gitlab CI/CD for a Large Satellite Simulation Software

Suditi Chand, Mike Pfannenstiel and Stefanie Bremer

suditi.chand@dlr.de, <https://www.dlr.de/de/si>

Institut für Satellitengeodäsie und Inertialsensorik (SI),
Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR), Bremen, Germany

Abstract: The aim of this paper is to demonstrate the need and benefits of using Continuous Integration and Continuous Delivery (CI/CD) workflow for the testing, documentation, build and release of numerous GitLab projects (science modules) for the core VENQS[®] simulation library developed at DLR SI. The objective of this library and the supporting software development and management processes is to provide a simulation tool-chain for satellite mission design. It is crucial to have standardised workflows for module versioning, dependency management and release management systems. A down-to-top and a top-to-down pipeline architecture are implemented to meet all the project requirements. Consequently, the error-free deployment of new features and/or modules has been accelerated from 3-5 weeks to less than a week. Concluding, this paper sheds light on the implementation of a CI/CD pipeline in a large software infrastructure. Moreover, the benefits we achieved in automating the DevOps of our research software in GitLab CI/CD are discussed.

Keywords: Satellite simulation software, VENQS, DevOps, Continuous Integration, Continuous Delivery, CI/CD architecture, GitLab CI/CD, Research software, Software management, Build automation, Release management, Very Small Software Development Entity

1 Introduction

Satellite simulation software is crucial to analyse, improve and validate any new instrument, subsystem or spacecraft technology. A large software library, called VENQS[®], has been developed over the last two decades at the German Aerospace Center (DLR) and its partnering institutes in Bremen for the advanced orbit propagation of Earth-observation satellites. Moreover, it has been instrumental to analyse the effects of various forces and space environmental effects acting on a spacecraft and its instruments. At the DLR Institute for Satellite Geodesy and Inertial Sensing (DLR-SI), a small team is focusing on automating and streamlining the development and operations of VENQS[®] using DevOps best practices.

DevOps is an approach in software management with a coordinated development and operations of the software. Training the research team in DevOps practices and tools can greatly cut down the time and personnel costs that grow disproportionately along with the growth of the research software (RS). Although ideal to implement them from the start, they are beneficial even

when introduced at more advanced phases of the research. When a highly specialised or esoteric RS is developed, most RS engineers have to also become its database engineer, infrastructure developer, quality assurance manager and operations lead, despite the fact that they are no trained software engineers. Hence, only in the later stages of the project the team improves the code robustness, infrastructure, workflows and overall management as the need arises to handle the increasing complexity of the software. The solutions are found with trial-and-error methods as there is no prior knowledge of the best practices of software engineering.

This paper aims to offer road maps and best practices for adopting DevOps to tackle aforementioned challenges in developing and maintaining RS in small to large research teams. This is achieved by showcasing the need for design, implementation and results of DevOps, especially a GitLab Continuous Integration and Continuous Delivery (CI/CD) pipeline, in the scope of our software and its infrastructure. At DLR-SI, our objective is to automate the testing, documentation, build and release of each science module of the VENQS[®] library. As a result, a CI/CD pipeline is built for individual modules as well as for a package of modules. This paper gives a status of the workflows being implemented for a single-pipeline, down-to-top architecture for the former and a dynamic, multiple-pipeline, top-to-down architecture for the latter.

1.1 DevOps in Research Software

Based on the context used, the term DevOps refers to the practices, tools and/or work culture that focuses on automated, safe, robust, agile and streamlined software development and production. A systematic literature research of DevOps [LRK⁺19] presents its benefits and challenges for the developers, managers and researchers in a team. The study shows the implications of using DevOps in terms of run-time, delivery, processes and the people involved. It lists out the multitudinous DevOps tools available for the key DevOps concepts - knowledge sharing, source code (version) management, build process, continuous integration, deployment automation, and monitoring and logging. It also provides possible ways to re-structure teams or complex projects to implement DevOps for better collaboration and automation.

The benefits of adapting DevOps in research software engineering have been increasingly recognised in the last decade [Nuy23, SFR23]. A team from the IBM research lab in Brazil coined the term *ResearchOps* for their approach of incorporating the research, development and operations in a continuous cycle [DAC15]. They use their baseline practices in all the research and development projects customised to the requirements of applied research. One such requirement is to ascertain the feasibility of the proposed scientific hypothesis or solution as early in the project life-cycle as possible. ResearchOps achieves this by fast release cycles of prototypes and trying alternative solutions upon failure of the planned methods. Supporting tests are created and run by developers rather than a separate team as it requires expertise in the specific research topic. Moreover, the automation of environment configuration and the use of Infrastructure as Code (IaC) tools helped them in increasing the efficiency of the project.

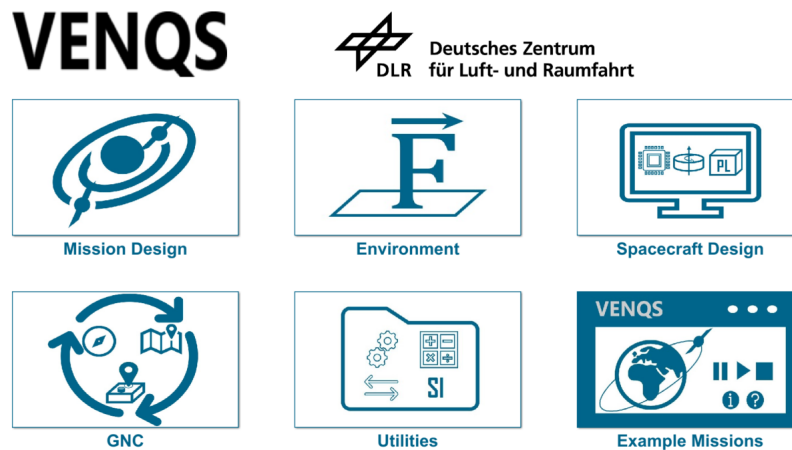
Another differentiating factor in adapting management approaches is the size of research teams. A systematic literature review of the use of CI/CD and DevOps practices in very small software development entities (VSEs), a team of 25 or less members, highlights the need of valuable trade-offs for smaller teams [CQ24]. Tool complexity, time, structural and resource limitations, and the need for standards and implementation guidelines are crucial challenges for

VSEs to adapt DevOps. Nonetheless, after the implementation of desired workflows in all case studies, all of them benefited with increased efficiency, lesser time for testing and earlier detection of errors. Another finding from their study showed that around 84% papers were only presented in conferences. Thus, there is a scarcity of case studies of CI/CD in RSE in peer-reviewed literature, especially for VSEs, to help new teams incorporate new tools and practices. As a team of six people, VENQS is definitely a VSE in the context of applied research.

The [next section](#) elaborates on why the VENQS team prioritised the setting up of DevOps, despite the scarcity of time and resources. [Section 3](#) presents the implemented workflows as CI/CD pipelines in two different architectures for the two different use cases. Lastly, [Section 4](#) encapsulates the goals achieved while shedding light on the challenges and lessons learned along the way and the work planned for the future.

2 Motivation to Adopt DevOps in VENQS Management

The VENQS[®] simulation library ([Figure 1](#)) provides functions for orbit propagation, including e.g. satellite dynamics, coordinate transformations, ground station simulations, disturbance force models and models for the Attitude and Orbit Control System (AOCS). These functions are written in either C/C++ or Matlab and are provided via library blocks in a dedicated Simulink library.



Copyright (c) 2025, Institute of Satellite Geodesy & Inertial Sensing, German Aerospace Center (DLR)

Figure 1: The VENQS[®] simulation library homepage

The library was built upon a large heritage of code which became increasingly difficult to maintain without expert knowledge. Hence, the on-boarding of new teammates was very time-consuming. The code base itself and the DevOps required a re-organisation. Further reasons that motivated us to establish automated DevOps CI/CD pipelines in the project are given in the following sub-sections.

2.1 Need for Modularization

The code base was originally organised in a monolithic repository. Its compilation was done with Makefiles triggered manually on each computer where the software should be used. In addition, the Simulink library could only be edited manually. The latter point especially led to many problems when several persons worked at the same time on the same parts of the code base. A solution to this situation was the modularization of the code whereby all functions were re-organised into science modules, such as coordinate transformations and environmental models. Each module was stored into a single GitLab project. Metadata files were established containing e.g. information on code contributors, function descriptions and most importantly, the dependencies to other science modules. Furthermore, a Matlab script was developed to automatically build the Simulink library from the information provided in the metadata files.

2.2 Need for Versioning and Module Packages

Although the modularization and the automatic library generation solved some major issues of the earlier software structure, new challenges arose: the independent development of modules induced interface problems and incompatibilities of functions that should be used together in a simulation setup. Hence, a versioning system for the science modules has been established in the Semantic versioning convention [PW] according to the following schema with major (x), minor (y) and patch (z) releases:

`module_name-vx.y.z-OS_ryyyya`

This versioning system was chosen over the Calendar versioning convention [HLW⁺] as the different types of releases of a science module could be defined in a more comprehensible way. Major releases are defined for interface changes, minor releases include function updates that don't change the existing interfaces and patch releases are used for bug fixing. In addition, the release name gives information on the operating system (OS) and the Matlab version (ryyyya) that were used for the build process.

In order to ensure compatibility between different science modules, released modules are aggregated in so-called VENQS[®] packages that guarantee matching interfaces between all functions inside one package. The distribution and installation of these packages on a PC is handled by the VENQS[®] desktop application. It interacts with the GitLab repositories and provides the user with an easy user interface to the existing science library. It was developed in parallel to the re-structuring of the research modules stated previously.

2.3 Need for Automation of the Module Release

At the beginning, the module release process was done manually. This required the following major steps:

- Clone the repository of the correct version of the source code of module to the PC
- Download the repository of the correct version of the released modules that contain required object files (module dependencies)

- Build the executable files from module sources in the desired Matlab version
- Create release folder with correct naming and content on the PC
- Upload released module to the GitLab repository

This manual process was extremely error-prone, very time-consuming and highly dependent on a single person who had the knowledge of the release procedures. In total, the release of a new package, e.g. when migrating to a newer Matlab version, could take 3 - 5 weeks.

2.4 Need for Automation of the Library Documentation and Testing

The User Guide needs to be updated for each new release with new or upgraded modules. It is difficult and inefficient to maintain such a live and dynamic document manually. Additionally, two static documents - a Contributor Guide and a Software Architecture Document, are also prepared. The workflow mentioned in the previous subsection did not include documentation. Testing was also not standardised and prepared individually by each developer without strict guidelines which resulted in lower test coverage and re-usability of the functions. The test logs, cache and artefacts generated also needed to be in a single place for traceability and control.



Figure 2: Overview of errors for module releases faced before CI/CD implementation based on a 4x4 risk matrix schema

2.5 Summary of Status before CI/CD Implementation

Based on a 4x4 risk matrix, [Figure 2](#) provides an overview of typical errors that occurred during the aforementioned manual release process. The errors are rated into the following two categories:

1. Detection point: A minor error would happen or be discovered already "during build" right at the beginning of the process, whereas the most severe problems will show up at execution of an example mission simulation in "Matlab/Simulink".
2. Time to debug: A minor error would be solved within minutes, whereas severe ones would at minimum take several hours.

As consequence, an automated process for the building, documentation, testing and releasing was planned in order to overcome these time-consuming errors. Although automation is well-established in the software engineering industry ([Subsection 1.1](#)), it is not so common in the world of R&D as time costs on DevOps are usually not credited as part of the science work. Hence, a careful trade-off and prioritisation of tasks was done to identify how much time can be allotted away from research to set up the software management infrastructure. The next section provides the solutions developed by the team, customised for sustainable and efficient research software engineering in a small team with limited resources.

3 Implementation of Customised CI/CD Pipelines

The needs outlined in the [previous section](#) helped draft the requirements for the design of the pipelines and DevOps infrastructure for VENQS[®]. This section presents the two architectures in [Subsection 3.1](#) and [Subsection 3.2](#) respectively and discusses for each - the design decisions made, the implementation of the workflows and the results.

Software and Tools Used The source code of the VENQS[®] library is written as either C/C++ functions that are compiled as s-function blocks in Simulink or as Matlab functions. The configuration code for CI/CD is written in YAML files with PowerShell scripting and Python. Metadata is stored in YAML format files, *GitLab Variables* and an in-house database and requirements management tool called MultiStage. Everything is stored and managed in a DLR domain of GitLab. Hence, GitLab's inbuilt CI/CD environment was chosen to implement the DevOps plan in accordance with the official documentation [[Git25b](#)]. The pipelines are run on a GitLab runner instance in a Windows virtual machine in the institute server.

Henceforth, the term *module* will be used synonymously to its GitLab project where the source code is saved. All modules are consolidated in the *modules* subgroup of the root library GitLab group. An additional project, called *00_data_for_module_release* (abbreviated henceforth as *00_data*), is created inside *modules* for the storage of all GitLab CI/CD pipeline scripts and as a starting point for multi-project pipelines. [Figure 3](#) shows an overview of the project infrastructure. Conceptually, we split the pipelines to two different approaches. Firstly, a down-to-top architecture, which focuses on pipelines limited to a single module. Secondly, a top-to-down architecture, where the build pipelines of multiple modules are initiated in parallel via a single

manual pipeline in the *00_data* project. A second use case of the manual pipeline is to trigger multiple API (Application Programming Interface) requests to each repository to create a User Guide document.

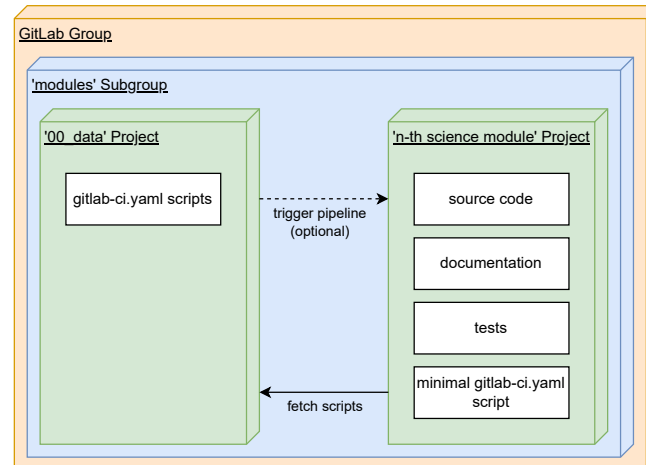


Figure 3: Overview of project structure

Single source of truth The *00_data* project is utilised as a single-source-of-truth for the scripts. It minimises the duplication of code for CI/CD scripts in each module and acts as a starting point for multiple-projects processes. The script saved in the modules essentially imports the YAML files from the *00_data* project that contain the actual jobs. Thus, the code is flexible and easily adaptable as any change in the configuration code automatically reaches all the modules. Additionally, triggering manual pipelines from the *00_data* project allows the developer to select the source branch and the different CI/CD scripts to be included in the module's pipeline for testing new features.

GitLab Features and YAML Syntax In GitLab CI/CD pipelines instructions are provided through the use of a YAML file, usually called *.gitlab-ci.yml*. This file specifies the execution of stages, jobs, and scripts, where instructions are provided through a set of keywords. These keywords are crucial to understanding the operations of the pipelines and are thus briefly introduced here. The following keywords are utilised most often:

- `extends` - Reuse configurations of job templates by providing the job name. This prevents duplication of code.
- `parallel:matrix` - Run multiple instances of a job in parallel in a single pipeline providing each instance with a different variable value.
- `trigger:include` - Set pipeline scripts and inputs, which are triggering a downstream or child pipeline.

- `spec:inputs` - Used to parameterise the behaviour of a pipeline when a configuration is added to the pipeline with the `include` keyword.
- `.pre` and `.post` stage - Jobs defined at these stages are run at the beginning or the end of a pipeline. Skipped if the pipeline contains no executable job.

The `extends` keyword is usually paired with hidden jobs denoted by a dot in front of the job name (e.g. `.job_name`), since hidden jobs are skipped during execution of the pipeline, making them perfect candidates for template jobs. The `.pre` and `.post` stage jobs are not further explained, but they played a role in creating the python virtual environment (with poetry) at the start and cleaning up the pipeline cache upon completion, in order to optimise the storage space and performance on the GitLab runner. These keywords and many more are explained in greater detail in GitLab's CI/CD YAML syntax documentation [[Git25c](#)].

3.1 Individual Modules DevOps - Down-to-Top Architecture

The first step in automating the DevOps of the library was to implement the build and release pipeline for each module. The requirements for [modularity](#), [version and package management](#) and [automation of releases](#) dictated the choice of tools, APIs and locations of data and releases in GitLab. An *Epic* was created in the parent GitLab Group, *modules* to document all requirements at the start and decisions taken later during the planning phase. The code was implemented first in a single module where all GitLab issues and merge requests (MR) that were created for testing the CI/CD pipelines, were linked to the *Epic*. Hence, discussions with the stakeholders (project leader and users) and developers were fast-tracked and transparent. The *Epic* served as a single place to document concerns and track progress via current and closed Issues. Consequently, any discordance between the requirements of the other teammates and the solution developed due to misinterpretation was detected earlier and resolved.

Design decisions A number of formative decisions were agreed upon through the iterative discussions described above. While existing software management practices were applied, it was also crucial to remember the different constraints and vantage points in research software engineering that affected us.

1. The build pipeline should iterate through the directory and find all source files to be compiled in the *src* and *contrib* folders. Earlier, the filenames were written manually in the *module.yaml* file which contains all the vital information about the module. This process was prone to human errors or lapses in updates.
2. The build script shall be developed such that it can be customised for the compilation of science modules in one or more programming languages. This is useful in the future in case of a migration to or addition of modules in a new proprietary or open source simulation software used by the institute or partnering universities or institutes.
3. The build pipeline should be triggered more often than a release so that its artefacts are usable by developers to test and validate the results after any major commits.

4. The pipeline shall be designed to be agile in terms of scripts and metadata. Thus, when crucial variables or functionality need to be changed, it can be done easily at a single point and automatically applied in all child projects. Later, for another operating system, e.g. Linux, or a new simulation software, a separate plug-and-play stage can be implemented in the pipeline.
5. The modules are built for the three most recent major versions of the software (e.g. Matlab) for slight forward and backward compatibility of the library.
6. The access of the sites of releases and source code of the module can be customised for users and developers with varying levels of access. GitLab provides that for the *Deploy* section of any project where the release can be pushed to either *Releases* or *Package Registry* subsections.
7. A release of a module is chosen to be published in its *Package Registry* for accessibility via API requests, ease of maintainability and artefacts storage. Each release is published in a *package* named after the tagged version. It consists of all releases for that tag with varying software versions and operating systems.
8. The release tags should always follow the naming schema ($v \times . y . z$) so that they can be configured in GitLab to be protected tags. This prevents unwarranted deletions of release tags via git bash commands.
9. With each release the *changelog.md* file in the root repository should be updated with the major additions and fixes.

The resulting design of the build and release pipeline is depicted in [Figure 4](#). It shows the required input for each stage as well as the job artefact or action performed as the output.

Implementation The developed CI/CD configuration script is discussed based on the different stages of the pipeline along with some snippets of the code.

- **build** - The build stage ([Listing 1](#)) is triggered in every branch for every push, defined in the rules as `$CI_PIPELINE_SOURCE == "push"`. It runs three jobs in parallel for each value of the `MATLAB_VER` fetched from the *CI/CD Group Variables* of the parent group. The following tasks are executed in each job:
 1. Read metadata from *module.yaml* files in the parent directory.
 2. Check for dependencies and fetch them from the *Package Registry* of their source repository. They can also be from external GitLab repositories in the same GitLab domain.
 3. Find and compile C or C++ files from the *src* and *contrib* folders.
 4. If an s-function exists, find all object files and dynamically create a script for compiling the Matlab s-function.
 5. Push all built files as the job artefacts for review and testing.

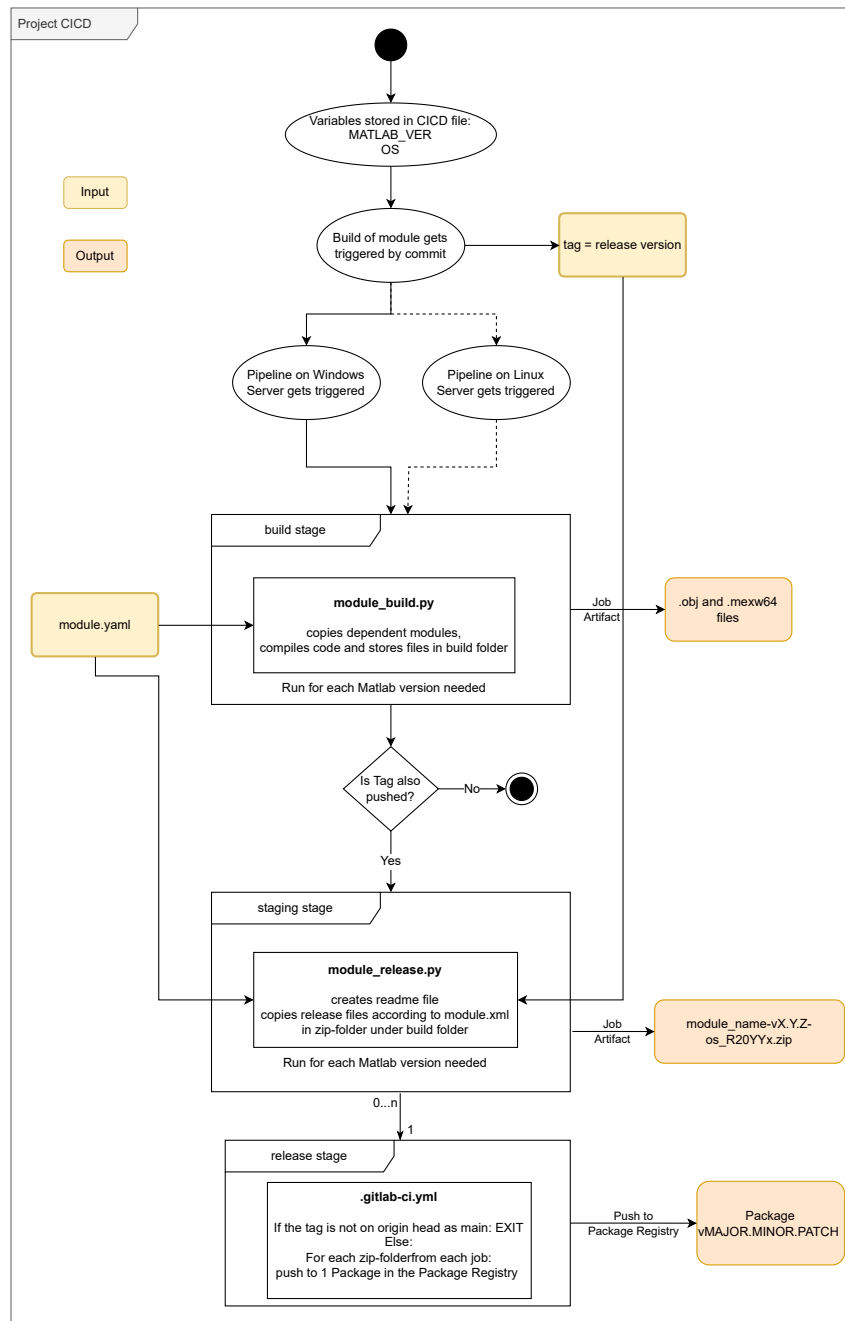


Figure 4: Activity diagram of the build and release pipeline for each module

- **staging** - It is triggered only in the main branch when a commit is tagged and once the linked build job is complete. It consists of three jobs as follows:

1. Create a folder according to the [set naming schema](#) of the module release with the appropriate suffix from the job variables.

2. Move all object files and required folders to the created folder and zip the folder.
3. Save the folder in the job artefacts.

```
1 # .module-build-release-config.yml
2 build-win:
3   stage: build
4   tags:
5     - win
6   ...
7   script:
8     - echo "Start building for $CI_JOB_NAME ..."
9     - if ( $2ND_REPO_TOKEN -ne '' -and $2ND_REPO_TOKEN -ne $null ){
10       - echo "Group variable 2ND_REPO_TOKEN is detected. It is passed
11         to method."
12       - poetry run python -m module_build $CI_MOD_GRP_TOKEN
13         $CI_SERVER_HOST $MATLAB_VER $WIN_OS_1 $2ND_REPO_TOKEN} else {
14       - echo "Only parent group access token is passed. All pre-
15         requisite modules are in this repo."
16       - poetry run python -m module_build $CI_MOD_GRP_TOKEN
17         $CI_SERVER_HOST $MATLAB_VER $WIN_OS_1 "none" }
18     - echo "Build complete for $TAG_NAME."
19   rules:
20     - if: $CI_PIPELINE_SOURCE == "push"
21       changes:
22         - "**/*.c"
23         ...
24     - if: '$CI_PIPELINE_SOURCE == "pipeline" && "$[[ inputs.MODE ]]"
25       != "TEST"'
26   artifacts:
27     paths:
28       - build/*.obj
29       - build/*.mexw64
30       - contrib/*.obj
31     expire_in: 1 week
32   parallel:
33     matrix:
34       - MATLAB_VER: [$MATLAB_1, $MATLAB_2, $MATLAB_3]
35   allow_failure: true
```

Listing 1: Code snippet for build stage

- **release** - It runs only if the staging stage is successfully executed for at least one of the three jobs. The following tasks explain the code shown in [Listing 2](#):

1. Check if the tag is valid. Else, abort.
2. Make the path of the package using predefined *GitLab CI/CD Variables* and save it to \$PACKAGE_REG_URL.
3. Push all the zip folders created in the staging jobs from the *build* folder in the GitLab runner to the module's *Package Registry* in the correct package. This is done using *Invoke-WebRequest* API. If a package with the current version name does not exist then it is created automatically.

```

1 # .module-build-release-config.yml
2 release_to_registry:
3   stage: release
4   rules:
5     - if: $CI_COMMIT_TAG
6       when: always
7       allow_failure: false
8     - if: '$CI_PIPELINE_SOURCE == "pipeline" && "$[[ inputs.MODE ]]" != "
        TEST"'
9       when: manual
10      allow_failure: true
11  dependencies:
12    - production-1
13    - production-2
14    - production-3
15  script:
16    - git fetch origin --tags --prune
17    - $tag_info = git log -1 --pretty='%D' $TAG_NAME
18    - if ( -not ($tag_info -like '*origin/main*')){
19      - echo "The tag does not belong to the main branch. Aborting release.
        "
20      - exit 1 }
21    - echo "Start release for $TAG_NAME ..."
22    - $HEADER = @{"JOB-TOKEN" = "$CI_JOB_TOKEN"}
23    - $PKG_NAME = "${TAG_NAME}"
24    - $PACKAGE_REG_URL = "$CI_API_V4_URL/projects/$CI_PROJECT_ID/packages
        /generic/$PKG_NAME/$TAG_NAME"
25    - $TEMP_PATHS = Resolve-Path -Path "build/*$TAG_NAME*.zip" -Relative
26    - foreach ($p in $TEMP_PATHS){
27      - $RELEASE_FILENAME = Split-Path $p -leaf
28      - echo "Source of released modules is $p"
29      - Invoke-WebRequest -Infile $p -Method Put -Uri $PACKAGE_REG_URL/
        $RELEASE_FILENAME -Headers $HEADER }
30    - echo "Pushed all zips to Package $PKG_NAME"

```

Listing 2: Code snippet for release stage

Result The release is published in the *Package Registry* and available for users to download via the VENQS[®] App or for the GitLab Runner for the build of a dependent module. After every push to any branch, the artefacts of the build job are created and configured to delete themselves in a week. In the meantime, they can be downloaded to be tested for interface compatibility with other modules of interest or for functional validation without the need to create a release. This helps in bolstering the Shift Left testing practices with earlier function and interim integration tests. In compliance with the software security concept of DevOps, all jobs can only start with a group access token created by the developers and passed to the job via the group variable `$CI_MOD_GRP_TOKEN`. Only members who have *maintainer* roles or higher can create or delete a tag. This can be configured in the repository settings under *protected tags*. This guarantees that a new feature is merged to the main (protected) branch only after the predetermined quality assurance measures for the branch are carried out by the responsible person. Concluding, a pipeline is implemented for the build and release of individual science modules as shown in Figure 5.

Create release for v1.0.0

✓ Passed Chand, Suditi created pipeline for commit c6e5fb34 10 months ago, finished 10 months ago

For v1.0.0

latest tag 8 jobs 1 minute 16 seconds, queued for 0 seconds

Pipeline Jobs 8 Tests 0

Group jobs by Stage Job dependencies Show dependencies



Figure 5: A module pipeline in GitLab

3.2 Multiple-project DevOps - Top-to-Down Architecture

While the down-to-top architecture aims to create pipelines for single module repositories, certain workflows related to a package release require communication between multiple module repositories. Two such workflows are needed to satisfy demands made by stakeholders. These demands are:

- Creation of a User Guide for the package. This is a single PDF document or HTML page, which combines documentation present in each module repository specified in the package. This document is mainly focused on the mathematical description of the physical models used.
- Re-release of a package for an updated software version (e.g MATLAB version, new operating system version). The package contains multiple modules each necessitating a new build and subsequent release.

From these demands the specifications for the top-to-down architecture can be inferred. Required are communication with multiple module repositories in order to fetch data from these repositories or trigger new workflows. We utilised GitLab's downstream pipelines and REST API calls to achieve this goal [Git25e]. Conceptually, the top-to-down architecture can be split into three different aspects, which will be explained in the following subsections:

1. A manual pipeline as a starting point. It utilises user information and consequently initialises the corresponding workflow.
2. A pipeline which fetches documentation files from all module repositories with the REST API and creates the User Guide.
3. A pipeline which triggers child pipelines in each module repository and runs the build and release workflow specified in the down-to-top architecture.

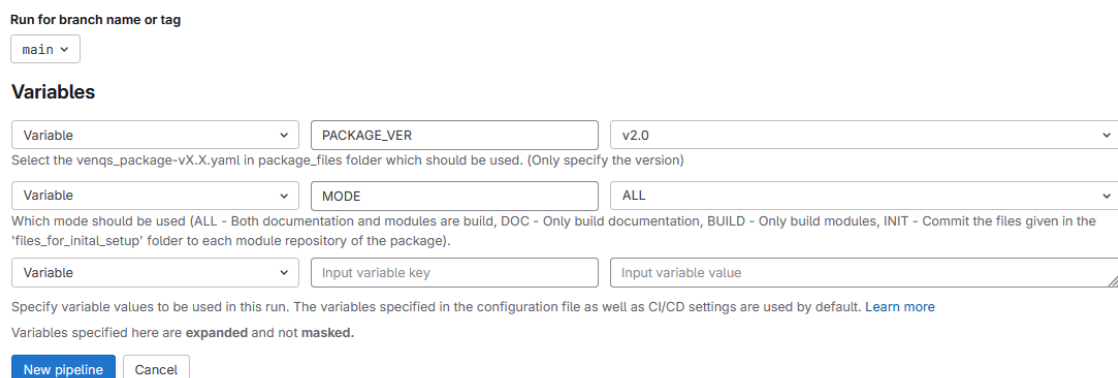
3.2.1 Implementation with Manual Pipelines

In essence the manual pipelines goal is to start new workflows with a set of dynamic variables which should be easy to provide by users. Gitlab provides a solution for that. It is possible to start pipelines from the web interface of the *Pipelines* tab in every Gitlab project for which CI/CD is enabled, by clicking the *New Pipeline* button. This results in a interface where input fields appear for each variable specified in the global *variables* keyword of the *gitlab-ci.yml*. With the release of Gitlab v17.11 *spec:inputs* can also be used instead of *variables* keyword, this was however not available to us at the time of implementation [Git25a]. With this we were able to pass information from the web interface to a pipeline script and thus run jobs with dynamic variables.

Design decisions The first step is to define a location from which the the workflows will originate. Since the *00_data* project is already utilised for the pipeline scripts it is the perfect candidate. Next we defined a set of variables containing necessary information. The following variables were decided upon and an example of the resulting web UI is shown in [Figure 6](#):

1. `PACKAGE_VER`: It determines which package is targeted.
2. `MODE`: It determines which workflow is started and contains the following options:
 - `BUILD`: Used to start the workflow for the re-release.
 - `DOC`: Used to start the workflow for the creation of the User Guide.
 - `ALL`: Used to start both workflows.

Run new pipeline



Run for branch name or tag

main ▾

Variables

Variable ▾ PACKAGE_VER v2.0 ▾

Select the `venqs_package-vX.X.yml` in `package_files` folder which should be used. (Only specify the version)

Variable ▾ MODE ALL ▾

Which mode should be used (ALL - Both documentation and modules are build, DOC - Only build documentation, BUILD - Only build modules, INIT - Commit the files given in the 'files_for_init_setup' folder to each module repository of the package).

Variable ▾ Input variable key Input variable value

Specify variable values to be used in this run. The variables specified in the configuration file as well as CI/CD settings are used by default. [Learn more](#)

Variables specified here are **expanded** and not **masked**.

New pipeline Cancel

Figure 6: GitLab UI element for manual pipeline creation

For each package a metadata file, *venqs_package-vX.Y.yml* exists, where X.Y is the given package version. The metadata file contains module name and a module version which corresponds with a commit tag in the module repository. Thus providing a package version is sufficient information to determine a set of modules for the foreseen workflows.

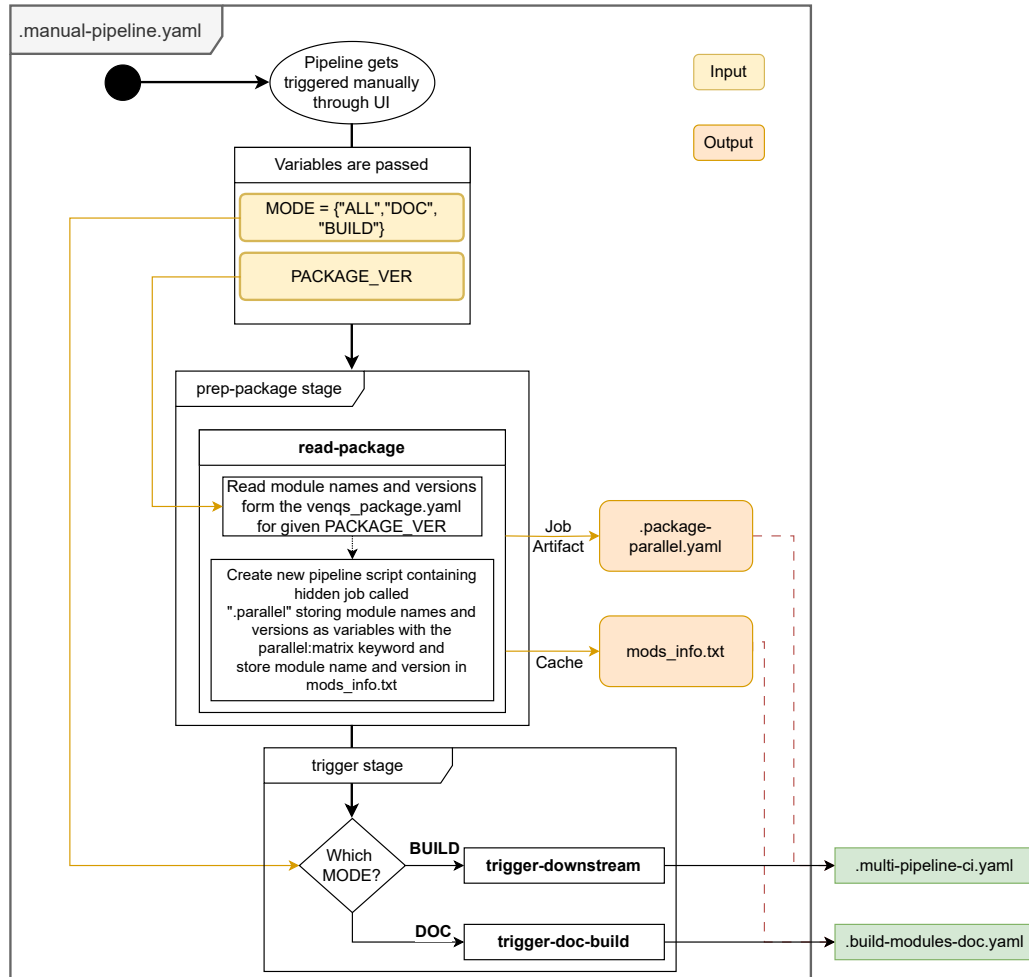


Figure 7: Overview of the pipeline jobs for the top-to-down architecture

Implementation The *.manual-pipeline.yaml* script is started every time a new pipeline is created in the Gitlab web UI, since it was set as the default CI/CD configuration file in the Gitlab CI/CD settings of the *00_data* project. In each run, the subsequent steps are preformed, which are also depicted in [Figure 7](#).

1. Pipelines can be triggered in the GitLab web UI (Build ⇒ Pipelines ⇒ New Pipeline).
2. User inputs values for `PACKAGE_VER` and `MODE` in the web UI. These variables are defined in the *.manual-pipeline.yaml* with the `variables` keyword.
3. A generator job called `read-package` starts and reads the *venqs_package_vX.X.yaml* for the specified `PACKAGE_VER`. Information about the module repository name and version are gathered and stored in two files:

- (a) **.package-parallel.yaml** - A new .gitlab-ci.yaml file. Its task is to parallelise execution of other jobs for all modules. It is passed as an artefact to the jobs in the next step. The syntax of this script and use case is detailed at the end of this paragraph.
 - (b) **mods_info.txt** - Comma separated values of all module names and versions contained in the package. It is stored in cache and can be used by jobs that require knowledge on the modules.
4. The repository name and tag are stored in these files. The information is later used to determine which commit should be used for the task. The *.package-parallel.yaml* is stored as an artefact, while the *mods_info.txt* is stored in the cache.
 5. Depending on the selected MODE in step 2, different workflows get triggered from here using a trigger job. These jobs are specified in the following files:
 - (a) **.multi-pipeline-ci.yaml** - Trigger multiple module releases ([Subsubsection 3.2.2](#)).
 - (b) **.build-modules-doc.yaml** - Create package documentation ([Subsubsection 3.2.3](#)).

The trigger job used for the *.multi-pipeline-ci.yaml* is shown in [Listing 3](#). A similar job exists for the *.build-modules-doc.yaml*. The rules keyword limits the job execution based on the MODE and the trigger:include keyword starts a new child pipeline for the jobs defined in the *.multi-pipeline-ci.yaml* while also including the job specified in *.package-parallel.yaml*.

```

1 # .manual-pipeline.yaml
2 trigger-downstream:
3   stage: trigger
4   needs:
5     - job: read-package
6       artifacts: true
7   rules:
8     - if: '$MODE == "BUILD" || $MODE == "ALL"'
9   trigger:
10    strategy: depend
11    include:
12      - artifact: .package-parallel.yaml
13        job: read-package
14      - local: .multi-pipeline-ci.yaml
15    inputs:
16      SRC_PROJECT: $CI_PROJECT_PATH
17      SRC_BRANCH: $CI_COMMIT_REF_NAME
18      MODE: $MODE

```

Listing 3: Code snippet from the .manual-pipeline.yaml

To elaborate more on the syntax of the *.package-parallel.yaml* file an exemplary code snippet is shown in [Listing 4](#). The file defines the hidden job *.parallel*, which contains the *parallel:matrix* keyword initialising a list of key-value pairs. This allows jobs defined with the *extends* keyword to run multiple times in parallel with different environment variable values. Here, we assign *DOWNSTREAM_PROJECT_NAME* and *DOWNSTREAM_PROJECT_VER* with the module name and version for each module in the package.

```
1 # .package-parallel.yaml
2 .parallel:
3   parallel:
4     matrix:
5       - DOWNSTREAM_PROJECT_NAME: module_name_1
6         DOWNSTREAM_PROJECT_VER:  module_version_1
7       - DOWNSTREAM_PROJECT_NAME: module_name_2
8         DOWNSTREAM_PROJECT_VER:  module_version_2
```

Listing 4: Example code contained in the `.package-parallel.yaml`

Results Consequently, the resulting child pipeline from the trigger job will now know about the `.parallel` job and any subsequent job can run for all modules of a package using the `extends` keyword. This is now possible by only specifying the package version in the web UI.

3.2.2 The Multi-Project Trigger Pipeline

The workflow of the `.multi-pipeline-ci.yaml` script is shown in [Figure 8](#). The goal is to start the build pipeline, mentioned in the down-to-top architecture in each module. This can be achieved by utilising GitLab’s REST API to start new child pipelines in each module repository. The `multi-proj-trigger-job` (shown in [Listing 5](#)) performs the following tasks:

1. The inputs from `.manual-pipeline.yaml` are passed via the `spec:inputs` keyword and can be referenced using an interpolation format (`$(inputs.variable_name)`) anywhere in the pipeline script.
2. The `extends` keyword enables parallel execution of the script section for each module. This is achieved through the `.parallel` job, by referencing the module name and version with `DOWNSTREAM_PROJECT_NAME` and `DOWNSTREAM_PROJECT_VER` respectively.
3. In the script section REST API calls are made using GitLab’s pipeline triggers API endpoint.
4. Different dynamic variables can be passed through the API call to the child pipelines.

This results in a set of pipelines running the build scripts in each module repository, thus enabling re-releases for a package. In addition, variables can be passed to the `.gitlab-ci.yaml` scripts in each module repository. This, in contrast to the down-to-top approach, allows dynamic selection of the `SRC_PROJECT`, `SRC_BRANCH` and `MODE` variables. Passing these all the way from the manual pipeline, enables the child pipeline in the module repositories to run with different behaviour based on these variables. This for example grants the benefit, that development of the pipeline scripts is possible in dedicated development branches in the `00_data` project. Thus, the main branch in `00_data` is only used if pipelines have been tested in the development branches. This in turn makes the main branch as the default from which the pipeline scripts are cloned.

```

1 # .multi-pipeline-ci.yaml
2 spec:
3   inputs:
4     SRC_PROJECT:
5     SRC_BRANCH:
6     MODE:
7   ---
8   stages:
9     - multi-proj-trigger
10
11 multi-proj-trigger-job:
12   stage: multi-proj-trigger
13   extends: .parallel
14   script:
15     - $project_url = "https://<GITLAB_DOMAIN>/api/v4/projects/
16       $CI_PROJECT_ROOT_NAMESPACE%2Fmodules%2F$DOWNSTREAM_PROJECT_NAME/
17       trigger/pipeline"
18     - $Body = @{"token = $CI_JOB_TOKEN;
19       ref = $DOWNSTREAM_PROJECT_VER;
20       'variables[SRC_PROJECT]' = "$[[ inputs.SRC_PROJECT ]]";
21       'variables[SRC_BRANCH]' = "$[[ inputs.SRC_BRANCH ]]";
22       'variables[MODE]' = "$[[ inputs.MODE ]]"}
23     - Invoke-RestMethod -Method Post -Body $Body -Uri $project_url -
24       UseBasicParsing

```

Listing 5: Code snippet from the .multi-pipeline-ci.yaml

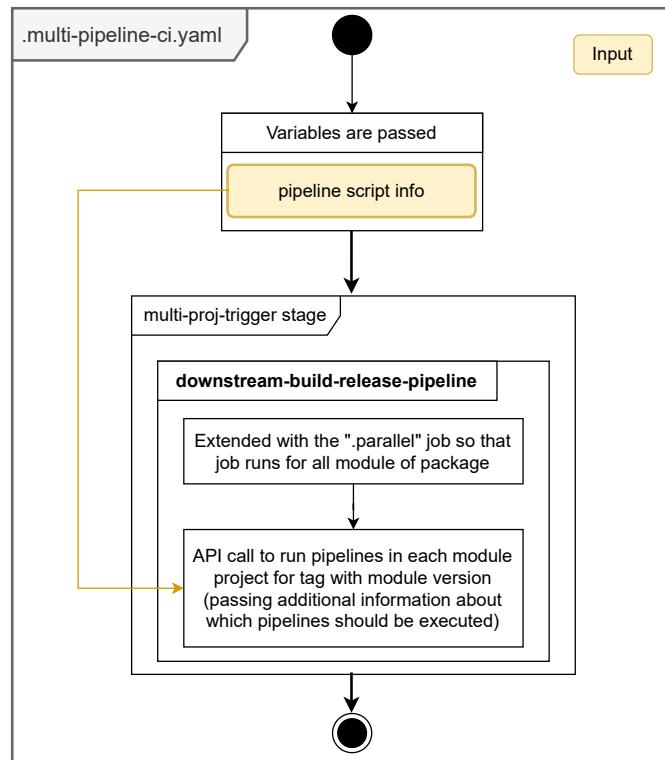


Figure 8: Overview of the multi-project trigger pipeline

3.2.3 The Documentation Pipeline

The workflow of the `.build-modules-doc.yaml` script is shown in Figure 9. Instead of creating child pipelines for each repository, this pipeline invokes multiple *REST API* requests to interact with the module repositories. This ensures that the User Guide is built with updated documentation of each module of the science library. It also allows for cross-referencing of the different module documentation.

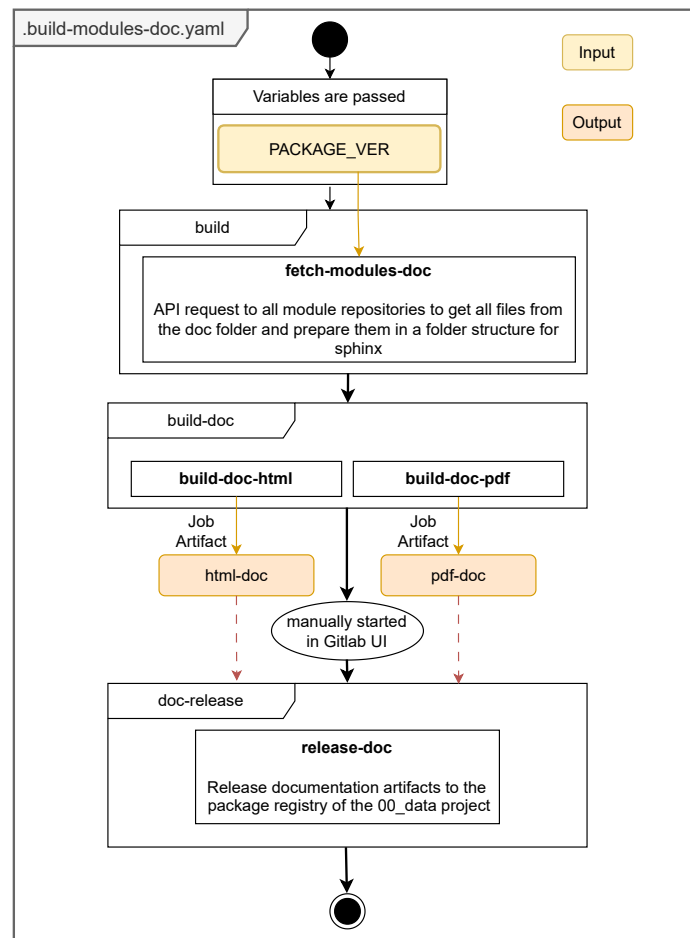


Figure 9: Overview of the documentation pipeline for the User Guide

The first job called `fetch-modules-doc` utilises a PowerShell script focused on the following tasks executed in each child module from the list in `mods_info.txt`:

1. List the repository tree and find all documentation files.
2. Download each documentation file.
3. Store the files in a directory structure usable by Sphinx.
4. Create the `index.md` file containing the `toctree` directive.

The first two tasks can be accomplished by using the repository API [Git25d] and obtaining the git tree object for each repository. With this information all documentation file paths can be extracted from the git tree object. Consequently, these file paths can be utilised with the repository files API to get the `base64` string encoded file content of the markdown and image files. Decoding them generates all the necessary documentation files on the runner. The files are then organised in separate folders for each module and the necessary metadata files for Sphinx are created.

Next the `build-pdf-doc` and `build-html-doc` jobs are run, which use Sphinx to create the PDF and HTML documentation respectively. The results are passed as artefacts to the `release-doc` job which stores the documentation, versioned for each package, in the *Package Registry* of the *00_data* project. This process allows the documentation of multiple modules to be interlinked and cross-referenced, without additional steps. Finally, the final User Guide documentation file for the given library package release is attached to the latter's release as *release notes* in the *Deploy/Releases* section of the GitLab project dedicated for VENQS[®] desktop application's deployment.

4 Conclusion

This paper is a gist of the journey of our team from identifying recurring or long-term challenges in adapting VENQS with DevOps tools and practices as a solution. Through the presented CI/CD pipelines we are able to address all the needs that were given in Section 2. The detailed design and implementation of the DevOps workflows in Section 3 can be reused for research project with similar project needs.

For a comparative performance analysis, we revised the error matrix introduced in Subsection 2.5 after the implementation of the pipelines. Figure 10 shows stark improvement in the new matrix with no severe D3 or D4 risks. Some errors are permanently resolved by automated pipelines whereas others degraded to a lower severity level. The errors of the "D"-class will be reduced further with system level tests.

Error Description	Matrix	Error Description	Matrix
Wrong source branch cloned	A1	Wrong source branch cloned	n.a.
Compilation errors when building object files without dependencies	A1	Compilation errors when building object files without dependencies	A1
Compilation errors when building object files with dependencies	A2	Compilation errors when building object files with dependencies	A2
Compilation errors when building mex executable	A3	Compilation errors when building mex executable	A2
Spelling error in released module folder	B1	Spelling error in released module folder	n.a.
Error in metadata: Missing dependencies for installation	C3	Error in metadata: Missing dependencies for installation	C1
Files for module execution are missing	D2	Files for module execution are missing	D1
Wrong Simulink model in released module folder	D3	Wrong Simulink model in released module folder	n.a.
Example mission is not running	D4	Example mission is not running	D3

Figure 10: Comparison of matrix on errors before (left) and after (after) CI/CD implementation

In agreement to results of other RS teams using DevOps in literature we also benefited in multifaceted ways. The code quality has improved. Documentation, build and release processes have been automated, thereby exponentially reducing their run-time and making them robust. Logging

and monitoring have improved. Consequently, the collaboration, transparency and communication amongst developers, stakeholders and operations lead of the VENQS team has advanced. The workflow scripts are generic, modular and standardised for faster re-usability for external repositories of project partners. The on-boarding of new team members is faster. Lastly, after automated DevOps tools and practices went into effect, DevOps time was freed and re-allocated to core research software development.

4.1 Challenges and Lessons Learned

As a team full of physicists and spacecraft engineers with no prior knowledge on DevOps and CI/CD, the journey to becoming DevOps practitioners was adventurous and insightful. The unique architecture of our software project made it challenging to adapt existing CI/CD workflows to fit our needs, and we had to design them mostly from scratch. The process was time-costly in the beginning but as the initial learning curve was surpassed and the initial CI/CD framework was set-up, further development and improvements were fast to implement. One challenge was to debug the Powershell commands used in the CI/CD configuration files as sometimes they exhibit different behaviour in the GitLab Runner as compared to tests done in a Powershell window in the virtual machine. But this was overcome by trial-and-error testing. Another problem was that sometimes job failures can be time-consuming to debug. But in general, GitLab pipelines are comparatively easier to setup, while still allowing for complex jobs by utilising GitLab APIs. The use of UML Activity diagrams and documentation helped for lucid design processes and knowledge transfer. Furthermore, a review with research software experts at the end of the design phase helped to save time and to avoid the foreseen errors during the following implementation phase.

4.2 Future Work

Future development of the pipelines is focused on automating the testing of modules. Investigating testing architectures which can be used for C++ or Matlab are of interest for the science modules. The next step is to figure out how extensive the tests should be and incorporate them into the pipelines. Focus will remain on unit and integration tests, but other testing strategies will be explored as well like functional, acceptance and end-to-end tests. Moreover, some other features might be evaluated in the future for relevance to our requirements, such as, Docker Images, Protected Environments and CI/CD Components.

Bibliography

- [CQ24] M. Ccallo, A. Quispe-Quispe. Adoption and Adaptation of CI/CD Practices in Very Small Software Development Entities: A Systematic Literature Review. 09 2024. National University of the Altiplano of Puno.
[doi:10.48550/arXiv.2410.00623](https://doi.org/10.48550/arXiv.2410.00623)
- [DAC15] M. De Bayser, L. Azevedo, R. Cerqueira. ResearchOps: The case for DevOps in scientific applications. Proceedings of the 2015 IFIP/IEEE International Symposium

- on Integrated Network Management, IM 2015, pp. 1398–1404, 06 2015.
[doi:10.1109/INM.2015.7140503](https://doi.org/10.1109/INM.2015.7140503)
- [Git25a] GitLab. GitLab 17.11 Release. July 2025.
<https://about.gitlab.com/releases/2025/04/17/gitlab-17-11-released/>
- [Git25b] GitLab. GitLab Documentation: CI/CD pipelines. May 2025.
<https://docs.gitlab.com/ci/pipelines/>
- [Git25c] GitLab. GitLab Documentation: CI/CD YAML syntax reference. May 2025.
<https://docs.gitlab.com/ci/yaml/>
- [Git25d] GitLab. GitLab Documentation: Repositories API. May 2025.
<https://docs.gitlab.com/api/repositories/>
- [Git25e] GitLab. GitLab Documentation: REST API. May 2025.
<https://docs.gitlab.com/api/rest/>
- [HLW⁺] M. Hashemi, S. LaPorte, M. Williams, G. Lefkowitz, A. Brown, H. Schlawack. Calendar Versioning. A timely project versioning convention.
<https://calver.org/overview.html>
- [LRK⁺19] L. Leite, C. Rocha, F. Kon, D. Milojevic, P. Meirelles. A Survey of DevOps Concepts and Challenges. *ACM Comput. Surv.* 52(6), Nov. 2019.
[doi:10.1145/3359981](https://doi.org/10.1145/3359981)
- [Nuy23] P. Nuyujukian. Leveraging DevOps for Scientific Computing. Oct. 2023. Bio-X, Wu Tsai Neurosciences Institute, Stanford University.
<https://arxiv.org/pdf/2310.08247>
- [PW] T. Preston-Werner. Semantic Versioning 2.0.0. A software versioning convention.
<https://semver.org/spec/v2.0.0.html>
- [SFR23] M. Steidl, M. Felderer, R. Ramler. The pipeline for the continuous development of artificial intelligence models—Current state of research and practice. *Journal of Systems and Software* 199, 05 2023.
[doi:10.1016/j.jss.2023.111615](https://doi.org/10.1016/j.jss.2023.111615)