DECICE

DEVICE-EDGE-CLOUD INTELLIGENT COLLABORATION FRAMEWORK

Grant Agreement: 101092582

D4.1 Implementation Report on CI/CD Environment

This project has received funding from the European Union's Horizon Europe Research and Innovation Programme under Grant Agreement No 101092582.

Document Information

Deliverable number:	D4.1
Deliverable title:	Implementation Report on CI/CD Environment
Deliverable version:	1.0
Work Package number:	WP4
Work Package title:	Cloud Management Framework Integration
Responsible partner	USTUTT
Due Date of delivery:	2023-05-31
Actual date of delivery:	2023-05-31
Dissemination level:	PU
Type:	R
Editor(s):	Steven Presser (USTUTT)
Contributor(s):	Felix Stein (UGOE) Julian Kunkel (GWDG/UGOE)
Reviewer(s):	Denise Drossos (SYNYO) Julian Kunkel (GWDG/UGOE)
Project name:	Device-Edge-Cloud Intelligent Collaboration framEwork
Project Acronym:	DECICE
Project starting date:	2022-12-01
Project duration:	36 months
Rights:	DECICE Consortium

Document History

Version	Date	Partner	Description
0.1	2023-05-17	USTUTT	Initial Draft
0.2	2023-05-22	SYNYO	Review 1
0.3	2023-05-23	GWDG	Review 2
1.0	2023-05-23	USTUTT	Final Draft

Acknowledgement: This project has received funding from the European Union's Horizon Europe Research and Innovation Programme under Grant Agreement No 10192582.

Executive Summary

This document outlines the policies and practices of the DECICE project as related to the software development environment. Specifically, it details the Continuous Integration/Continuous Delivery (CI/CD) environment that is used to build software.

Continuous Integration/Continuous Delivery (CI/CD) is a series of methods related to software development. These encourage writing automated tests for software as well as automating common or critical tasks (such as releasing software). The aim of CI/CD practices is to unite the development and test phases of software development, and always have software be "ready to release". Primarily this is accomplished via automated testing. Additionally, it automates tasks like software release. By automating software release, CI/CD attempts to make it so releases of software happen more often – potentially even weekly. Though these methods, CI/CD attempts to build higher quality, better tested software and get that software to users more quickly.

The software that implements CI/CD is often discussed as a pipeline. At the start of the pipeline, the software code is input. The code is then run through a series of steps to produce artifacts – usually compiled or packaged software. The steps in the pipeline may include running tests, compiling the software and running security tools against the software. Running these steps for every change to the software finds issues extremely early in the development process, which makes it much easier to fix them.

This deliverable discusses the process of designing a CI/CD pipeline for the DECICE project, discusses the decisions and trade-offs made, and summarizes the required features. It then includes code listings implementing the aforementioned functionality within a GitLab repository. These code listings were taken directly from the DECICE example software repository and serve as a base for all DECICE.

This deliverable also details policies and requirements for repository creation and use, as well as branching and merging within repositories. These are important to help software developers navigate the project's software, ensure consistency, and generally ease the software development process.

Finally, the deliverable demonstrates how the written policy and the CI/CD environment function in concert to produce high quality code.

Contents

List of Figures

List of Abbreviations

- API – Application Programming Interface
- CD – Continuous Delivery
- CI – Continuous Integration
- DECICE – Device-Edge-Cloud Intelligent Collaboration framEwork
- WP – Work Package
- WPL – Work Package Leader

1 Purpose and Scope of the Deliverable

Technical description of the selected CI/CD environment including the chosen code quality and security tools. This further includes the description of the configuration of quality gates preventing poor quality or insecure code from being built. Additionally, the deliverable discusses policy. The policy is designed to work with the quality gates to produce high-quality code. Finally, the code for a sample repository implementing the CI/CD environment is included.

2 Abstract / publishable summary

This document details the development process, features and sample implementation of a Continuous Delivery/Continuous Deployment pipeline. This pipeline is used within the DECICE project to ensure the software is produced with a very high level of quality. The pipeline performs automated tasks, including building and testing of the software. This checks code quality and security to ensure that released software is of extremely high quality.

3 Project objectives

This deliverable contributes directly and indirectly to the achievement of all the macro-objectives and specific goals indicated in section 1.1.1 of the project plan:

Macro-objectives	Contribution of this deliverable
(O1) Develop a solution that allows to leverage a compute continuum ranging from cloud and HPC to edge and IoT.	Working in such disparate environments requires software that is flexible, written once, and is provably correct in all relevant environments.
(O2) Develop a scheduler supporting dynamic load balancing for energy-efficient compute orchestration, improved use of green energy, and automated deployment.	A well-crafted CI/CD pipeline increases the speed and quality of implementation, enabling faster iteration on the scheduler and ensuring that the scheduler is as effective at meeting its goals as possible.
(O3) Design and implement an API that increases control over network, computing and data resources.	A well-crafted CI/CD pipeline increases the speed and quality of implementation of the DECICE API, allowing the project to provide better quality software and build more features to cover more use cases.
(O4) Design and implement a Dynamic Digital Twin of the system with AI-based prediction capabilities as integral part of the solution.	Use of a CI/CD framework allows automated testing of Dynamic Digital Twins and AI prediction capabilities on an ongoing basis. The DECICE project can thus quickly determine what effects any software change has on the Digital Twin and AI prediction within minutes, greatly speeding up the development cycle and minimizing the need for later revisions to either.
(O5) Demonstrate the usability and benefits of the DECICE solution for real-life use cases.	A CI/CD framework will allow automated deployment of the DECICE framework to testbeds which demonstrate the utility of the project software in real-world use cases.
(O6) Design a solution that enables service deployment with a high level of trustworthiness and compliance with relevant security frameworks.	The use of a CI/CD pipeline allows integration of security tools, ensuring that DECICE software is secure from the beginning of the development process.

4 Changes made and/or difficulties encountered

No significant changes to the project plan were made. No significant challenges were encountered during implementation.

5 Sustainability

The CI/CD portion of Work Package (WP) 4 is tightly coupled to Work Package 1, as WP 1 sets many of the standards in use for software. It was the responsibility of WP4 to select software to enforce these standards. In the case of this pipeline, WP1 set minimum standards for test coverage, code style and security. WP4 then selected and implemented tools in the CI/CD pipeline to verify that the code meets these standards.

CI/CD pipelines are a major topic within EuroHPC Joint Undertaking (JU) projects. USTUTT also has significant involvement in the CI/CD pipeline effort for all JU Hosting Sites and Centers of Excellence via CASTIEL2. Significant synergies were exploited between the two simultaneous CI/CD efforts. DECICE provided a use case and testbed for CASTIEL2 to test CI/CD features before deployment by Hosting Sites and Centers of Excellence. In exchange, DECICE benefited from the expertise of the CASTIEL2 project.

DECICE has also benefited by being a testbed for some features expected to be deployed in the CASTIEL2 CI/CD pipelines. For example, the DECICE CI/CD pipeline uses a script-based decoupling between the CI management software and the tasks that are part of the pipeline. This feature allows the DECICE project to easily move between CI management systems without rewriting significant portions of the CI management tasks. It also better enables individual contributors to run tasks exactly as they will run within the CI/CD system. Contributors can therefore better determine exactly how any changes they make will affect the DECICE software. Finally, it ensures that the same software is run both by individual contributors and by the CI/CD pipeline, which reduces the software maintenance required and ensures tasks are performed in a consistent manner by both.

While implementing the CI/CD pipeline, the DECICE project learned several important lessons. Dividing the decision-making and implementation between work packages had both positives and negatives. In terms of positive outcomes, WP1 was able to decide on standards entirely without concern for tooling or how the standards would be implemented. This also assisted WP4, which was able to choose the best tools to implement the selected standard. However, the project did also learn that it is important to set clear timelines and maintain open communication between work packages during the standards-setting process. The finalization of the standards came relatively late and reduced the working time WP4 had to implement them. Either setting the standards earlier in the project or more communication would have allowed WP4 to begin implementation of the standards sooner. However, this was not an issue for the DECICE project because WP4 was able to work on other CI/CD related tasks.

6 Dissemination, Engagement and Uptake of Results

6.1 Target audience

As indicated in the Description of the project, the audience for this deliverable is:

✓	The general public (PU)
	The project partners, including the Commission services (PP)
	A group specified by the consortium, including the Commission services (RE)
	This report is confidential, only for members of the consortium, including the Commission services (CO)

6.2 Record of dissemination/engagement activities linked to this deliverable

See Table 1.

Type of dissemination and communication activities	Details	Date and location of the event	Type of audience activities	Zenodo Link	Estimated number of persons reached
None	N/A	N/A	N/A	N/A	0

Table 1: Record of dissemination / engagement activities linked to this deliverable

6.3 Publications in preparation OR submitted

See Table 2.

In preparation or submitted?	Title	All authors	Title of the periodical or the series	Is/Will open access be provided to this publication?
None	N/A	N/A	N/A	

Table 2: Publications related to this deliverable

None. Publications based on or involving this effort may be written in the future because of the co-development with CASTIEL2 CI/CD pipelines for JU Hosting Sites and Centers of Excellence.

6.4 Intellectual property rights resulting from this deliverable

None.

7 Detailed report on the deliverable

This deliverable is divided into two parts: policy and implementation.

The policy portion consists of written policies that are intended to guide development for the DECICE project. These vary from step-by-step documents describing how to perform tasks to aspirational documents which describe the goals of the project and are intended to act as guidance for how to achieve them.

The implementation consists of code. Primarily this code implements or supplements the policy. Additionally, it performs critical steps in building the software.

The rest of this document discusses methodology, features, and architecture of the resulting work product. Additional details and results of this work may be found in appendices A and B, at the end of this document.

7.1 Methodology

The CI/CD pipeline is somewhat unique, in that it impacts or is impacted by almost every work package within the DECICE Project. Specifically, it impacts all the software development work packages (WP2-5) and is impacted by WP1. Therefore, this work, by necessity, involved significant inter-Work Package communication.

WP4 chose to apply a guided user-assisted design approach to the CI/CD pipeline. In such an approach, knowledgeable experts discuss software with the users, in order to determine what software architecture is best suited to solving the problem. It is particularly well-suited to use with users knowledgeable about software. Unlike other methods, this methodology does not simply gather perceived requirements from the user. Instead, it engages users in a dialog and encourages them to be active design participants while the knowledgeable experts guide the discussion. Initially the experts simply ask questions in order to create a sketch of the architecture. Later, they propose an architecture and discuss the strengths and weaknesses with users. At this stage, the experts may challenge user assumptions about requirements, if they feel they may not be well-founded or may be founded in a user's assumption about software architecture.

Thus, the work began with members of WP4 building and refreshing their knowledge of CI/CD systems. Additionally, members of WP4 researched technology and solutions already available to them within DECICE member organizations.

The next step was a discussion with users. It took place at the first DECICE in-person all-hands meeting. This meeting was held in a world café format [Caf15]. In this format, participants are split into groups and move between stations. At each station, there is a moderator, who guides discussion for the station and takes notes. The last group at each station then also selects a speaker and summarizes the notes for presentation to the entire group. Finally, there is a discussion with the entire group.

The World Café format worked very well for the guided user-assisted design approach. By having a member of WP4 moderate, a knowledgeable expert was able to manage the discussion. Further, the repeated changes between stations ensured that fresh opinions and thoughts were repeatedly

contributed (and that no individual or idea could dominate the entire discussion). Finally, the summarization and discussion stage at the end ensured that any conflicting ideas could be discussed and resolved.

After the World Café, members of WP4 took the derived requirements from the users and created initial policy and design based on how the users envisioned working with a CI/CD pipeline. The policy then went through several rounds of discussion via email and online meetings, resolving inconsistencies and points which were insufficiently clear. The final policy is attached in Appendix A.

With the policy completed, WP4 began work on implementation. Because the users were extremely clear that they wanted to be able to work in multiple software languages, WP4 decided to create sample repositories for possible languages. One such sample repository (for Python) is attached and discussed in Appendix B.

Implementation began by cataloging the available options. Ultimately, WP4 decided to use GWDG's GitLab [Gitd] instance to host both the repositories and the CI/CD pipeline. This decision was driven by the availability of the instance, the available features, and synergies with CASTIEL2. First, the availability of the instance. The GWDG GitLab instance was already configured and professionally managed (in production). The DECICE project could have set up its own instance, but this would require project members to manage it, which was deemed to be too time-intensive and would hinder the development of the DECICE project. Alternatively, an existing commercial hosting solution could be selected, such as GitHub [Gita]. However, the cost-free tiers of these solutions had reduced capabilities compared to the GWDG GitLab instance and paid tiers would require use of project budget in order to get features already available to the project at no cost. Therefore, GWDG's GitLab instance was seen as the only reasonable option, providing an optimal balance of the need for features, use of project member time, and cost to the project. Additionally, this exploited synergies with the CASTIEL2 project, which had already decided to base their CI/CD pipeline on GitLab and was looking for a testbed for certain features. By providing this testbed, the DECICE project benefited from the CASTIEL2 project's expertise.

7.2 Features

The following were identified as critical features in the World Café and following discussions:

- Software language independence. Users anticipate using a number of different languages in the project, including Python, Golang, and Java, depending on the task being performed. Therefore, the users identified it as very important that the CI/CD pipeline function no matter what language was in use.
- Ability to work within multiple branches at various levels of stability, including a branch with all completed features, a branch with release-quality and fully tested software, and an intermediate branch for testing software before promoting it to the release branch.
- Automated testing of varying levels for different branches. For example, users specified that they would like to have all code unit tested, but would like to have machine learning models tested for degradation only before promoting code to release branches because such validation

can be very expensive.

- Ability to run most CI/CD pipeline tasks locally. Users identified this as extremely important to them, especially when it came to testing. Specifically, users identified that they had had issues in other projects where the only way to perform testing was to upload code to a CI system and that performing testing in this manner slowed or interrupted their workflow.
- Minimize manual tasks related to software release. In the software development cycle, release is a relatively rare task. Therefore, automation will ensure it is done consistently.
- Automated checking of software standards from WP1. While it would be possible for the project to manually review code for the WP1 software standards, that time could be better spent on development.

7.3 Architecture

GitLab provides a multi-stage pipeline. Inside each stage, multiple tasks may run. Additionally, tasks may run conditionally, based on many different conditions.

The final pipeline uses the following stages:

- `prepare` - short tasks which are necessary for running a build
- `test` - basic tests of functionality, such as unit tests
- `quality` - code quality checks, such as code formatting or security checks
- `build` - performs build and packaging steps
- `extra-tests` - performs extra, in-depth testing. For example, performs integration or release tests
- `upload` - uploads built code to any relevant repositories

Not all tasks run on each build. For example, when a user pushes code to a branch, it does not make sense to run the complete test suite including possibly expensive or long-running tests. Expensive or long-running tests may only be run for (for example) release builds.

As a more concrete example, the following screenshot shows a pipeline overview for a standard push to a development branch:

Figure 1 shows that four stages of the pipeline ran: `prepare`, `test`, `quality` and `build`. These are the critical stages that will run each time: `prepare` to do basic preparation tasks (like set the version number), `test` to perform basic/unit tests, `quality` to ensure the code is up to the quality standards, and `build` to compile and package the code. These basic tasks are enough for a developer to determine that their code works and meets quality standards. As an additional feature, the pipeline builds the user's code so that the user may install a packaged version of the code. A user might wish to do this in order to perform additional manual testing, to perform manual integration testing, or to develop additional tests.

After developing their feature and completing it, the user would then want to merge it to the main code-base. At this point, it makes sense to run some additional tests.

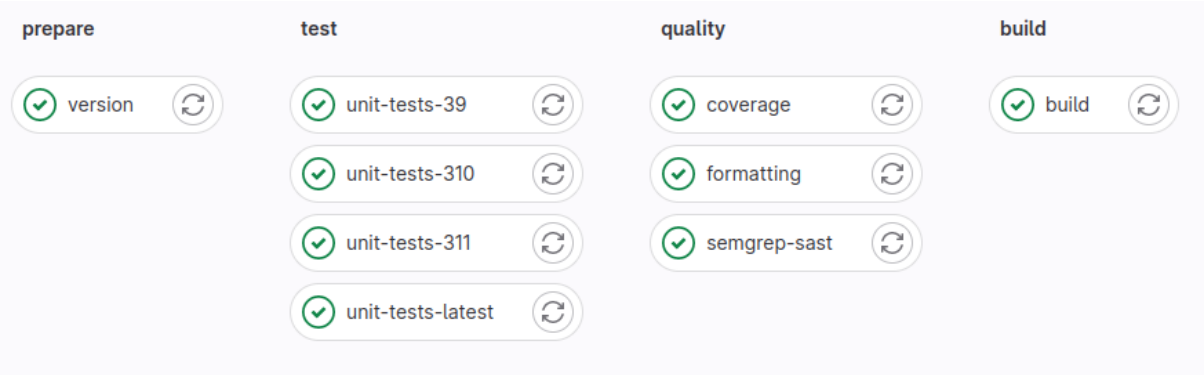

Figure 1: Development Pipeline, showing that only four stages ran: prepare, test, quality and build.

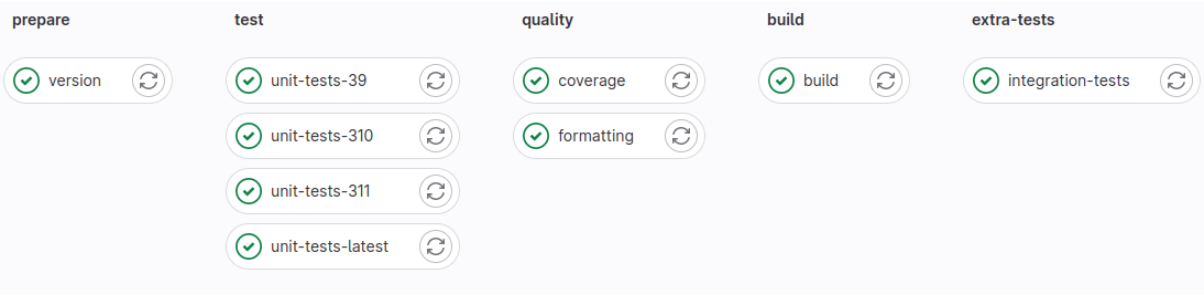

Figure 2: Development Pipeline for a Merge Request, showing that five stages ran: prepare, test, quality, build, and extra tests

Figure 2 shows that in this case all the same steps as previously ran, with one additional stage: `extra-tests`. This stage runs more in-depth tests to ensure the new code integrates correctly with the old. In this case, one task was run: `integration-tests`. The integration tests are intended to demonstrate that new features work correctly with the DECICE software as a whole, but do not go into extreme depth testing the project. In this case, the additional tests indicate that the code is likely ready to be included in the `main` branch and be used with the rest of the DECICE software.

Jumping into the future, the software will, at some point, be ready for release and use by the public.

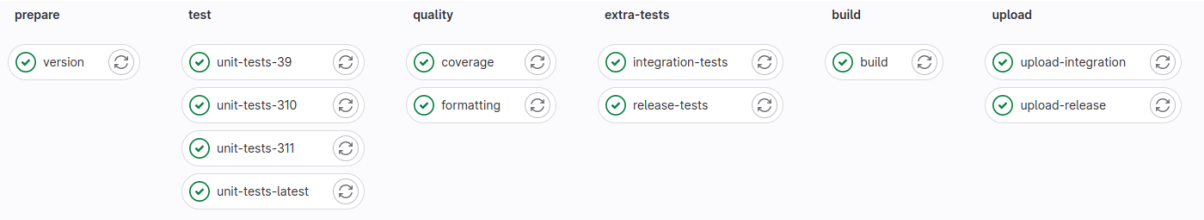

Figure 3: Development Pipeline for a tagged release, showing that all stages ran: prepare, test, quality, build, extra tests, and upload

Figure 3 shows a release build from the sample repository. Compared to the merge request build, there are several additional items that have run. First, in the `extra-tests` stage, another task has been added: `release-tests`. These tests are the complete set of all possible tests to run, no matter how expensive or detailed. These tests might include performance tests and re-training and verification of machine learning models. Passing them indicates that the software is ready for release

and is of the highest quality.

Second, a new stage is seen: `upload`. This has two tasks: `upload-integration`, which uploads the built package to a DECICE-internal package repository used by developers and to verify that all packages integrate together. This task would also be run when code is merged to the `integration` branch, so that the DECICE developers are always able to use the latest pre-release versions of all software within the package. The second task is `upload-release`, which uploads the built package to public repositories for members of the public to use. For example, this might upload a package to PyPI (for python packages) or to dockerhub (for a docker container).

Each task is calling a single bash script which performs the task. The bash scripts can then be customized to the repository and language without having to adjust the core CI/CD pipeline configuration. This prevents the CI/CD pipeline configurations from diverging across different repositories and reduces maintenance load on the developers. Additionally, it provides users with simple scripts to call in order to perform tasks in exactly the same way as the CI/CD pipeline would perform them - making it extremely easy for users to perform tests on their own.

Put together, this architecture is simple and achieves all of the critical features outlined earlier.

Finally, although none of the sample repositories make use of it, GitLab provides a framework for custom "runners". These run tasks in a specific environment. This may be very advantageous later in the project when it becomes important to test the function of code in testbed environments. For example, a custom runner could be used to allow running within a hardware-based testbed simulating a real-world deployment.

8 References

[Bra] Scott Bradner. *RFC 2119*. URL: https://www.ietf.org/rfc/rfc2119.txt (visited on 04/06/2023).

[Caf15] The World Cafe. *World Cafe Method*. The World Cafe. July 4, 2015. URL: https://theworldcafe.com/key-concepts-resources/world-cafe-method/ (visited on 05/23/2023).

[Gita] GitHub. *GitHub: Let's build from here*. GitHub. URL: https://github.com/ (visited on 05/23/2023).

[Gitb] GitLab. *Protected branches · Project · User · Help · GitLab*. GitLab. URL: https://gitlab-ce.gwdg.de/help/user/project/protected_branches (visited on 04/11/2023).

[Gitc] GitLab. *Protected tags · Project · User · Help · GitLab*. GitLab. URL: https://gitlab-ce.gwdg.de/help/user/project/protected_tags (visited on 04/11/2023).

[Gitd] GitLab. *The DevSecOps Platform*. URL: https://about.gitlab.com/ (visited on 05/23/2023).

[Pre] Tom Preston-Werner. *Semantic Versioning 2.0.0*. Semantic Versioning. URL: https://semver.org/ (visited on 04/11/2023).

[TH] Linus Torvalds and Junio Hamano. *Git*. URL: https://git-scm.com/ (visited on 04/06/2023).

A CI/CD Branches, Merging, and Review Policy

A.1 Preamble

In order to allow the use of common tools, promote consistency, and minimize the effort required for project members to work across multiple parts of the project, it is critical that the DECIE project agrees and adheres to a common layout for software repositories. The goal of this document is to define a common repository layout for all DECICE software repositories, taking into account the project requirements, language-specific quirks/requirements and general software development best practices.

A.2 Terminology

The RFC 2119 words "must", "must not", "required", "shall", "shall not", "should", "should not", "recommended", "may" and "optional", should be interpreted as defined in RFC2119 [Bra].

A.3 Abbreviations and Acronyms

- API – Application Programming Interface
- CD – Continuous Delivery
- CI – Continuous Integration
- DECICE – Device-Edge-Cloud Intelligent Collaboration framEwork
- WP – Work Package
- WPL – Work Package Leader

A.4 Exceptions, Clarifications, and Amendments

Despite the best efforts of the project, there may arise situations that are not anticipated by this document or situations where this document, other documents or policies from the DECICE project, and/or the common best practices conflict. While every effort has been to avoid such situations, it nevertheless is important to have a defined process for handling them, should they occur.

In the event of a situation not anticipated or a conflict between this document and other documents or between portions of this document, the project member who discovers such a conflict should immediately report the issue to their work package leader (WPL). It is recommended that the project member also report a suggested solution or solutions. The work package leader is then responsible for adding the issue to the agenda for the next Work Package Leader meeting.

The WPL meeting shall discuss the issue and decide on a course of action via majority vote. It is recommended that the WPL meeting approve the suggested solution of the project member who discovered the issue, as doing so will allow the project member to continue work while waiting for approval from the Work Package Leader meeting.

Additionally, should this policy require modification, it may be updated by a majority vote at the WPL meeting.

A.5 Repository Hosting

Each DECICE repository shall consist of a git [TH] repository. Each repository shall have the canonical copy located on the GWDG GitLab server under the DECICE group (`https://gitlab-ce.gwdg.de/decice`) and may be mirrored in other locations.

A.6 Repository Names

Each repository shall be named with a short, descriptive name. Such names should not be longer than five words. See Acceptable and unacceptable repository names for details and examples of clear and unclear repository names.

Repository names should be lowercase with words separated by dashes, unless referring to some external resource which uses specific capitalization or punctuation.

Repository Name	Acceptable?	Reason
edge-digital-twin	Yes	Follows guidelines for descriptiveness, capitalization and punctuation
twins	No	Insufficently descriptive
PostgreSQL-connector	Yes	Refers to an external piece of software with specific capitalization requirements

Table 3: Table 1: Acceptable and Unacceptable Repository Names

A.7 Repository Branches

Each repository shall have the following branches and shall use them for the described purpose. Each repository should have additional branches.

- Main – Primary branch for integration within a repository. Each completed feature, bugfix or other code-related task is initially merged into main. Development work should not take place directly on the main branch. The main branch is expected to represent the latest completed work within a repository, but may not yet be ready to interact with work in other repositories.
- Integration – Work in the integration branch has been merged from the main branch. It is ready to interact with work from other integration branches in other repositories.
- Release – Work in release branches is fully complete, tested, and ready for production use. Additionally, when DECICE software is released to the public under a specific version number (eg: 1.4.2, 1.3.4-RC3, etc.), a corresponding tag must be assigned to a commit in the release branch (see, "Repository Tags", below)

The "Integration" and "Release" branches shall be configured as "protected branches" [Gitb] so that no alteration may be made without a pull request. Additional, task-specific branches should be created when project members are working on a specific task. These branches should clearly identify the task they are associated with (eg: by having the same name as a bug/issue). Additionally, once work is complete and merged to the main branch, such "working branches" should be deleted.

A.8 Pull Requests

Code shall be moved between branches via opening a pull request. Each pull request shall be reviewed by at least one other project member who has significant experience with the relevant repository and/or code.

Individual work packages may add additional requirements for pull requests within repositories on which they are the primary developers. Pull requests shall include a written description of what changes were made. They shall also specifically describe any known trade offs or any place in which the code is known to be complicated or difficult to understand with a description of why this code is the best solution.

A.9 Repository Tags

Developers may use tags in any manner they wish. However, certain tags will delineate software releases. These tags shall be in Semantic Versioning 2.0.0 format [Pre]. Repositories shall be configured such that valid Semantic Versioning tags are "protected" [Gitc] tags.

Semantic Versioning tags indicating release versions shall only be applied to commits in the "Release" branch of the repository. Semantic Versioning tags indicating pre-release versions (eg: 1.2.3-RC3) shall be applied only to commits in either the "Integration" or "Release" branches.

Semantic versioning tags including build metadata (eg: 1.2.3+abcdef) may be applied to any commit.

A.10 Continuous Integration

In order to built higher quality software faster, we use a continuous integration (CI) strategy. This allows the DECICE project to consistently produce high quality software, as well as to ease the process of integrating multiple, simultaneously developed modules within that software.

In this context, "Continuous Integration" refers to the process of writing software in relatively small units and merging those units together quickly. Additionally, it includes the practice of running automated tools that check for quality issues, such as code formatters, security scanners, or measurements of unit test coverage.

A.11 Code Tests

The DECICE project categorizes tests into five categories. These are:

- unit tests – self-contained tests of individual units of functionality. These tests must be capable of running without any infrastructure and with minimal one-time setup, for example on developer machines. They run for only a short amount of time and check that each unit is working correctly. These are intended to be tests that developers run frequently on their own machines in order to verify new code they have written functions and does not break the functionality of any existing code.
- quality checks – automated tools are run against the code to ensure quality (see Automated Code Quality Checks, below). These checks are run whenever code is pushed from a developer's

system to the central repository. These test may fail within a development branch without consequence. However, when a pull request is opened to move code into a named branch (main, integration or release), the pull request must not be merged until and unless all code quality tests pass.

- basic tests – tests of this repository's functionality. These tests will typically instantiate the software from the repository and whatever infrastructure it requires. They will then interact with the software as a user or other software component might and ensure that all command lines, Application Programming Interfaces (APIs), and other methods of interacting with the software are functioning as expected. These tests may run on developer machines (and may require preexisting configuration or setup)
- integration tests – tests of how this repository interacts with other repositories. These tests typically instantiate software from this repository and from other repositories. The software then interacts with other software in order to validate that the entire system functions as expected. These tests may use significant resources and infrastructure.
- release tests – tests which verify that the software in the repository is production ready and can be deployed in cooperation with other deployed release software from the DECICE project. The most intensive tests are made against release software – for example, for performance, training time, cost, etc.

A.12 Automated Code Quality Checks

Work Package (WP) 4 supports Work Package 1 by automating the code quality checks specified by WP1.

A.13 Continuous Delivery

Continuous Delivery refers to the practice of (semi-)automatically releasing software to users. Combined with fast development and Continuous Integration, it may allow users faster access to new features and greatly increase the ability of the DECICE project to meet the needs of its users.

The DECICE project implements Continuous Delivery via the release branch of each repository. When a tag is made in the release branch, the software is automatically published via whatever means the project determines is most appropriate for that software. For example, for a containerized service (eg: one in a Docker container), the software may be built, packaged into the container, then pushed into a container registry. As another example, a software package intended for installation directly on a computer (e.g.: one packaged as a .deb, .rpm, or in another software manager format) might be built, packaged, and then published within a package repository run by the DECICE project itself.

A.14 Appendix

A.14.1 Required files and interfaces

This appendix shall be considered binding. This appendix defines the required files and interfaces that must be in every DECICE repository in order for the CI/CD configuration detailed in the main body of this document to function.

- .gitlab-ci.yml – describes the CI pipeline for this repository. Contents consistent (if not identical) across repositories
- VERSION – A file containing the current version of the software
- tests/ - A directory containing scripts to run tests
  - unit.sh – Runs unit tests on the local machine. Passing no arguments must run all tests, returning a non-zero exit code if any test fails. Passing "help" or "–help" must result in output describing how to use the tool. The results of passing any other arguments are repository dependent and intended only for use by developers, not as part of the CI/CD pipeline.
  - integration.sh – Runs integration tests in the CI environment. Passing no arguments must run all tests, returning a non-zero exit code if any test fails. The results of passing any other arguments are repository dependent and intended only for use by developers, not as part of the CI/CD pipeline.
  - release.sh – Runs integration tests in the CI environment. Passing no arguments must run all tests, returning a non-zero exit code if any test fails. The results of passing any other arguments are repository dependent and intended only for use by developers, not as part of the CI/CD pipeline.
  - coverage.sh – Runs code coverage checks. Exits with 0 if the threshold for code coverage in unit tests is met. Otherwise exits 1
- build.sh – performs a full build and package of the software, placing the results in a directory named dist. If no argument is passed, will use the version in the version file with a git commit hash appended. If an argument is passed, the argument shall be used as the version number.
- release.sh – performs the task of releasing the software, based on the built and packaged software in dist/. This script is expected to run in the CI/CD environment only.

A.14.2 Step by Step Workflow Example

This appendix shall be considered informational. In case of any conflict between this appendix and the main body of the document, the text of the main body of the document shall be considered correct.

The purpose of this appendix is to step through portions of the development process in order to give examples of how the strategy outlined in the main body of this document functions and ways in which this can facilitate good development.

Let's start with a DECICE developer named Dave. Dave needs to add certain functionality to the AI scheduler to better handle network interruptions.

Dave starts by visiting the DECICE GitLab in order to check out the AI scheduler. He finds three repositories related to the AI scheduler: ai-scheduler-core, ai-scheduler-api, ai-scheduler-training-data. Based on his previous knowledge, he knows he will be working with the core of the AI scheduler and checks out the ai-scheduler-core repository.

Dave also knows he cannot simply do his development on the main branch of the repository. Therefore, he creates a branch with the name of the ticket assigned to him – AISCHED-1234. He can do this either through the GitLab web interface or git command line tools on his workstation.

Now Dave is ready to begin development. He creates a few example scenarios to validate the scheduler is performing as expected with his modifications, then writes the code necessary for the feature.

Once the code is written, Dave performs manual testing of the scheduler on his workstation and validates that it behaves as expected. He runs the unit tests in the repository and verifies that they all pass.

Dave then commits his code to the git repository, pushes it, and begins to prepare a pull request to merge his changes into main.

But everything is not correct – the CI pipeline fails when checking his pull request! Dave forgot to add new unit tests for his code and the threshold for unit test coverage is no longer satisfied. Merging Dave's code to the main branch at this point would make it more difficult for other developers to ensure they haven't broken any functionality when working on the repository and thereby reduce overall code quality.

Fortunately, Dave already has the test cases he prepared earlier. He creates unit tests based on them and runs the code coverage check on his workstation. The code coverage is now above the threshold again, so Dave commits his new tests and pushes his changes to GitLab.

Unfortunately, when he looks at his most recent commit on GitLab, it shows that the unit tests have failed. Dave forgot to add a required dependency for his unit tests, so they can't run. He quickly fixes this by adding the exact version of the package he used to the project dependencies and pushes again.

This time, the unit tests work, but the security check fails. The version Dave used has a known critical security issue and can't be used in the DECICE project. Fortunately, it is fixed in the next version of the package, so Dave adjusts his dependencies, checks the unit tests locally, commits, pushes, and looks at the CI results. They're passing.

Now Dave can create the pull request. He writes a detailed explanation of what he changed and why, picks two other members of the project he knows are familiar with the ai-scheduler-core codebase to be reviewers, and creates the pull request.

The two reviewers examine Dave's commits, ask some questions, and approve his merge request, confident that because of the automated unit tests and security checks, his code works as expected

and creates no known vulnerabilities.

The next week, Larry, Dave's work package leader, is asked to merge Dave's code into the integration branch. Dave's code is needed for work to progress in the edge-digital-twin repository. Larry creates a pull request to merge the changes in the main branch into the integration branch. Automated tests run, checking Dave's code (and the other changes being merged) more thoroughly, for interaction with code from other repositories. Larry is relieved to see that Dave's code handles the additional checks well. If it had not, he would have needed Dave to make adjustments to his code.

The pull request is merged. At this point, the CI system sees a new integration commit and builds an integration version of the repository. This receives a version number like "1.4.3-RC3-acedbe", indicating it is an automated, non-production build. This build is pushed to a DECICE-internal repository so that other packages may use it in their integration tests.

Development in the edge-digital-twin repository continues and it is soon time to make a release. Larry merges all the changes in the integration branch into the release branch, while other work package leaders to the same for the repositories they manage. Larry watches the tests run and checks the performance and the expected cost of training the AI model. Although it wasn't a goal of Dave's changes, Larry is glad to see that Dave's changes reduced the estimated training time for the AI model. Larry is also glad to see the tests pass.

With the changes merged into the release branch and all the tests passing, Larry is ready to make a new release. He does so by adding a tag to the most recent commit in the release branch. This tag is formatted as a semantic-versioning-compatible version number. In this case, 1.4.2-RC3. When the CI system observes this tag, it begins a production build of the repository. Once that build is complete, it automatically pushes the build to a location where the public at large may use it.

B CI/CD Code Listing

This appendix consists of two parts. First, a listing of the basic files included in every repository, along with a description of their general purpose. Second, code listings of content for files relevant to the CI/CD pipeline, along with a short description of the purpose of that file. Please note that these files have been taken from the sample repository for Python and may contain Python-specific references. Additionally, please also note that python-specific files have been excluded.

B.1 File Tree

```
repository root
├── .gitlab
│   ├── issue_templates
│   └── merge_request_templates
├── test
│   ├── coverage.sh
│   ├── integration.sh
│   ├── release.sh
│   └── unit.sh
├── .gitignore
├── .gitlab-ci.yml
├── .pre-commit-config.yaml
├── build.sh
├── README.md
├── set_version.sh
└── VERSION
```

B.2 File Listings

B.2.1 .gitlab-ci.yml

The heart of the CI/CD pipeline, `.gitlab-ci.yml` controls the stages of the CI/CD pipeline, the order in which they run, and the commands run.

Listing 1: .gitlab-ci.yml

```
1 # You can override the included template(s) by including variable overrides
2 # SAST customization: https://docs.gitlab.com/ee/user/application_security/
    ↪ sast/#customizing-the-sast-settings
3 # Secret Detection customization: https://docs.gitlab.com/ee/user/
    ↪ application_security/secret_detection/#customizing-settings
4 # Dependency Scanning customization: https://docs.gitlab.com/ee/user/
    ↪ application_security/dependency_scanning/#customizing-the-dependency-
    ↪ scanning-settings
5 # Container Scanning customization: https://docs.gitlab.com/ee/user/
    ↪ application_security/container_scanning/#customizing-the-container-
    ↪ scanning-settings
6 # Note that environment variables can be set in several places
7 # See https://docs.gitlab.com/ee/ci/variables/#cicd-variable-precedence
```

```
stages:
- prepare
- test # Unit tests
- quality # Code quality checks
- build # Perform actual build of the software
- extra-tests # Extra tests for integration and release
- upload # Upload step for integration and release branches

sast:
  stage: quality
include:
- template: Security/SAST.gitlab-ci.yml

# Default run rules
.default_rules:
  rules:
    - if: $CI_PIPELINE_SOURCE == 'merge_request_event'
    - if: $CI_COMMIT_BRANCH # Any branch

# Updates the VERSION file for our build
version:
  stage: prepare
  image: python:latest
  script:
    - ./set_version.sh
    - cat VERSION
  artifacts:
    paths:
    - VERSION
  rules:
    - !reference [.default_rules, rules]

# Unit tests with the latest version of python
unit-tests-latest:
  stage: test
  image: python:latest
  script:
    - ./test/unit.sh
  rules:
    - !reference [.default_rules, rules]

# Unit tests with python 3.11
unit-tests-311:
```

```
51    stage: test
52    image: python:3.11-slim
53    script:
54      - ./test/unit.sh
55    rules:
56      - !reference [.default_rules, rules]
57
58  # Unit tests with python 3.10
59  unit-tests-310:
60    stage: test
61    image: python:3.10-slim
62    script:
63      - ./test/unit.sh
64    rules:
65      - !reference [.default_rules, rules]
66
67  # Unit tests with python 3.9
68  unit-tests-39:
69    stage: test
70    image: python:3.9-slim
71    script:
72      - ./test/unit.sh
73    rules:
74      - !reference [.default_rules, rules]
75
76  # Unit test code coverage check
77  coverage:
78    stage: quality
79    image: python:latest
80    coverage: '/(?i)total.*? (100(?:\.0+)?\%|[1-9]?\d(?:\.\d+)?\%)$/'
81    script:
82      - ./test/coverage.sh
83    artifacts:
84      reports:
85        coverage_report:
86          coverage_format: cobertura
87          path: coverage.xml
88    rules:
89      - !reference [.default_rules, rules]
90
91  # Code formatting/consistency check
92  formatting:
93    stage: quality
```

```
94    image: python:latest
95    script:
96      - pip install -r requirements-dev.txt
97      - black --check .
98      - flake8 --exclude .venv/*
99    rules:
100     - !reference [.default_rules, rules]
101
102 integration-tests:
103   stage: extra-tests
104   image: python:latest
105   rules:
106     - !reference [.default_rules, rules]
107     - if: $CI_MERGE_REQUEST_TARGET_BRANCH_NAME == "integration" # Merge request
          ↪  to integration branch
108     - if: $CI_MERGE_REQUEST_TARGET_BRANCH_NAME == "release" # Merge request to
          ↪ release branch
109     - if: $CI_COMMIT_TAG =~ /^(\d+\.)?(\d+\.)?(\*|\d+)$/ # SemVer release tags
110   script:
111     - ./test/integration.sh
112
113 build:
114   stage: build
115   image: python:latest
116   script:
117     - ./build.sh
118   artifacts:
119     paths:
120     - dist/*
121   rules:
122     - !reference [.default_rules, rules]
123     - if: $CI_PIPELINE_SOURCE == 'merge_request_event'
124
125 upload-integration:
126   stage: upload
127   rules:
128     - if: $CI_COMMIT_BRANCH == "integration" # Integration branch commit
129     - if: $CI_COMMIT_BRANCH == "release" # release branch commit
130     - if: $CI_COMMIT_TAG =~ /^(\d+\.)?(\d+\.)?(\*|\d+)$/ # SemVer release tags
131   image: python:latest
132   script:
133     - pip install build twine
134     - python -m build
```

```
135     - TWINE_PASSWORD=${CI_JOB_TOKEN} TWINE_USERNAME=gitlab-ci-token python -m
            ↪ twine upload --verbose --repository-url ${CI_API_V4_URL}/projects/${
            ↪ CI_PROJECT_ID}/packages/pypi dist/*
136
137
138 release-tests:
139   stage: extra-tests
140   image: python:latest
141   rules:
142     - if: $CI_COMMIT_BRANCH == "release" # release branch commit
143     - if: $CI_MERGE_REQUEST_TARGET_BRANCH_NAME == "release" # Merge request to
            ↪ release branch
144     - if: $CI_COMMIT_TAG =~ /^(\d+\.)?(\d+\.)?(\*|\d+)$/ # SemVer release tags
145   script:
146     - ./test/release.sh
147
148 upload-release:
149   stage: upload
150   rules:
151     - if: $CI_COMMIT_TAG =~ /^(\d+\.)?(\d+\.)?(\*|\d+)$/ # SemVer release tags
152   image: python:latest
153   script:
154     - pip install build twine
155     - python -m build
156     - TWINE_PASSWORD=${PYPI_API_KEY} TWINE_USERNAME=__token__ python -m twine
            ↪ upload --verbose dist/*
```

B.2.2 build.sh

`build.sh` performs the build of the software. This version is Python-specific. Another language would have differnt contents. However, by using a script as the entry point, users may call the script no matter which repository and langauge they are working in and know that it will result in a built package. Additionally, the use of a script ensures the CI/CD system need not be adjusted for each and change to the build process, isolating the CI/CD pipeline from these changes.

Listing 2: build.sh

```
1 #! /bin/bash
2
3 ./set_version.sh
4 mkdir -p dist
5 pip wheel --no-deps -w dist .
6 pip freeze > dist/pip-reproducable.txt
```

B.2.3 set_version.sh

`build.sh` manipulates the `VERSION` file at build time. In most cases, it takes the current version number and appends a short hash identifier. This marks the build as a local (non-authoritative) build. Builds on developer computers as well as CI/CD builds which are not tagged with a Semantic Versioning number are marked this way. Only builds where a version number id directly specified or is specified via a git tag are build with non-local Semantic Versioning numbers; it is assumed these will be release builds.

Listing 3: set_version.sh

```
#! /bin/bash
set -x
# Store the current version, tossing any local version
VERSION=$(cat VERSION | grep -oE '^([[:digit:]]+\.)*[[:digit:]]+')

if [ -n "$CI" ]
then
    if [ -n "$( echo $CI_COMMIT_TAG | grep -oE '^([[:digit:]]+\.)*[[:digit:]]+$
        ↪ ')" ]
    then
        # If statement checks if the tag is a SemVer version number
        # If there's a tag, use the number from that.
        echo -n $CI_COMMIT_TAG > VERSION
    else
        # Otherwise, git hash on current version (local-ish build)
        echo -n "$VERSION+$CI_COMMIT_SHORT_SHA" > VERSION
    fi
else
    if [ -n "$1" ]
    then
        # If passed an argument, use that
        echo -n "$1" > VERSION
    else
        # Otherwise, append commit hash
        echo -n "$VERSION+$(git rev-parse --short HEAD)" > VERSION
    fi
fi
```

B.2.4 VERSION

Contains the base semantic versioning number.

B.2.5 test/coverage.sh

Runs unit tests with test coverage measurement turned on. This script must exit with a non-zero (i.e.: error) exit code if coverage is below the threshold set by WP1. In this instance, that threshold is set in `tox.ini`, which is a python-specific file and is omitted from this listing.

Listing 4: test/coverage.sh

```
#! /bin/bash

if [ -z "$CI" ] # $CI is set in GitLab CI environments, where we don't need to
    ↪ use a virtualenv
then
    # Change to the root of the repository
    pushd .
    cd $(git rev-parse --show-toplevel)

    if [ ! -d .venv ] # Check for an existing virtualenv
    then
        # Build a virtualenv if none exists
        python3 -m venv .venv
        . .venv/bin/activate
        pip install -r requirements.txt -r requirements-dev.txt
    fi

    . .venv/bin/activate # load the virtualenv
else
    # In the CI environment, just do the install
    pip install -r requirements.txt -r requirements-dev.txt
fi

tox -e coverage -- "$@"
RES=$?
[ -z "$CI" ] && deactivate # Don't run deactivate if we didn't build a
    ↪ virtualenv
[ -z "$CI" ] && popd # return the user to the expected directory
exit $RES
```

B.2.6 test/unit.sh

Runs unit tests. This script must exit with a non-zero (i.e.: error) exit code if any unit test fails.

Listing 5: test/unit.sh

```
#! /bin/bash

```

```
if [ -z "$CI" ] # $CI is set in GitLab CI environments, where we don't need to
    ↪ use a virtualenv
then
    # Change to the root of the repository
    pushd .
    cd $(git rev-parse --show-toplevel)

    if [ ! -d .venv ] # Check for an existing virtualenv
    then
        # Build a virtualenv if none exists
        python3 -m venv .venv
        . .venv/bin/activate
        pip install -r requirements.txt -r requirements-dev.txt
    fi

    . .venv/bin/activate # load the virtualenv
else
    # In the CI environment, just do the install
    pip install -r requirements.txt -r requirements-dev.txt
fi

tox --skip-env coverage -- "$@"
RES=$?
[ -z "$CI" ] && deactivate # Don't run deactivate if we didn't build a
    ↪ virtualenv
[ -z "$CI" ] && popd # return the user to the expected directory
exit $RES
```

B.2.7 test/integration.sh

`integration.sh` runs integration tests for a repository and exists with a non-zero (i.e.: error) code if any test fails.

This file is customized to each repository. At the time this report was completed, no sensible environment existed for running integration tests with the example Python repository and therefore no code listing is provided here.

B.2.8 test/release.sh

`release.sh` runs integration tests for a repository and exists with a non-zero (i.e.: error) code if any test fails.

This file is customized to each repository. At the time this report was completed, no sensible environment existed for running release tests with the example Python repository and therefore no code listing is provided here.

B.2.9 .gitlab/issue_templates/bug_report.md

Contains a template that GitLab displays to the user when filing a bug report. This is a good way to ensure that the reporter of a bug is prompted for all the relevant information.

Listing 6: .gitlab/issue_templates/bug_report.md

```
---
name: Bug report
about: Create a bug report to help us improve
title: "Bug Summary"
labels: "bug"
assignees: ""
---

**Describe the bug**

<!-- A clear and concise description of what the bug is. -->

**To Reproduce**

Steps to reproduce the behavior:

1. ...
2. ...
3. ...

**Expected behavior**

<!-- A clear and concise description of what you expected to happen. -->

**System [please complete the following information]:**

- OS: e.g. [Ubuntu 18.04]
- Language Version: [e.g. Python 3.8]
- Virtual environment: [e.g. Conda]

**Additional context**

<!-- Add any other context about the problem here. -->
```

B.2.10 .gitlab/merge_request_templates/pull_request_template.md

Contains a template that GitLab displays to the user when opening a pull request. This file prompts developers to provide all information required in the policy for merge requests.

Listing 7: .gitlab/merge_request_templates/pull_request_template.md

```
<!-- Many thanks for contributing to this project! -->

**PR Checklist**

<!-- Please fill in the appropriate checklist below (delete whatever is not
    ↪ relevant). These are the most common things requested on pull requests (
    ↪ PRs). -->

- [ ] This comment contains a description of changes (with reason)
- [ ] Referenced issue is linked
- [ ] If you've fixed a bug or added code that should be tested, add tests!
- [ ] Documentation in `docs` is updated
- [ ] `CHANGELOG.rst` is updated

**Description of changes**

<!-- Please state what you've changed and how it might affect the user. -->

**Technical details**

<!-- Please state any technical details such as limitations, reasons for
    ↪ additional dependencies, benchmarks etc. here. -->

**Additional context**

<!-- Add any other context or screenshots here. -->
```