COSMOS

DevOps for Complex Cyber-physical Systems

Project Number 957254

D3.3 Automated bad practice detectors for CPS DevOps pipelines

Version 1.0
29 June 2022
Final

Public Distribution

University of Sannio

Project Partners: **Aicas, Delft University of Technology, GMV Skysoft, Intelligentia, Q-media, Siemens, Siemens Healthcare, The Open Group, University of Luxembourg, University of Sannio, Unparallel Innovation, Zurich University of Applied Sciences**

Project Partner Contact Information

Aicas James Hunt Emmy-Noether-Strasse 9 76131 Karlsruhe Germany Tel: +49 721 663 968 0 E-mail: jjh@aicas.com	**Delft University of Technology** Annibale Panichella Van Mourik Broekmanweg 6 2628 XE Delft Netherlands Tel: +31 15 27 89306 E-mail: a.panichella@tudelft.nl
Intelligentia Davide De Pasquale Via Del Pomerio 7 82100 Benevento Italy Tel: +39 0824 1774728 E-mail: davide.depasquale@intelligentia.it	**GMV Skysoft** José Neves Alameda dos Oceanos Nº 115 1990-392 Lisbon Portugal Tel. +351 21 382 93 66 E-mail: jose.neves@gmv.com
Q-media Petr Novobilsky Pocernicka 272/96 108 00 Prague Czech Republic Tel: +420 296 411 480 E-mail: pno@qma.cz	**Siemens** Birthe Boehm Guenther-Scharowsky-Strasse 1 91058 Erlangen Germany Tel: +49 9131 70 E-mail: birthe.boehm@siemens.com
Siemens Healthineers David Malgiaritta Siemensstrasse 3 91301 Forchheim Germany Tel: +49 9191 180 E-mail: david.malgiaritta@siemens-healthineers.com	**The Open Group** Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of Sannio Massimiliano Di Penta Palazzo ex Poste, Via Traiano I-82100 Benevento Italy Tel: +39 0824 305536 E-mail: dipenta@unisannio.it	**University of Luxembourg** Domenico Bianculli 29 Avenue J. F. Kennedy L-1855 Luxembourg Luxembourg Tel: +352 46 66 44 5328 E-mail: domenico.bianculli@uni.lu
Unparallel Innovation Bruno Almeida Rua das Lendas Algarvias, Lote 123 8500-794 Portimão Portugal Tel: +351 282 485052 E-mail: bruno.almeida@unparallel.pt	**Zurich University of Applied Sciences** Sebastiano Panichella Gertrudstrasse 15 8401 Winterthur Switzerland Tel: +41 58 934 41 56 E-mail: panc@zhaw.ch

Document Control

Version	Status	Date
0.1	Document outline	30 March 2022
0.2	Introduction and Design Draft	26 April 2022
0.3	Draft of the proposed Tools	17 May 2022
0.4	First full draft	17 June 2022
0.5	Further editing draft	22 June 2022
0.6	QA review	28 June 2022
1.0	Final Version	29 June 2022

Table of Contents

Executive Summary

Setting up a Continuous Integration and Delivery (CD) pipeline has important advantages for software development, allowing for early discovery of defects, faster release cycles, and increased productivity. In the context of Cyber-Physical Systems (CPSs), the usage of CI/CD is even more important, especially because of the complex environments, the intensive usage of simulators, and the diverse targeted hardware devices. At the same time, difficulties in automating builds in such complex environments, and the need for deploying the software onto hardware or simulators make CI/CD adoption even more challenging. As a follow-up of our previous investigation on challenges, barriers, and mitigation strategies related to the adoption of CI/CD for CPS development in the industry and open source (D3.2), in this deliverable we describe three tools for CI/CD bad practice detection, featuring detection rules particularly suitable for CPS development. Specifically, we have developed (i) CI/CD CONFIGURATION ANALYZER, a tool to detect bad practices in CI/CD configurations, (ii) BUILD LOG ANALYZER, a tool that analyzes the CI/CD at run-time by mining its logs to detect CI/CD misuses, and (iii) a tool for the identification and optimization of fat containers. The detection infrastructure, developed to support the GitLab CI/CD technology (i.e., as project' decision), can be easily expanded to support further detectors based on specific organizational needs. Due to the specificity of different CI/CD frameworks, it is important to highlight that, to support a different technology it is required to customize the existing detection rules in order to fit those specificity. Finally, all tools will be released as open-source under the MIT license.

1 Introduction

The usage of Continuous Integration (CI) and Delivery (CD) in software development entails an early discovery of defects [11], as well as increased productivity [28] and fast release cycles [6]. However, a proper application of CI/CD requires to fulfill suitable principles, as well as properly setup of the software and hardware infrastructure [5, 6, 12]. Difficulties in doing that have been pointed out as major barriers for CI/CD adoptions in software projects [10, 19].

Looking into specific problems and challenges related to CPSs, in a previous deliverable (i.e., D3.2) [27] we have analyzed challenges encountered by the industry and open source when setting up CI/CD pipelines for CPSs. First and foremost, developers have to cope with a very complex environment, making use of simulators and hardware, and in some cases, the portability of complex environments towards CI/CD pipelines or for deployment purposes is achieved through containerization. Also, there is a need for deploying software onto diverse hardware and software environments. In this context, design and implementation choices might cause the build outcome to be inconsistent among these environments. Finally, the complex environments and the need for deploying onto hardware or simulator may tend to cause a decay of the build time as the software evolves, or be the cause of flakiness.

Following up the CI/CD bad practices catalog elicited in previous research [36], and, above all, following up our previous investigation concerning challenges, barriers, and related mitigation strategies in adopting CI/CD for CPSs [27], in this deliverable, we introduce an infrastructure aimed at supporting developers in the detection of bad practices and decay in CI/CD pipelines. The infrastructure is composed of the following three components:

1. A *CI/CD Configuration Analyzer* which extracts facts from the CI/CD configuration files and leverages rules to detect bad choices in the pipeline configuration.
2. A *Build Log Analyzer* that analyzes CI/CD build logs with the aim of identifying decays in the pipeline by observing its run-time;
3. A *Fat Docker detection and de-bloating*, which determines the extent to which images used for containerizing run-time environments contain unused components/libraries, hence slowing down the download and usage of images in a CI/CD pipeline.

The CI/CD CONFIGURATION ANALYZER and the BUILD LOG ANALYZER have been implemented to support GitLab CI/CD framework, although they can be ported to other technologies by customizing the currently implemented detection strategies taking into account the specificity of the new technology.

Both infrastructures provide the implementation of some bad practice detectors, chosen based on what emerged as particularly relevant from previous literature and the analysis of CPS CI/CD bad practices made in the D3.2 deliverable [27]. Specifically, the CI/CD CONFIGURATION ANALYZER provides detectors for manual triggering, build retry, and hiding failures, whereas the BUILD LOG ANALYZER analyzes (i) the extent to which the release branch is broken and for how long, (ii) whether jobs are skipped because of their failures, (iii) whether the build time exhibits an uptrend, and/or the build exhibits a duration significantly longer than previous builds, (iv) improper cache handling, (v) inconsistent build behavior over multiple environments, and (vi) suboptimal build ordering due to frequently-failing stages occurring at a later stage of a build.

Noteworthy, as it is detailed in this deliverable, both tools provide a generic infrastructure with supporting APIs, allowing to add further detectors other than those already implemented.

The FAT CONTAINER DETECTOR leverages the Docker containerization technology, and, given an image, analyzes (i) its installed components/libraries as well as their inter-dependencies, and (ii) the library usages from source code and build automation scripts. Then, it tries to remove (groups of) dependencies so that the image execution and tests do not fail. Finally, it provides the user with (i) information about the extent to which

the images contain unused components/libraries, (ii) the list of components/libraries that can be removed, and (iii) the flattened, reduced image.

The tools are available at URL `https://cosmos-devops.cloudlab.zhaw.ch/cosmos-devops/cosmos-devops-internal/-/tree/master/WP3/BadPracticeDetector/`, and will be released as open-source under the MIT License.

The rest of this deliverable is organized as follows. Section 2 provides some background notions about (i) the CI/CD technology leveraged in our implementation, i.e., GitLab CI/CD; and (ii) containerization principles and technology, i.e., Docker. Section 3 provides an overview of the related literature, while Sections 4–6 describe the three tools, providing their functionality, architecture, and design choices, as well as the user manual together with a usage example. Finally, Section 7 concludes the deliverable.

2 Background

This section provides background information aimed at better contextualizing the provided bad practices detector tools. Specifically, we start by describing what is the GitLab Continuous Integration (GitLab CI/CD) framework focusing more on the key aspects that give the possibility to properly adapt the CI/CD process to specific organizational needs. After that, we introduce and detail the key features of containerization with Docker.

2.1 GitLab CI/CD

Before going deeper into the GitLab CI/CD framework it is important to highlight what we mean when talking about continuous integration, continuous delivery, and continuous deployment. **Continuous Integration (CI)** is a software development practice used to BUILD AND TEST software every time a developer pushes changes to the versioning system (even to development branches). Based on its definition, CI is usually enacted several times per day. This practice helps decrease the chances that errors are introduced in the software under development, as well as, allows early discovery of defects as soon as they are introduced in the codebase. In other words, CI allows ensuring that the incoming changes pass all tests, as well as, adhere to organizational guidelines and code compliance standards. **Continuous Delivery (CD)**, instead, is a software engineering approach built on top of continuous integration where not only the system is built and tested each time a code change is pushed to the codebase, but the system is also DEPLOYED CONTINUOUSLY. It is important to note that, when relying on Continuous Delivery, the deployment to production is still strategically defined and very often triggered manually. Finally, **Continuous Deployment**, similarly to CI and CD, is a software development practice in which every code change goes through the entire pipeline and is PUT INTO PRODUCTION AUTOMATICALLY. The latter allows having many production deployments per day. The difference between Continuous Delivery and Continuous Deployment is that for the latter the whole process is fully automated, meaning that there is no human intervention at all.

GitLab CI/CD is a GitLab framework giving the possibility to use all of the continuous methods (Continuous Integration, Delivery, and Deployment). In other words, by relying on GitLab CI/CD an organization can automatically test, build, and publish their software systems without needing to rely on external third-party applications. Figure 1 shows the common steps (i.e., workflow) in the GitLab process, where the features at each stage of the DevOps lifecycle are highlighted. Specifically, the process can be enacted by discussing a specific code feature/defect in an issue and working locally on the proposed changes. After that, once the changes are "ready" to be submitted, it is possible to push the changes to a feature branch in a remote repository hosted in GitLab. The push triggers the CI/CD pipeline of the project under development. At this point, GitLab CI/CD starts running (sequentially or in parallel) a set of automated scripts included in the CI/CD configuration file to: (i) build and test the system and (ii) preview the changes in a Review App. As soon as the developers have a guarantee that the proposed changes work as expected (the code changes are approved by the reviewer

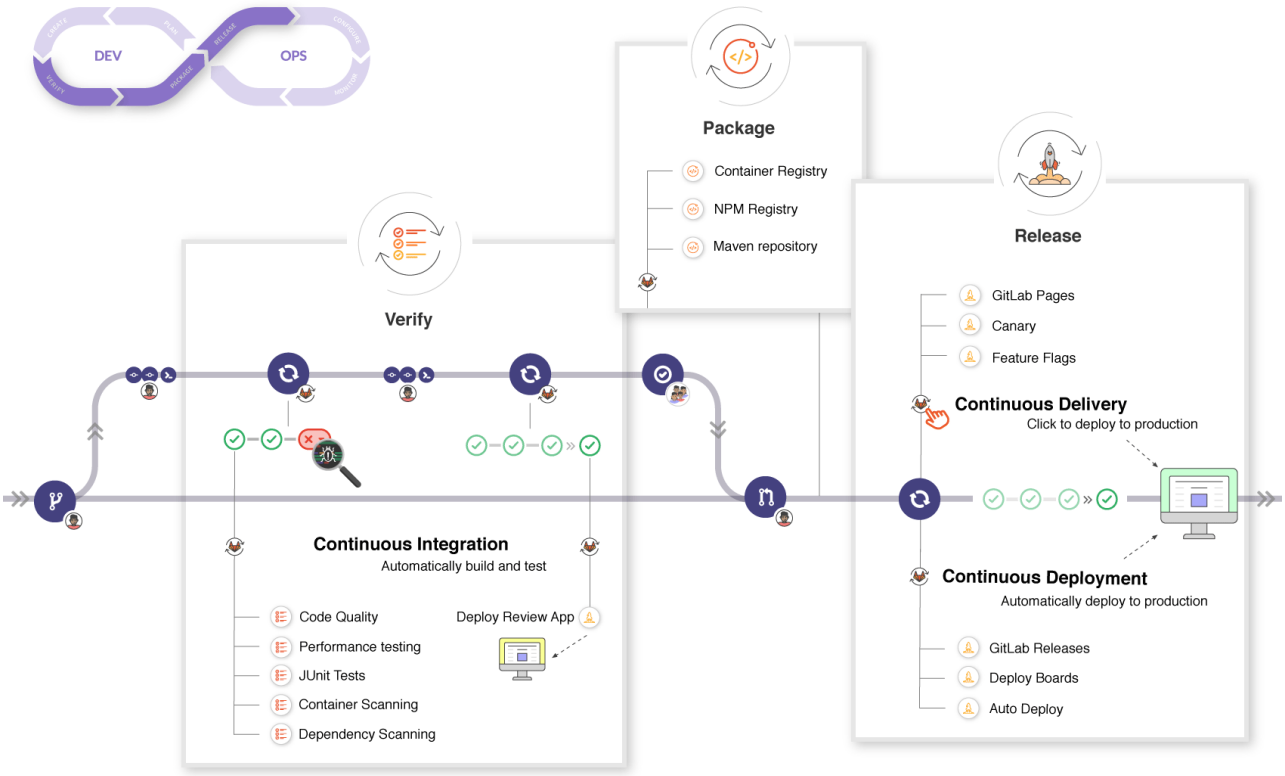

Figure 1: GitLab CI/CD workflow example [2]

teams), the GitLab CI/CD framework merges the feature branch into the default branch. At the same time, GitLab CI/CD automatically deploys the approved changes into a production environment. Of course, if there are some errors in the process, GitLab allows to roll back the proposed changes.

GitLab CI/CD uses several concepts to describe and run a build and deploy it into a production environment that we are going to summarize in the following.

PIPELINES are the top-level component of continuous integration, delivery, and deployment. A pipeline is a group of jobs (defining what to do) that get executed in stages (defining when to run). All the jobs belonging to a stage are executed in parallel, and if all the jobs succeed, the pipeline moves on to the next stage, otherwise, the next stage is not (usually) executed. Very often, pipelines are executed automatically, however, there are also cases where a developer can manually interact with a pipeline. The default and comprehensive pipeline is made up of four different stages: (i) a build stage usually running a COMPILE job, (ii) a test stage running jobs (e.g., TEST) aimed at executing different testing activities, (iii) a staging stage, with a job called DEPLOY-TO-STAGE, and (iv) a production stage, running a job DEPLOY-TO-PROD to automatically deploy into a production environment.

STAGES can be used by jobs and are usually defined globally. By specifying the stages it is possible to have flexible multi-stage pipelines. The ordering of the elements in a stage defines the ordering of the jobs' execution. Specifically, jobs belonging to the same stage are usually run in parallel, while jobs belonging to the next stage are run after the jobs from the previous stage end successfully. For the latter, there can be some exceptions such as a case where a job belonging to a stage is allowed to fail without having a failure in the overall CI/CD process.

JOBS are the most fundamental element of a `.gitlab-ci.yml` file characterized by an arbitrary name and containing at least the `script` clause. Jobs are usually defined with constraints stating when (i.e., under which circumstances/conditions) they should be run. It is possible to specify an unlimited number of jobs, which are picked up by Runners and executed within the environment of the Runner. When specifying the set

of jobs for a pipeline it is important to ensure that each job must be run independently from the others, at least for the ones belonging to the same stage.

CI/CD VARIABLES are environment variables mainly used to control the execution order and the behavior of both jobs and pipelines, guarantee re-usability, and avoid hard-coding values in the CI/CD configuration file. GitLab CI/CD has a default set of predefined CI/CD variables that can be used in pipeline configuration and job scripts. As an example, the `CI_JOB_STAGE` predefined variable is used to access the name of the stage to which a specific job belongs to.

ENVIRONMENTS are mainly used to describe where the system has to be deployed. In other words, each time GitLab CI/CD deploys a version of the system to an environment, a "deployment" is created. Note that, GitLab CI/CD is able to track the whole history of the deployments to all the environments being specified. In this way, it is possible to know what is the current version of the system actually deployed to a specific environment. To create an environment to deploy in the `.gitlab-ci.yml` it is needed to specify a name for the environment and optionally, a URL, which determines the deployment URL. As an example, Figure 2 shows an excerpt for the creation of the environment *staging* belonging to the stage *deploy*, and specifically to the job *deploy_staging*.

```yaml
deploy_staging:
  stage: deploy
  script:
    - echo "Deploy to staging server"
  environment:
    name: staging
    url: 'https://staging.example.com'
```

Figure 2: Example excerpt of GitLab environment configuration

CACHING. A cache is one or more files a job downloads and saves. Subsequent jobs that use the same cache do not have to download the files again, so they execute more quickly. It is a good practice to use the caching feature for dependencies management such as to avoid downloading more than once the same packages from the internet. It is important to report some assumptions that must be known to properly use the caching feature by GitLab CI/CD: (i) subsequent pipelines can use the cache; (ii) subsequent jobs in the same pipeline can use the cache if the dependencies are identical, and (iii) different projects cannot share the cache.

GITLAB RUNNERS are applications that work with GitLab CI/CD to run jobs in a pipeline. By using a Runner it is possible to (i) run multiple jobs concurrently, (ii) use multiple tokens[1] with multiple servers, and (iii) limit the number of concurrent jobs per token. Some of the benefits of relying on GitLab runners are: (i) they are available as binaries without any other requirements, (ii) they are compatible with multiple Operating Systems, and (iii) they allow to properly customize the job running environment and the configuration is automatically reloaded without restart, and (iv) they enable caching of Docker containers.

2.2 Docker

A container is a standard unit of software that packages up code and all its dependencies and allows to run the application quickly and reliably in any environment. Thus, a container encapsulates an application with its dependencies, source code, files, and environment variables to provide an isolated run-time environment for its execution.

One of the most popular containerization solutions is Docker [1]. An application encapsulated using Docker is distributed as a Docker image: an executable package including everything needed to run the application. A docker image is built through a Dockerfile. This file contains all commands and environment variables required to support the application's execution.

[1] A token is a unique identifier assigned by the GitLab Runner to each job that is ready to run.

A Docker container is an instance of a deployed Docker image that contains the encapsulated application. Thus, a Docker image is like a snapshot of an application and it allows to build a container in a really simple and reproducible way.

To build a new Docker image it is not necessary to write a new Dockerfile from scratch but it is possible to inherit image definitions from another base image. This can be done using the FROM command into the Dockerfile allowing to inherit all properties and files encapsulated in the base image into the new image. Hence, any Docker image can be extended and used as a base image.

To build an image, Docker executes all the statements reported into the Dockerfile and generates a layer for each instruction. Each layer contains only a collection of differences from the previous layer. Thus, a Docker image is composed of a large pileup of layers. Furthermore, each Docker image has a unique SHA-256 code as its ID.

Docker Hub is a service provided by Docker to share container images with the community. Docker Hub has many public repositories allowing to share container images. A Docker image can be published to Docker Hub simply by pushing it onto the repository.

An image hosted on Docker Hub is considered official if it is published by a certified company or organization while it is not official if it is published by community developers. Most available Docker images use an official image (e.g., from Ubuntu) as their base image. Docker images also support various hardware architectures, such as ARM and x86.

Three are types of official images hosted on Docker Hub: OS, language run-time, and application. Docker images that only encapsulate a base operating system are OS Docker images; examples include Ubuntu and Debian. Language run-time images are Docker images that provide the run-time environment for a given programming language, such as Python and Golang. Docker images that encapsulate a ready-to-use application are application images, such as Nginx and PostgreSQL.

3 Related Work

This section contextualizes our deliverable with the current literature. More precisely, it describes related work about bad practices, challenges, and barriers. Then, it makes an overview of identification strategies for such bad practices. Finally, this section discusses studies focused on Docker.

3.1 CI/CD bad practices, challenges and detection strategies

This section describes related work about bad practices, challenges and barriers together with their identification in CI/CD and infrastructure-as-code scripts.

3.1.1 Bad Practices and Challenges in CI/CD

A proper application of the CI/CD process requires a “cultural shift” [25] among both the development and operation teams. Several barriers may arise when moving towards CI mainly related to quality assurance, security, and flexibility in performing tasks such as source code debugging [10]. Barriers can also be experienced when moving towards CD. Specifically, Olsson et al. [19] found that the complexity of the deployment environment, the need to achieve timely delivery, and the lack of a complete overview of all the development projects hinder development teams adopting a full CI/CD process. Once software projects have in place a CI/CD process, it is still possible that it could be wrongly or sub-optimal applied. Specifically, landmarks books about CI [6] and CD [12] outlined wrong decisions (i.e., anti-patterns) when applying CI/CD, such as the lack of build automation and project visibility together with the inability to create deployable software. On the same

line, Zampetti et al. [36] empirically elicited bad practices by relying on semi-structured interviews and by mining Stack Overflow posts. Finally, in a follow up study, Zampetti et al. [34] focused on what are the typical actions mainly applied during the CI/CD process evolution.

The use of Infrastructure-as-Code (IaC) is mentioned as one of the best practices associated with CI. Sharma et al. [24] proposed a catalog featuring 13 implementation and 11 design configuration smells (i.e., violations about recommended best practices for configuration code). Furthermore, Rahman and Williams [22] investigated if text features and text-mining approaches can be used to identify properties able to characterize "defective" IaC scripts. Their results point out that file-system operations, infrastructure provisioning, and managing user accounts are the main properties that could be used for characterizing an IaC script as defective. In a follow up study, Rahman et al. [21] looked deeper on the security smells introduced by practitioners in IaC scripts leading to possible security breaches. Their results point out that hard-coded passwords are the main responsible for security weaknesses, and more in general, security smells tend to have a long lifetime ($\simeq$ 20 months).

The aforementioned work focuses on conventional CI/CD, without looking at challenges and barriers being specific to CPS development. As pointed out by several studies, CPS development is much more challenging than conventional software. Specifically, Törngren and Sellgren [26] investigated CPS-design related challenges focusing on the complexity of the environment in which these systems operate. The complexity of the environment, together with compliance to standards, security issues and long build execution times are among the main factors identified by Mårtensson et al. [18] that must be considered when applying CI to software-intensive embedded systems. The specificity of CPSs is also highlighted when looking at the root causes of the defects introduced in them. In this context, Garcia et al. [9] and Wang et al. [33] studied bugs in two CPS-related domains: autonomous cars and unmanned aerial vehicles pointing out the presence of CPS-specific bugs such as anomalous behavior of a vehicle in the presence of traffic lights. Also testing activities are more difficult to automatize in CPSs, indeed Afzal et al. [3] looked deeper on the testing challenges encountered by roboticists pointing out four challenges about designing testing platforms, and five challenges about running and automating tests such as proper defining oracles capturing the systems' behavior from sensors, and compare it with the expected one.

The CPSs peculiarities pointed out by previous literature indicate the need of having specific verification and validation activities, e.g., making intensive usage of simulators before testing on real hardware, that have an impact on the way the CI/CD pipeline is configured and used. Recently, Zampetti et al. [35], starting from existing knowledge in the field, and by mining pull requests belonging to open source CPS projects (i) extended the well known bad practices and restructuring actions from the literature by accounting for new CPS-specific problems and solutions, (ii) identified challenges and mitigation strategies mainly occurring due to the CPS nature of the system under development, and also (iii) linked bad practices to restructuring actions, as well as challenges to mitigation.

3.1.2 Bad Practices Detection in Development Workflows

Researchers have proposed approaches aimed at automatically identifying, and in some cases, removing problems arising in the build process. Specifically, as regards the inspection and resolution of specific kinds of build failures, Macho et al. [17] proposed BUILDMEDIC to automatically repair Maven builds that break due to dependency-related issues, while Vassallo et al. [32] proposed BART, a tool that summarizes the reasons of the build failure and suggests possible solutions found on the Internet.

As regards the detection of smells in IaC scripts, Rahman et al. [21] implemented a linter to detect seven types of security issues that might occur in IaC scripts developed in `Puppet`.

Finally, moving on the mis-uses of the CI/CD features provided by CI/CD frameworks, Gallaba et al. [8] proposed (i) an anti-pattern detection tool for TRAVIS CI specifications, i.e., HANSEL and an anti-pattern removal tool for TRAVIS CI specifications, i.e., GRETEL, which can remove $\simeq 70\%$ of the most frequently occurring anti-patterns automatically. Deviations from the application of good CI/CD principles have also been

investigated by Vassallo et al. [29] who proposed CI-ODOR a reporting tool for CI processes that detects the existence of four relevant anti-patterns, e.g., builds becoming slow or developers working on feature branches for a longer period, by analyzing regular build logs and repository information. In a different study, Vassallo et al. [31], instead, proposed CD-LINTER, a static analysis tool able to identify four CD smells right when they are introduced in the pipeline configuration.

Based on existing work in the field, we tried to adapt some of the detection strategies already known in the literature to the GitLab CI/CD framework and extend the detection to other anti-patterns that might be specific to the CPS domain.

3.2 Docker De-Bloating

Application containers are becoming very popular and Docker is one of the most popular and largely used containerization technologies[2]. For this reason, Docker has been extensively studied over multiple aspects such as security, quality, and evolution.

For instance, Cito et al. [4] characterized the Docker ecosystem by discovering prevalent quality issues and studying the evolution of Docker images. They performed this analysis on a dataset of 70,000 Dockerfiles and compared this general population with samplings containing the top 100 and top 1,000 most popular projects using Docker. Their results show that popular projects change more often than the others. Furthermore, they assessed that 34% of all Docker images, from a representative sample of 560 projects, could not be built from their Dockerfiles. During the building process of a Docker image, temporary files are often used. If such temporary files are imported and subsequently removed in different layers by developers, it leads to the presence of unneeded files, resulting in unneeded large images. This restricts the quality of an image and can affect the scalability of a provided service. Lu et al. [16] investigated the presence of such temporary file smell through an empirical case study on 3,242 real-world Dockerfiles on Docker Hub. Kula et al. [13], instead, conducted an empirical study on library migration that covers over 4,600 GitHub software projects and 2,700 library dependencies, to investigate to which extent developers update their library dependencies. They found that 81.5% of the studied systems still keep their outdated dependencies. Moreover, based on a developer survey, they found that 69% of the interviewees claimed to be unaware of their vulnerable dependencies.

Other studies focused on the security of Docker image containers. For instance, Zerouali et al. [38] conducted an empirical analysis in which they studied the relationship between outdated system packages in Debian-based image containers, their severity vulnerabilities, and their bugs. In a follow-up study, Zerouali et al. [37] evaluated how outdated and vulnerable third-party packages are in 961 official node-based images coming from three Docker Hub repositories, i.e., node, ghost, and mongo-express. They found that the presence of outdated npm packages in official node images increases the risk of security vulnerabilities, suggesting that maintainers of official Docker images should keep their installed JavaScript packages up to date, since official images are used as the basis for creating community images.

To sum up, previous studies highlighted that Docker images often pack unnecessary resources, which not only results in excessive space usage but also potential vulnerabilities. This is the reason why our focus is to identify unused dependencies in Docker images and to produce a de-bloated image having reduced size compared with the original one.

Similarly to our tool, Rastogi et al. [23] designed and implemented CIMPLIFIER. This tool can divide a container into many simple ones, enforcing privilege separation between them, by eliminating the resources that are not necessary for application execution. To achieve this goal, they developed techniques for identifying resource usage, performing partitioning, and for gluing the partitions together to retain original functionality. Results show that CIMPLIFIER can reduce the container size preventing application functionality. Differently from them, we do not use system calls to verify if a resource is used or not. Specifically, we start for packages

[2] https://insights.stackoverflow.com/survey/2020

installed inside an image and we remove them checking if the application still works without the removed packages.

4 CI/CD Configuration Analyzer

This section introduces an automated reporting tool (i.e., CI/CD CONFIGURATION ANALYZER) that can be integrated into CI/CD pipelines to help developers increase their awareness about possible bad practices negatively impacting the overall CI/CD process in place, e.g., maintainability issues in the long term, as well as performance issues increasing the time interval between the change and the feedback for developers.

Starting from the existing list of CI bad practices experienced in "traditional" software development [36], as well as common restructuring activities applied when evolving the CI/CD process [34] to improve the effectiveness of a CI/CD pipeline, we conducted an internal selection to identify a subset of bad practices to be automatically detected by only relying on versioning information. During the selection process, we have also considered the outcomes of our deliverable D3.2 [27] which details: (i) a list of challenges and barriers for CI/CD pipeline setting and evolution in the CPS domain, together with (ii) a list reporting the mitigation mainly used to overcome them. The latter has been done to focus on bad practices and/or challenges that can occur both in "traditional" software development, as well as can struggle developers during CPS development.

We ended up with the selection and the implementation of three detectors, for which we add an explanation of the related bad practices and a description of the detection strategy.

4.1 Which bad practices to detect, and how?

There may be many bad practices in CI/CD pipelines, however, it is possible to detect only a subset of them. Specifically, the rationale behind our internal selection was two-fold. We wanted to cover different aspects of the CI/CD pipeline that can be felt problematic also when using the process for developing CPSs. At the same time, we selected the bad practices that can be detected using data typically produced by every CI/CD pipeline independently from custom settings, i.e., versioning information. Note that, since the tool works only by looking at the configuration file of the CI/CD pipeline, i.e., `.gitlab-ci.yml` file, it is considered a "linter" meaning that the selected bad practices are statically identified. Specifically, this tool does not consider CI/CD bad practices requiring historical information for the detection, e.g., requiring artifacts such as build logs available only when the CI/CD pipeline is already in use and not when it is configured. In other words, we only considered for this static tool the feasibility of detecting the CI/CD bad practices from configuration files alone, without relying on other artifacts.

The following list provides the CI/CD practices selected for the static tool, together with their detection strategies.

> *Manual triggering for pipeline stages.* This bad practice deals with the organization of the overall CI/CD process in terms of the adoption of poor (i.e., non-fitting with the organization) build triggering policies, e.g., the way the CI/CD process is enacted. Specifically, it is always suggested to avoid including manual tasks, stages, and/or jobs in the CI/CD process, since the main goal of a CI/CD process is to keep the codebase in a deployable status at any given time. However, there might be special circumstances (e.g., release on a production environment) where the organization needs to manually start a build. Missing full automation will make the overall build process non-repeatable, as well as might introduce errors and delay the delivery of code changes to the customers. This bad practice accounts for stages that must be manually started by a user.
> In GitLab CI/CD, it is possible to configure a job to require manual intervention before it runs, i.e., the job is added to the pipeline, but does not run until the user clicks the play button on it. To do this, you must set the `when: manual` option when adding and configuring the job into the

`.gitlab-ci.yml` file. It is important to highlight that, in the presence of manual jobs/stages in the pipeline, the GitLab runner will skip all of them, and the pipeline completes successfully even though the manual jobs are not triggered. This is because manual jobs are considered optional by the GitLab runner. However, not all manual triggers are a problem though [31]. CI/CD advocates the automated execution of all stages to ensure a releasable project state at every point in time. However, it is acceptable to manually decide when this release should happen. Therefore, our tool is able to exclude cases where the manual execution affects deployment stages. An example of the usage of the `when: manual` feature used for manual triggering a GitLab CI/CD job is reported in Figure 3.

```
1 code_quality:
2   stage: build
3   script: 'mvn sonar:sonar'
4   when: manual
```

Figure 3: Example excerpt of GitLab CI/CD jobs manually triggered

Retry Failure. Flaky behavior leads to non-determinism in the execution of the build process, while the build process should be deterministic and repeatable. One highly investigated root cause behind non-deterministic behavior is related to the execution of "flaky" tests [7, 14, 15, 20, 40]. However, the root causes behind flaky behavior in CPSs may be different from traditional software. Specifically, the CI/CD pipeline can suffer from flakiness because simulators could not cope with timing issues, because of the complex interacting environment (e.g., the complexity of the subsystems whose features interact across many indirections), or external resources changing their behavior while the pipeline owner has no control over them (e.g., the load on the server-side or the inclusion of external tools that may behave differently). Finally, when using remote hardware devices, flakiness can be determined by the presence of noise in the measured values.
When configuring and using a CI/CD pipeline for CPS development, it is important to avoid the non-determinism, since it may hinder the development experience, slow down progress, and hide real bugs. Some pipelines address this issue by rerunning a job multiple times after failures [34]. However, this might not only hide an underlying problem but also make issues harder to debug when they only occur sometimes (i.e., its abuse is a bad practice [31]).
We propose to warn developers when the `retry` feature is used in the GitLab CI/CD configuration files. Specifically, the `retry` parameter allows developers to configure how many times a job is going to be retried in case of a failure. Our tool detects all cases where `retry` is set to a positive value. Figure 4 shows an excerpt of how developers can use the `retry` parameter when defining a pipeline job with GitLab CI/CD. Specifically, the example reports the definition of a *test* job, wherein in the presence of specific failing scenarios (i.e., runner_system_failure and stuck_or_timeout_failure) the job can be re-executed two more times before ending the build process with a failure status.

```
1 test:
2   script: rspec
3   retry:
4     max: 2
5     when:
6       - runner_system_failure
7       - stuck_or_timeout_failure
8
```

Figure 4: Example excerpt of GitLab `retry` configuration

Failures intentionally hidden. When configuring a CI/CD pipeline, developers try to include several stages, each one aimed at spotting a specific type of issue/defect affecting the system under development. As an example, stages dealing with testing are mainly aimed at identifying misbehavior of the system at different granularity levels, e.g., unit, integration or performance, while stages dealing with static code analysis tools are used to spot the presence of pieces of code that do not adhere to organizational policies or that may lead to maintainability issues in the long term. Obviously, to identify the defects as soon as they are introduced in the codebase, it is important that each stage should be able to fail the build. If this is not the case, developers can miss or ignore the underlying issue, leading to the decay of the CI/CD process in the long term. However, GitLab CI/CD, similarly to other CI/CD frameworks [34], allows the definition of build stages and/or jobs which outcome do not affect the overall build outcome.
We propose to warn developers when the `allow_failure` parameter is set to true, i.e., the job/stage is allowed to fail without impacting the rest of the build. Specifically, "when jobs are allowed to fail (allow_failure: true), an orange warning in the dashboard indicates that a job failed. However, the pipeline is successful and the associated commit is marked as passed with no warnings"[3]. Figure 5 reports an example where we have two different jobs (i.e.,, job1 and job2) running in parallel, while job3 belonging to the "deploy" stage only runs when job1 ends without failures. Indeed, job2 is configured by setting the `allow_failure: true`: its outcome will never impact the overall build execution process/outcome.

```
1  job1:
2    stage: test
3    script:
4      - execute_script_1
5
6  job2:
7    stage: test
8    script:
9      - execute_script_2
10   allow_failure: true
11
12 job3:
13   stage: deploy
14   script:
15     - deploy_to_staging
```

Figure 5: Example excerpt of GitLab `allow_failure` configuration

4.2 Infrastructure

Figure 6 shows the overall infrastructure of the tool detecting three bad practices starting from the CI/CD pipeline configuration files. As it can be seen from the figure, we have a set of three Python modules (i.e., Plugged Detection Modules, reported in green), each one detecting a specific bad practice, i.e., `manual_steps_and_or_jobs`, `retry_usage` and `jobs_allowed_to_fail`. We also have a utility module, i.e., `detection_framework` with the goal of providing basic functionality, and the orchestrator module, i.e., `config_analyzer` containing the implementation logic for properly identifying the bad practices and storing the results as a textual file. Note that, the overall infrastructure has been thought so that it will be easier to further add new plugged detection modules based on specific organizational needs.

[3] `https://docs.gitlab.com/ee/ci/yaml/#allow_failure`

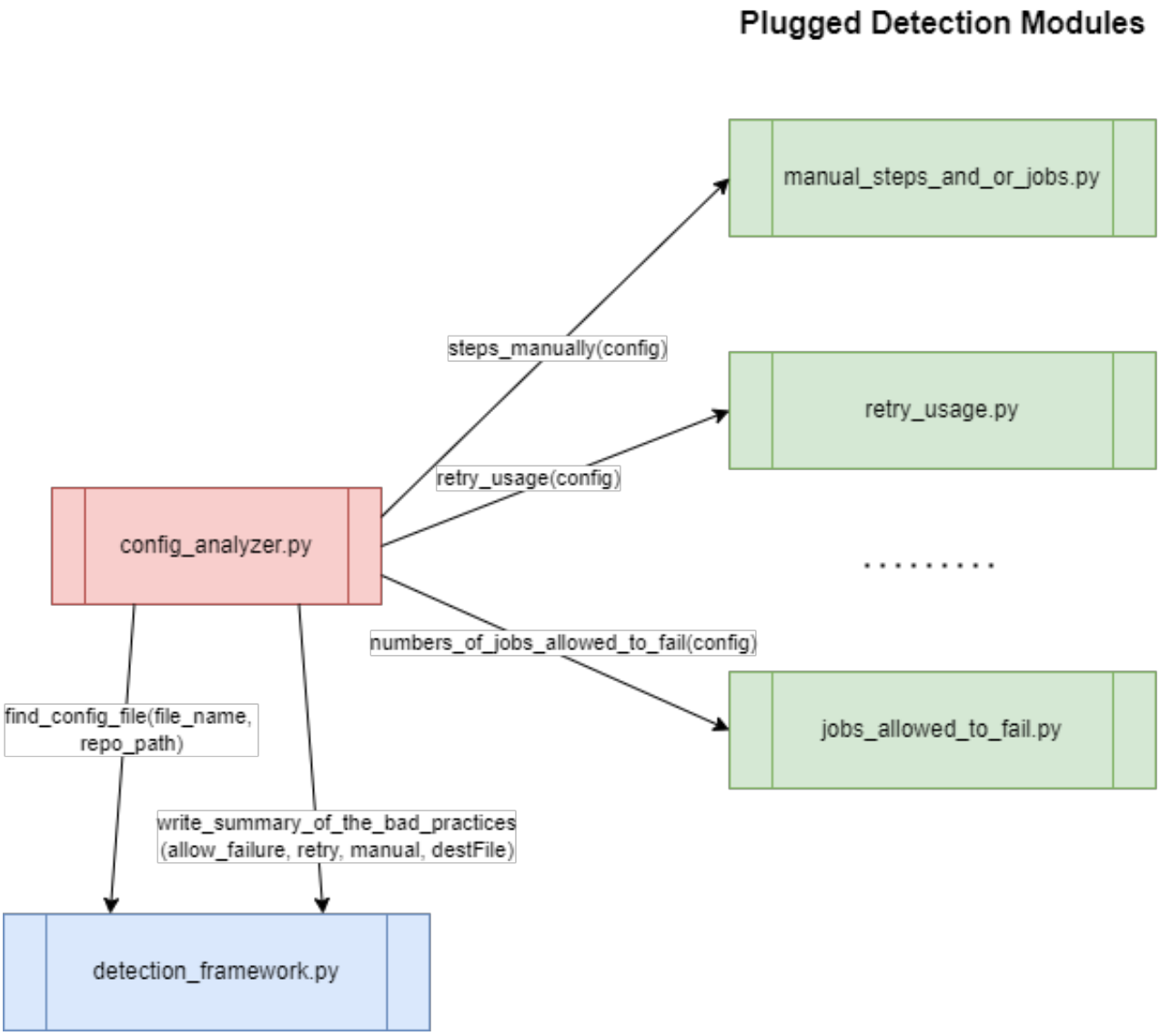

Figure 6: CI/CD CONFIGURATION ANALYZER Infrastructure

First of all, as reported in Figure 6, it is important, given the path of the repository to analyze, to identify the set of configuration files being defined that will become the object of the analysis (`find_config_file(file_name), repo_path`). In doing this, since developers can customize the name of the CI/CD configuration files, it is possible to pass as a parameter the name of the configuration file to use for searching, otherwise, as default the tool will rely on the common `.gitlab-ci.yml` file.

After having identified the set of configuration files matching the given name in the repository, for each of them, the orchestrator module will call one by one and sequentially the plugged detection modules for checking the presence of the bad practices in the CI/CD configuration file currently used within the organization. In doing this, it is important to properly load the configuration file to analyze and transform it into an easier-to-handle format (i.e., rely on the `yaml` module to transform the configuration file into an easy-to-handle Python object). The latter (i.e., `config`) is the only argument that must be passed as parameter to the functions implemented in the Plugged detection modules, each one aimed at detecting the occurrence of a specific bad practice, i.e., `steps_manually(config)`, `retry_usage(config)`, and `numbers_of_jobs_allowed_to_fail(config)`.

As regards the data generated by the previously introduced functions, it is important to note that the type of data being provided is highly dependent on the type of bad practices. Specifically:

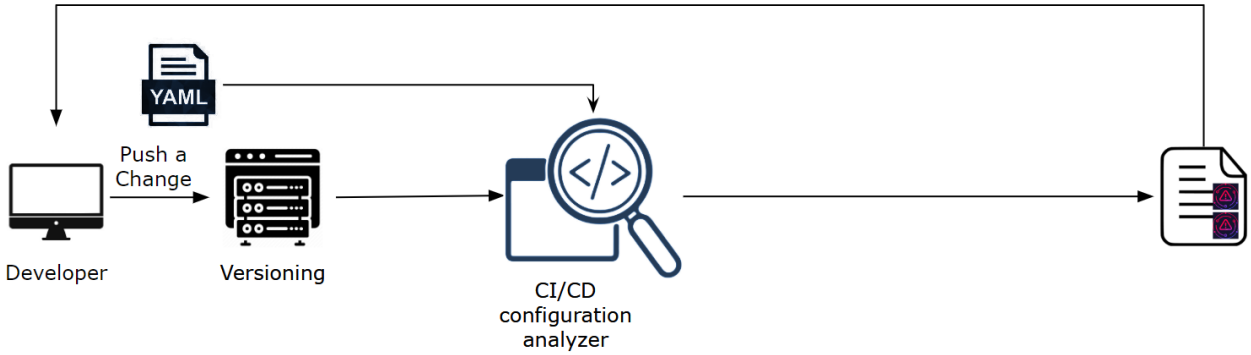

Figure 7: Possible location of the CI/CD CONFIGURATION ANALYZER in the context of a CI/CD process

- `numbers_of_jobs_allowed_to_fail(config)` returns, if any, a list containing the name of the jobs for which the `allow_failure` parameter is set to true, otherwise it returns an empty list;
- `steps_manually(config)` returns, if any, a list containing the name of the stages with at least one job for which the `when` parameter is set to manual, otherwise it returns an empty list. Note that if the tool is not able to properly assign the job to a specific stage in the CI/CD pipeline, it simply returns the name of the job that must be triggered manually;
- `retry_usage(config)`, instead, returns, if any, a list containing the name of the job(s) for which the `retry` parameter has a positive value (> 0) together with the value aimed at specifying what is the number of attempts scheduled for the job, otherwise it returns an empty list.

At this point, it is important to mention what are the possible usage scenarios considered by the CI/CD CONFIGURATION ANALYZER. As shown in Figure 7, we envision the possibility to execute our static configuration analyzer, each time a developer pushes a new change to the version control. Having said that, since at the moment the analyzer is not able to continuously monitor the changes occurring to a specific system under development, we provide the possibility to execute our static configuration analyzer in two different ways:

- specifying the repository path, leaving the tool to identify the last change being applied by a developer to a specified branch (in case no branch is specified the tool assumes as default the need to check on the release branch);
- specifying directly the identifier of the change (i.e., commit sha) the developer wants to process, together with the name of the branch to which the commit belongs. Also in this case, if the user does not specify the name of the branch, the tool assumes that the change is part of the stable release branch of the system under development (i.e., master branch). Differently from the previous case, the tool has to apply a `checkout` operation to have a picture of the system at the specified snapshot.

4.2.1 How to add a new detector

Based on the infrastructure shown in Figure 6, it is possible to state that it will be easy to extend the tool to account for different bad practices to be detected by looking at the CI/CD configuration file. Specifically, you only need to add a new plugged detection module in which you must define a function that, taking as input the pretty-formatted version of the configuration file, implements the desired detection strategy, and provides as result a list containing a summary of the detection results. Of course, you must also add in the utility module (i.e., detection_framework) all the basic functionality needed to properly deal with the bad practice that could also be used by other different detection strategies.

4.3 User Manual

CI/CD CONFIGURATION ANALYZER is a library/API written in Python that can detect bad practices in the configuration of a CI/CD process of a project relying on GitLab CI/CD framework (i.e., having the `.gitlab-ci.yml` file).

Currently, it supports the detection of the following bad practices:

- Manual triggering for pipeline stages;
- Retry Failure;
- Failures intentionally hidden.

Installation

CI/CD CONFIGURATION ANALYZER must be run as a command line application.

There are specific requirements that must be satisfied to properly run the tool. First of all, you must have a Python ≥ 3.6 installed locally. Furthermore, you have to install Python requirements by running:

```
python3 -m pip install -r requirements.txt
```

Running CI/CD CONFIGURATION ANALYZER from the command line

First of all, you must clone locally the repository in the desired location, and `cd` into the folder (or include it in the path). Then, run `python config_analyzer.py -h` to show its usage (the output of the command can be found in Listing 1).

```
python ./config_analyzer.py -h

usage: config_analyzer.py [-h] [-v] [-branch BRANCH] [-commit COMMIT]
   mode repo dest

positional arguments:
  mode            enter 1 for passing a repository and 2 for passing a
     commit
  repo            Repository location to analyze
  dest            Destination Location for summary

optional arguments:
  -h, --help      show this help message and exit
  -v, --verbose   increase verbosity (default: False)
  -branch BRANCH  branch to analyze (default: master)
  -commit COMMIT  Commit to analyze (default: None)
```

Listing 1: CI/CD CONFIGURATION ANALYZER USAGE

The tool can be executed by considering two different scenarios: (i) only specifying the GitLab repository to be analyzed, or (ii) specifying the identifier of the change (i.e., commit sha) that must be processed by the tool. If you want to let the tool identify what is the last change that occurred on the repository that must be analyzed, you must run:

```
config_analyzer.py 1 <GitLabRepoPath> <destinationPath> [-branch
<branchName>]
```

Otherwise, if you want to process a specific change being applied to the repository, you must run:

```
config_analyzer.py 2 <GitLabRepoPath> <destinationPath>
-commit<commitSHA> [-branch <branchName>]
```

In both cases, the `-branch` argument is optional, i.e., if the user does not specify the name of the branch that must be analyzed, the tool assumes that the one to be analyzed is the release branch (i.e., `master`). Moreover, while in the first scenario you must use as a mode the `1` option without the need to specify the optional argument `commit`, in the other scenario you must use as a mode the `2` option and the `commit` option becomes mandatory. If you do not specify the SHA of the commit, the tool will end with the following error:

```
config_analyzer.py: error: mode 2 requires a commit SHA to be analyzed.
```

Finally, in both cases, you must consider the other two positional and mandatory arguments that require specifying the `<GitLabRepoPath>`, i.e., the path where the versioning history of the repository to be analyzed is stored, and the `<destinationPath>`, i.e., the path of the textual file where the results of the detection are stored to be further analyzed by the user.

4.3.1 Running Example

In this section we report a simple running example showing how the CI/CD CONFIGURATION ANALYZER works. Specifically, we start detailing the open-source project hosted on GitLab we have used to check what are the outputs generated by our tool. After that, we will describe the structure of the `.gitlab-ci.yml` files adopted by the project at a specific snapshot the running example refers to, and finally we will provide the textual file being generated by our tool.

As regards the project, we used `gioxa/build-images/build-rpmbuild-ruby`[4]. The project has the aim to build a docker image with all dependencies to build and package the latest version of ruby with rpmbuild. The change we look at in the context of the running example is identified with the following SHA: `d65d5bc9`. It has been done directly on the stable release branch of the repository (i.e., master) and it is aimed at changing the CI/CD configuration file (i.e., *"Update .gitlab-ci.yml"*). Note that, the CI/CD CONFIGURATION ANALYZER does not only analyze the changes where the CI/CD process configuration changes while, given any type of change applied to a repository, it will retrieve the CI/CD configuration file and analyze it.

Figure 8 shows the content of the `.gitlab-ci.yml` file of the project that is used to all the three Plugged Detection Modules, each one implementing a specific detection strategy. As can be seen from the figure, the CI/CD process is made up of four different stages, i.e., build, test, deploy and cleanup, each one having only a single job. Indeed the configuration details four different jobs: build_base belonging to the build stage, test_fpm belonging to the test stage, push-image-tags belonging to the deploy stage and the cleanup job corresponding to the cleanup stage. All the jobs being defined are properly customized in terms of properties such as what are the commands that must be executed, i.e., specified in the `script` environment, or else what are the possible dependencies among different stages by configuring the `dependencies` environment. Moving the attention on the three detectors currently implemented in our CI/CD CONFIGURATION ANALYZER, it is important to highlight that:

[4] `https://gitlab.com/gioxa/build-images/build-rpmbuild-ruby`

```yaml
stages:
  - build
  - test
  - deploy
  - cleanup

variables:
  ODAGRUN_IMAGE_LICENSES: MIT
  ODAGRUN_IMAGE_VENDOR: Gioxa Ltd
  ODAGRUN_IMAGE_TITLE: >-
    CentOS 7 docker image with all dependencies to build and package ruby with
    rpmbuild.
  ODAGRUN_IMAGE_REFNAME: ruby-rpm-build-centos
  DISTRO_RELEASE: 7
  DOCKER_NAMESPACE: gioxa
  GIT_CACHE_STRATEGY: pull
  ODAGRUN_POD_SIZE: micro

build_base:
  image: gioxa/imagebuilder-c7
  stage: build
  retry: 1
  variables:
    ODAGRUN_POD_SIZE: medium
    GIT_CACHE_STRATEGY: push-pull
    WORK_SPACES: |
      - name: repocache C${DISTRO_RELEASE}
        key: x86_64
        scope: global
        path:
          - cache/yum/x86_64/$DISTRO_RELEASE/base
          - cache/yum/x86_64/$DISTRO_RELEASE/updates
        strategy: push-pull
        threshold:
          path:
            - cache/yum/x86_64/$DISTRO_RELEASE/base/packages/*.rpm
            - cache/yum/x86_64/$DISTRO_RELEASE/updates/packages/*.rpm
  script:
    - export OS_CONFIG=make_os.conf
    - make_os
    - >-
      registry_push --rootfs --ISR --reference=$CI_PIPELINE_ID
      --config=docker_config.yml
  tags:
    - odagrun

test_fpm:
  image: 'ImageStream:$CI_PIPELINE_ID'
  stage: test
  dependencies: []
  variables:
    GIT_STRATEGY: none
  script:
    - echo $PATH
    - uname -a
    - rpmbuild --version
  tags:
    - odagrun

push-image-tags:
  stage: deploy
  image: scratch
  environment: production
  script:
    - copy --to_var=DOCKER_CONFIG_YML --from_file=docker_config.yml
    - copy --to_var=MAKE_OS_CONF --from_file=make_os.conf
    - >-
      copy --to_var=ODAGRUN_IMAGE_DESCRIPTION --substitute
      --from_file=./full_description.md.in
    - >-
      DockerHub_set_description --allow-fail
      --image=$DOCKER_NAMESPACE/$ODAGRUN_IMAGE_REFNAME
      --set-private=no
    - >-
      registry_push --from_ISR --from_reference=$CI_PIPELINE_ID
      --image=$DOCKER_NAMESPACE/$ODAGRUN_IMAGE_REFNAME
      :$ODAGRUN_SHORT_DATE
      --skip_label
    - >-
      registry_tag_image
      --image=${DOCKER_NAMESPACE}/${ODAGRUN_IMAGE_REFNAME}
      :${ODAGRUN_SHORT_DATE}
      --tag=latest
    - >-
      MicroBadger_Update  --allow-fail
      --image=$DOCKER_NAMESPACE/$ODAGRUN_IMAGE_REFNAME
  only:
    - master
  tags:
    - odagrun

cleanup:
  image: scratch
  dependencies: []
  variables:
    GIT_STRATEGY: none
  script:
    - ImageStream_delete --name="" --allow-fail
  stage: cleanup
  allow_failure: true
  when: always
  tags:
    - odagrun
```

Figure 8: `.gitlab-ci.yml` file of `build-rpmbuild-ruby` at snaphot: `d65d5bc9`

- the **build_base** job belonging to the **build** stage has been configured so that in presence of a failure during its first run it is possible to re-run the same job another time and only if also this second execution results in a failure the overall build execution ends up with a failed status, i.e., `retry: 1`;
- the **cleanup** stage, instead, has been configured so that even when the execution ends with a failure status this failure will not have an impact on the overall outcome of the build process (i.e., `allow_failure: true`).

Let's assume that the local version of the repository to be analyzed is stored in the same directory where the user has downloaded the source code of the CI/CD CONFIGURATION ANALYZER, and that in the same directory the user wants to store the results generated by the tool in a textual file having as name the identifier of the change that the detection results refer to, i.e., `d65d5bc9.txt`. At this point, let's run our detection tool using the second scenario described in Section 4.2:

```
python ./config_analyzer.py 2 ./../build-rpmbuild-ruby/ ./../d65d5bc9.txt
-commit d65d5bc9 -branch master
```

The CI/CD CONFIGURATION ANALYZER will run all the detection strategies currently implemented

and will store the results in the destination location specified by the user. Figure 9 show the content of the textual file being generated. As it can be seen from the figure, the tool correctly identifies the presence of the **build_base** job that can be retried during the same build execution for two subsequent times before ending the overall process with a failure, as well as the **cleanup** job for which the `allow_failure` configuration parameter is set to "true".

```
The following jobs are allowed to fail:
=======================================:
cleanup
=======================================
The following jobs rely on the retry feature (possible flakiness?):
=====================================================================:
build_base,1
=======================================
```

Figure 9: Outcome of the CI/CD CONFIGURATION ANALYZER for the `.gitlab-ci.yml` file in Figure 8

5 Build Log Analyzer

This section introduces an automated reporting tool (i.e., BUILD LOG ANALYZER) that can be integrated into CI/CD pipelines to help developers increase their awareness about possible bad practices negatively impacting the overall CI/CD process in place. Differently from the tool presented in Section 4 which can be triggered as soon as a new change is pushed into a branch of a repository hosted on GitLab, the BUILD LOG ANALYZER requires as input, not only the configuration file of the CI/CD process in place, but also the outcome of the CI/CD process being triggered as a result of new changes coming into the versioning system. Specifically, while the detection strategies for the bad practices handled by the CI/CD CONFIGURATION ANALYZER can be statically detected by simply analyzing the way the development teams have configured the overall CI/CD process, for the BUILD LOG ANALYZER we cannot only rely on static information but also account for dynamic information (i.e., history of the build logs in a specified time interval).

Also in this case, to identify the set of bad practices to be detected with the use of the BUILD LOG ANALYZER, we relied on existing knowledge about CI bad practices experienced in "traditional" software [36], together with restructuring activities applied to evolve the CI/CD process in place within an organization [34]. The internal selection process has also been guided by the challenges and barriers mainly encountered by practitioners during the configuration and evolution of the CI/CD process customized for the CPS domain [27]. In this way, it is possible to guide the selection process by taking into account a set of bad practices and/or challenges that can be experienced not only in "traditional" software development but also in CPS development.

We ended up with the selection and implementation of six detectors, for which we add an explanation of the related bad practices and a description of the detection strategy.

5.1 Which bad practices to detect, and how?

Due to the high number of different bad practices and challenges that might be experienced by practitioners when configuring and evolving their CI/CD process, most of them are also highly coupled with the organizational policies, as well as the application domain, we applied an internal selection process with the aim of:

1. covering different aspects of the CI/CD pipeline that can struggle developers when adopting the process for CPS development;

2. being context- or application-insensitive, meaning that the bad practices must be detected by looking at data typically produced by every CI/CD pipeline independently from custom settings, e.g., we do not consider bad practices requiring specific configuration files other than the `.gitlab-ci.yml` file, such as the inappropriate configuration of static code analysis tools considering the high number and diversity of static analysis tools available;

3. using historical information for the detection, i.e., build logs generated by the CI/CD process once it is already in place in the organization, and not only when the CI/CD process is going to be configured in the initial phase.

The following list provides the CI/CD practices selected, together with their detection strategies.

A stable release branch is missing. The CI/CD process as a software development process is highly coupled with the management of features development branches. Lacking proper management of features development branches might hinder the overall CI/CD pipeline efficiency and effectiveness. In this context, it is important to have a stable release branch to guarantee the presence of a releasable version (where developers have executed different kinds of tests, such as regression, acceptance and performance, increasing the overall confidence level on the way the system actually works) of the system under development. In other words, the presence of a **broken release branch** that is not fixed timely might prevent the effectiveness of the overall CI/CD process.
We propose to warn developers by providing a summary, related to a specified time interval, in terms of (i) what is the percentage of build failures that occurred on the release branch, as well as (ii) the number of consecutive build failures experienced on the release branch. Since GitLab CI/CD can provide different types of build status, it is important to note that our detection strategy considers only the builds that ended with one of the following three statuses: success, errored, and failed, considering errored and failed builds as failed. Specifically, a build is errored when the `install` phase aimed at retrieving and installing the needed dependencies returns a non-zero exit code, while it is failed when any subsequent phase configured in the CI/CD configuration file returns a non-zero exit code.
Going deeper into the detection strategy being applied, since the number of development activities on a software repository may vary as the system evolves, we give the user the possibility to properly specify the time interval to be used for computing our summary, as well as the name of the branch used as "release" branch. If the latter is not specified, we assume that the release branch is the `master`. Knowing this information, we extract from the build history of the project the ones fitting the time interval and occurring on the release branch. The above data is then used to properly count the number of builds belonging to each of the three build statuses being considered for the analysis (i.e., success, errored and failed) and to provide the trend in terms of how many failures are experienced on the release branch, together with the number of consecutive failures occurring on it. It is important to highlight that, more than detecting a bad practice, this strategy is going to provide information that can alert the user. In other words, this is more the detection of a symptom that might result in the experience of the bad practice later on in the project.

Pipeline steps/stages are skipped arbitrarily. Very often, possibly due to the need of having a green build status, developers skip some "problematic" pipeline steps/stages, hiding the actual behavior of the CI/CD process and, more importantly, making untrustworthy the released version. Following this approach, developers will only delay the discovery of potential problems in the development code, going completely against the CI/CD principles. As an example, in the presence of build failures due to warnings/errors raised by static analysis tools, developers could simply skip their execution to make the build "green", instead of actually focusing on the problems' resolution [34, 36]. Furthermore, developers

could also simply skip the execution of some tests to avoid failures (i.e., to address a symptom rather than fixing the cause) in the build process, as well as to speed up the build in the "commit" stage.
Our detection strategy accounts for two different scenarios. Specifically, we propose to warn about cases in which a previously failed test does no longer occur in the next (fixed) build. At the same time, we propose to point out the presence of temporarily disabled job(s). Specifically, GitLab CI/CD allows to temporarily disable a job without deleting it from the configuration file (i.e., `.gitlab-ci.yml`) by simply starting a job name with a dot. Figure 10 shows an excerpt of the `.gitlab-ci.yml` file where the job handling the test execution is hidden.

```
1 .hidden_job:
2   script:
3     - run test
```

Figure 10: Example excerpt of GitLab configuration for `hiding` a job

More in detail, given the last build outcome obtained as a result of the execution of the CI/CD process, for identifying whether or not some tests are skipped, we will look at the number of tests being skipped as generated in the summary provided by GitLab CI/CD, and if the build status is "success" and the number of skipped tests is > 0, then we will compare with the previous build generated by the same event and in the same branch. The comparison has the main goal of checking whether or not the skipped test cases can be a symptom of a practice aimed at hiding the failure that occurred in the build process. So we will notify developers only when in the previous build there were no skipped test cases and the build status was one among failed or errored.
Moving on to the strategy used for detecting the presence of jobs being skipped, if by comparing the last build outcome with the previously generated one belonging to the same branch and triggered by the same event, we identify a change from failed or errored towards success, we will analyze the current version of the `.gitlab-ci.yml` file and warn developers about the possibility that the success of the build might be derived from the presence of having some jobs hidden in the configuration. In this specific case, it is important to note that, other than analyzing the build logs, the detection strategy also requires access to the static information stored in the CI/CD configuration file.

Build time too long. A slow build, caused by a coding issue or by a high workload of the build server, produces waiting times for developers and adds overhead to the CI process [30]. For CPSs, the problem can be further exacerbated upon deploying and executing software on simulators or hardware-in-the-loop (HiL). Based on the results of our interviewees [27] we found that practitioners highly struggle with this problem, and they try to use different mitigation strategies. Among them, we mention the prioritization and selection of only a subset of test cases in the test suite to be executed, the introduction of parallelization within the overall build process, and the use of nightly (or periodic) builds for time-intensive tasks. A different practice for dealing with slow builds is to, whenever possible, find an out-of-box Docker image instead of building a complex environment, making the build much faster [35].
Considering the high variability of the application domains, specifically for CPS development, it is not possible to identify a specific fixed value that must be considered as "long". For this reason, the detection strategy implemented in BUILD LOG ANALYZER gives the possibility to the user to specify both the time interval (i.e., number of previous months) to be considered in the analysis, and the percentage of shift with respect to the median average time computed in the time interval that is still considered acceptable by the organization. Based on the previously defined inputs, our strategy, each time a new build is triggered by the CI/CD process, identifies all the builds occurring in the specified time interval belonging to the same branch of the change enacting the process and ending with the same build status, and generates a warning each time the build execution time is higher than the median build duration plus the percentage tolerance. We know that there are cases where the build execution time might depend on external factors, such as the priority given to the project. However, based on what was reported

in previous literature [30], we do not consider this a threat, since even in these cases, developers might be interested in changing the CI/CD framework being used.

```
test-job:
  stage: build
  cache:
    - key:
        files:
          - Gemfile.lock
      paths:
        - vendor/ruby
    - key:
        files:
          - yarn.lock
      paths:
        - .yarn-cache/
  script:
    - bundle install --path=vendor
    - yarn install --cache-folder .yarn-cache
    - echo Run tests...
```

Figure 11: Example excerpt of GitLab configuration for `multiple caches`

Inappropriate cache handling. Caching strategies are very often used to store the content that does not change among subsequent builds or jobs in the same pipeline, and reuse them to speed up the overall build process. In other words, as regards GitLab CI/CD, caches allow avoiding the download of content, like dependencies or libraries, each time a job must be run. For instance, `Node.js, PHP, Ruby gems or Python` packages can be cached. However, once introduced, the caching specification may evolve as a consequence of the evolutionary history of the project [34]. The current version of GitLab CI/CD allows having a maximum of four different caches being specified for each job included in the context of a specific CI/CD process. Figure 11 shows an excerpt of the configuration where multiple caches are used. More in detail, to specify a cache for a specific job, you must use the `cache` key and, by properly setting its parameters (e.g., `files` and `path`), it is possible to customize its configuration to adapt it to specific process needs. The above parameters can be used to specify the type of files that are cached, as well as the location where the cached files are actually stored. It is important to remark that GitLab CI/CD allows defining `cache` globally so that all jobs inherit it. As an example, Figure 12 shows how to globally cache the installation of Node.js dependencies. Specifically, since by default, `npm` stores cache data in the home folder ($\sim$ $/.npm$), it is mandatory to specify using $./.npm$, and caching it per-branch.

```
image: 'node:latest'
cache:
  key: $CI_COMMIT_REF_SLUG
  paths:
    - .npm/
before_script:
  - npm ci --cache .npm --prefer-offline
test_async:
  script:
    - node ./specs/start.js ./specs/async.spec.js
```

Figure 12: Example excerpt of GitLab configuration for specifying `cache` to be used globally

Our strategy, each time a new build is triggered by the CI/CD process, selects all the jobs having an execution time higher than the median job's duration (in the pipeline) plus the percentage tolerance (specified by the user). After that, by analyzing the `.gitlab-ci.yml` file, it checks, for each job previously identified, whether or not they rely on the `cache` feature, and warns developers with the name of the jobs having a "long" duration for which probably the enabling of the caching feature could produce a saving of the overall execution time.

Different jobs consistently give a different outcome. In GitLab CI/CD, as well as in other CI/CD frameworks, it is possible to rely on jobs with the goal of having the same (or slightly different) build process running on multiple different environments (e.g., different operating systems, different simulators and also different versions for the hardware devices). As an example, Figure 13 shows an excerpt of a `.gitlab-ci.yml` file where the CI/CD process is made up of four stages. Specifically, for the build stage, there are two different jobs aimed at building the system by relying on two different operating systems (i.e., archlinux and centos7). In the presence of multiple environments, when a build ends with a failing status, it is important to discriminate whether or not the root cause behind the failure is related to the actual code change pushed to the repository, or else there is a specific "problematic" environment that consistently marks the overall build outcome as failed. In the above scenario, it might be possible to suggest developers introduce the "allow_failure" feature, since it is possible that the consistently failing environment could be considered unstable, and, only once the environment reaches enough maturity, it is important to make the job affecting the overall build status. In this way, it is possible to prevent developers from completely losing faith in the output provided by the CI/CD process, as well as to check the quality of their proposed changes to identify possible issues and/or drawbacks as soon as they are introduced into the system.
Based on the aforementioned problem, our BUILD LOG ANALYZER warns developers each time, in the specified time interval, there is at least one job having a consistent different outcome (i.e., failing) with respect to the other jobs belonging to the same pipeline. To do this, each time a new build log is generated from the CI/CD process, if the build status is marked as failed, our tool checks whether among the executed jobs all are failing or there are some with a different outcome. In the latter case, our tool extracts all the builds that occurred in the specified time window (on the same branch of the current triggered CI/CD process), and for each of them extracts the name of the executed jobs with their outcome. After that, we will compare whether across the considered history, in terms of outcome, the jobs show a consistent behavior. If this is the case, all the failing jobs having consistent behavior are reported back to the developers.

Build stages are not properly ordered, e.g., a stage responsible for the build failures is always executed towards the end of the CI/CD process, leading to wasting resources and time needed to provide feedback. GitLab CI/CD gives the possibility to include different stages in the pipeline that must be executed sequentially. So it is important to properly define the appropriate order of execution of the different stages. Whether there are cases where the order is dictated by the type of steps involved in the stages (e.g., the build stage must be done before the test stage), there might also be cases where developers can customize their order of execution. As an example, consider a stage aimed at simply checking the code quality, e.g., by running different kinds of static code analysis tools, scheduled after the stage aimed at running both unit and integration tests. In this specific scenario, it is possible that the code does not adhere to specific organizational guidelines and code standards, resulting in subsequent build failures, while the testing stage always ends with a successful outcome. If this is the case, it is possible to warn developers about the possibility to anticipate the code quality check stage before the test stage, so that developers are notified as soon as a problem is encountered in the CI/CD process.
Our detection strategy is enacted each time a new build process is triggered and ends with a failing status. At this point, by parsing the log traces of the jobs executed in the pipeline having a failing outcome, the BUILD LOG ANALYZER tries to identify the orders of the stages and what is the stage responsible for the failure. After that, our tool compares this information with similar information extracted by selecting all

```yaml
stages:
  - build
  - test
  - deploy
  - experiment

archlinux:
  image: 'archlinux/base:latest'
  stage: build
  tags:
    - newton
  before_script:
    - pwd
  script:
    - echo "do something"
  allow_failure: true

centos7:
  image: 'centos:centos7'
  stage: build
  tags:
    - newton
  before_script:
    - pwd
  script:
    - echo "do something"
  allow_failure: true
```

Figure 13: Example excerpt of GitLab configuration for specifying different jobs for different environments

the pipelines belonging to the same branch in the specified time window. If the stage responsible for the failure is consistent across the history, and it is usually executed as one of the last two stages (assuming that there are more than two stages specified for the process) defined in the `.gitlab-ci.yml` file, our tool warns the developers with a message suggesting the possibility to anticipate, if possible, the execution of the "failing stage".

5.2 Infrastructure

Figure 14 shows the overall infrastructure of the BUILD LOG ANALYZER detecting possible misuses of the CI/CD process, as described in Section 5.1. As it can be seen from the figure, in the current version of the tool, we have six different Python Plugged Detection Modules, each one aimed at detecting one of the possible CI/CD misuses, namely `broken_release_branch`, `skip`, `trend_build_log`, `cache_management`, `inconsistent_environments` and `stages_ordering`. Of course, similarly to the previous case, the infrastructure has been designed so that it will be easier to add new detection strategies dealing with different CI/CD misuses. For this reason, the infrastructure provides a utility module, i.e., `detection_framework` with the goal of providing basic functionality, such as identifying the build history related to a specific time interval based on the type of branch and the type of event which is used to trigger the CI/CD process or to extract the jobs information belonging to a specific pipeline (identified by means of the pipeline_id value assigned by GitLab CI/CD). Finally, we have the orchestrator module, `build_log_analyzer` containing the implementation logic aimed at properly determining, based on the

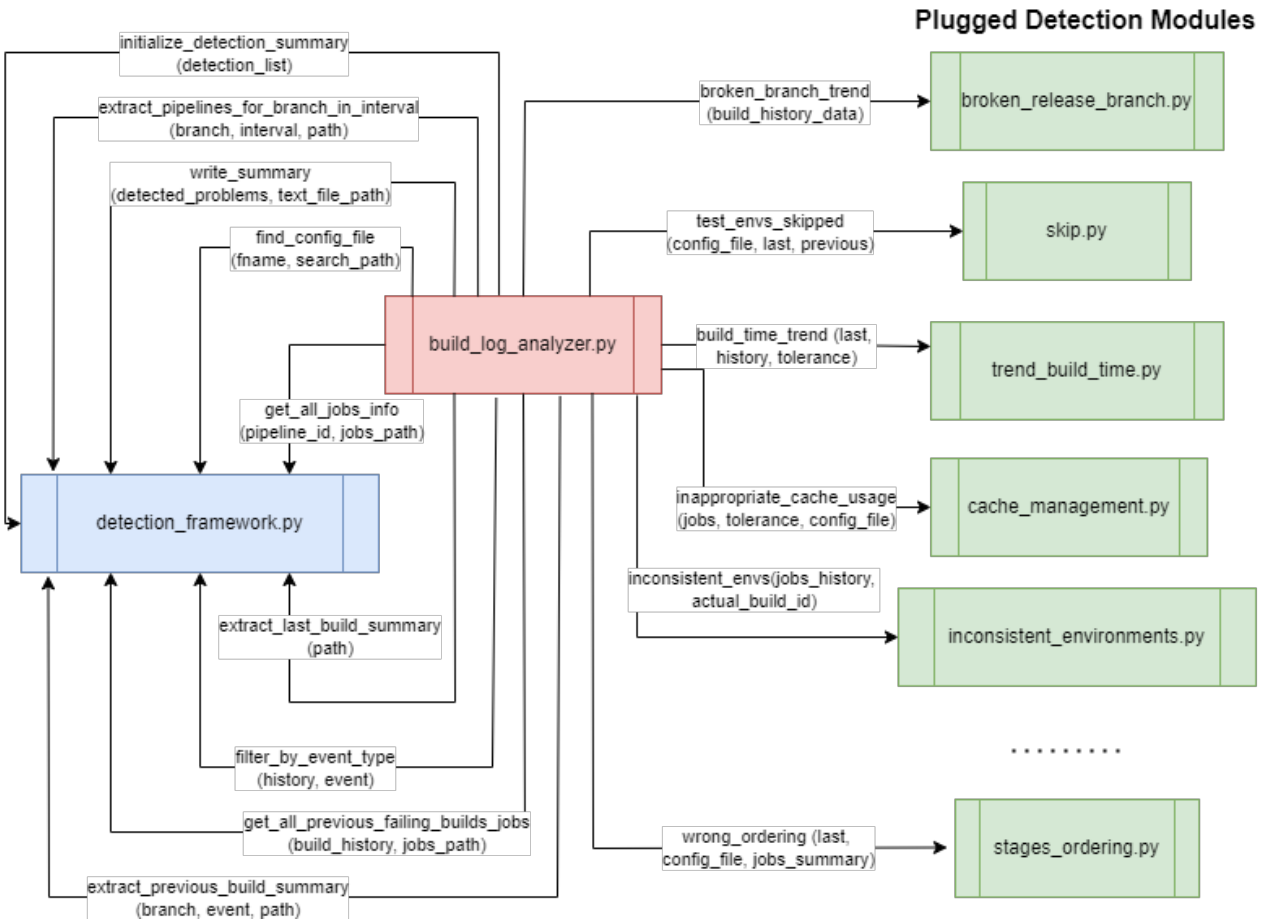

Figure 14: BUILD LOG ANALYZER Infrastructure

build outcome of the last build triggered by GitLab CI/CD, what are the possible CI/CD misuses that need to be inspected, and to store the summary being provided by our tool as a textual file.

Note that, differently from the CI/CD CONFIGURATION ANALYZER, in this case, the tool is enacted each time a new build is triggered by GitLab CI/CD and, as soon as, the build logs information is available. Moreover, differently from the static approach described in Section 4, in this case, the detector works by looking at the builds summary information, together with the information belonging to the single jobs executed in the pipeline and the log traces, and in some cases, also requires the analysis of the `.gitlab-ci.yml` configuration file.

Going deeper on the way BUILD LOG ANALYZER actually works, first of all, it is important to determine the status of the last pipeline triggered for a specific repository by GitLab CI/CD. The latter is done since the CI/CD misuses we are detecting are highly dependent on the status of the last pipeline being executed. Specifically, while the checks for the presence of an increasing build execution time and the inappropriate configuration of the caching are independent of the status of the last build outcome, the presence of steps and/or stages being arbitrarily skipped is only checked when the status of the last build is "success", while the presence of a broken release branch, environments consistently giving a different (failing) outcome, and the inappropriate ordering of build stages are only checked when the status of the last build is either "failed" or "errored". This step will provide a set of possible CI/CD misuses that must be verified.

In the following, we will provide a detailed description about what are the inputs to be given to each detector focused on a specific CI/CD misuse, as well as what are the outputs being produced and stored.

- `broken_branch_trend(build_history_data)` takes as input the summary of the build history occurring in a specific time interval (provided as input by the user and expressed in terms of months) on the release branch (if it is different from the "master" the user can specify it). Based on the above

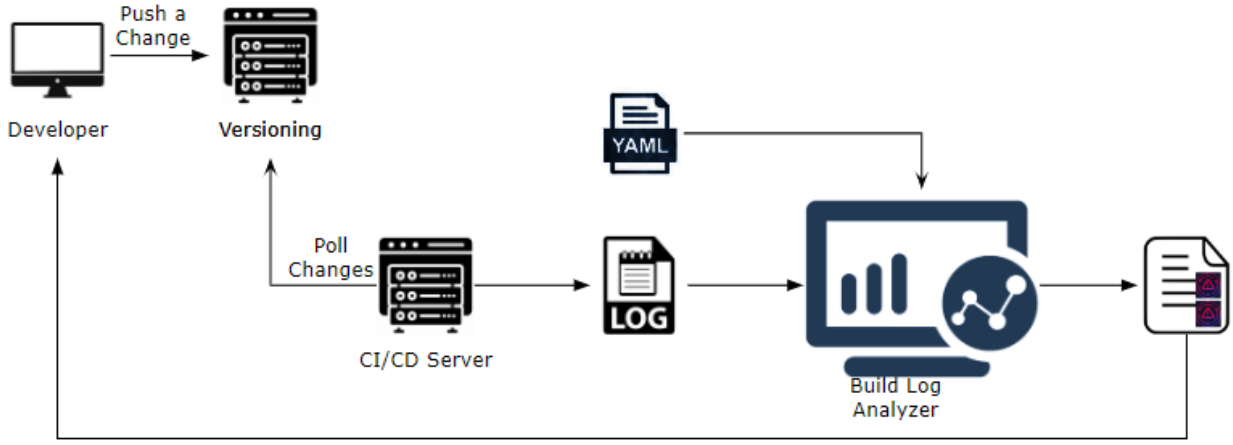

Figure 15: Possible location of the BUILD LOG ANALYZER in the context of a CI/CD process

information, it generates the maximum number of consecutive failures occurring on the release branch together with the percentage of failures experienced on it in the time window being analyzed;

- `tests_envs_skipped(config_file, last, previous)` takes as input the `.gitlab-ci.yml` file that has triggered the last build, together with the summary of the last executed build and the one executed immediately before. In retrieving the latter, BUILD LOG ANALYZER will identify among the whole set of pipelines occurring on a specific repository, the one immediately before the last one that occurs on the same branch and is generated by the same event. Based on the inputs, it will provide the number of tests being arbitrarily skipped (if any), and the name of the jobs that are hidden from the configuration file (if any);
- `build_time_trend(last, history, tolerance)` takes as inputs the summary of the last build triggered by GitLab CI/CD, the build history of the repository belonging to the specified time window, and the percentage of shift with respect to the median average execution time that is still considered acceptable by the user. As an outcome, it will provide a statement representing whether the build execution time is in line with what is considered acceptable by the organization, or else it is increasing, i.e., representing a possible CI/CD misuse;
- `inappropriate_cache_usage(jobs, tolerance, config_file)` takes as input the configuration file used for triggering the last build on the GitLab CI/CD server, a summary of the jobs executed during the last build and, as before, the percentage of shift with respect to the median average job execution time that is still considered acceptable by the user. Based on the aforementioned data, it will provide the name of the jobs (if any) whose execution time is higher than the median execution time in the same pipeline for which the cache feature is not enabled. In this way, the user can still check whether it is a good idea to avoid wasting resources and time by caching re-usable artifacts;
- `inconsistent_envs(jobs_history, actual_build_id)` takes as input the name of the jobs and their outcome in a specified time window occurring on the same branch of the last build, together with the identified of the last build, and generates as output the list of jobs (if any) which outcome is consistently failing with respect to the other jobs included in the pipeline;
- `wrong_ordering(last, config_file, jobs_summary)` takes as input the summary information of the last build, the history of jobs occurring on the same branch as the last build and generated by the same event, and the `.gitlab-ci.yml` configuration file of the last build. As an outcome, it highlights the name of the stages (if any) that are consistently responsible for the failure that are executed as one of the last two stages being defined in the configuration file.

Figure 15 shows how it is possible to integrate BUILD LOG ANALYZER in the CI/CD process. As it can be noticed, it is activated each time a new build is executed by the GitLab CI/CD framework. However, the current

version of our tool takes as input a list of build logs summary together with the summary of each job executed by them with the related log traces and applies the analysis considering as "last" the most recent build among the downloaded ones. Our idea is, given a repository hosted on GitLab to be continuously monitored, to have a crawler that periodically (e.g., every five minutes) will inspect for the presence of new data being available on the GitLab CI/CD server, download and store them, and generate the event for triggering our BUILD LOG ANALYZER. The generated events represent the presence of new data available for processing.

5.2.1 How to add a new detector

Based on the infrastructure shown in Figure 14, it is possible to state that it will be easy to extend the BUILD LOG ANALYZER tool to account for different CI/CD misuses to be spotted by looking at the build logs and CI/CD configuration files. It is only needed to add a new plugged detection module having a function that, taking a set of parameters as input, implements the desired detection strategy, and provides as a result a list containing a summary of the detection results representing possible CI/CD misuses. Of course, the utility module (i.e., detection_framework) can be extended too, for incorporating new basic functionality that could be also used in the future.

5.3 User Manual

BUILD LOG ANALYZER is a library/API written in Python that can point out the presence of possible CI/CD misuses by looking at the build logs history (build log traces) of a project relying on GitLab CI/CD framework (i.e., having the `.gitlab-ci.yml` file).

Currently, it supports the detection of the following CI/CD misuses:

- A stable release branch is missing;
- Pipeline steps/stages are arbitrarily skipped;
- Build time too long;
- Inappropriate cache handling;
- Different jobs consistently give a different outcome;
- Build stages are not properly ordered.

BUILD LOG ANALYZER must be run as a command line application.

There are specific requirements that must be satisfied to properly run the tool. First of all, you must have a Python ≥ 3.6 installed locally. Furthermore, you have to install Python requirements by running:

```
python3 -m pip install -r requirements.txt
```

Running BUILD LOG ANALYZER from the command line

First of all, you must clone locally the repository in the desired location, and `cd` into the folder (or include it in the path). Then, run `python build_log_analyzer.py -h` to show its usage (the output of the command can be found in Listing 2).

```
python ./build_log_analyzer.py -h
```

```
usage: build_log_analyzer.py [-h] [-v] [-time TIME] [-branch BRANCH]
   pipelineDataSummary jobsDataSummary logsFolder buildTimeShift
   jobsTimeShift repo destinationSummary

positional arguments:
  pipelineDataSummary      Path to the summary for builds
  jobsDataSummary          Path to the summary for jobs
  logsFolder               Path to the log traces
  buildTimeShift           Tolerance percentage for builds
  jobsTimeShift            Tolerance percentage for jobs
  repo                     Repository location to analyze
  destinationSummary       Destination Location for summary

optional arguments:
  -h, --help       show this help message and exit
  -v, --verbose    increase verbosity (default: False)
  -branch BRANCH   branch to analyze (default: master)
  -time TIME       time interval analyze (default: 1)
```

Listing 2: CI/CD CONFIGURATION ANALYZER USAGE

Based on what is reported in Listing 2, it is possible to note that there are many arguments that must be provided to make the tool properly work. Currently, BUILD LOG ANALYZER is not able to automatically monitor when a new build log is available for processing, while it is able to query by using the GitLab API to search for whether or not new builds have been triggered on a repository. For this reason, all the downloaded information, i.e., pipelines (i.e., pipelineDataSummary) with related jobs (i.e., jobsDataSummary) summary information, are properly stored in specific textual files, while logs are stored in a specific folder (i.e., logsFolder). The above information has to be given as argument to the BUILD LOG ANALYZER to spot out possible CI/CD misuses among the ones that it is able to detect.

There are other four positional arguments that are mandatory for the BUILD LOG ANALYZER tool. First of all, it is important to provide what is the percentage of shift with respect to the median average build execution time that is still considered acceptable by the organization, i.e., buildTimeShift. After that, the user must also specify the percentage of tolerance considered acceptable by the organization with respect to which to determine whether or not a job requires a long time to complete compared to the other jobs belonging to the same pipeline, i.e., jobsTimeShift. Furthermore, as for the CI/CD CONFIGURATION ANALYZER, some CI/CD misuses can only be detected by also looking at the `.gitlab-ci.yml` file. For this reason, it is important to specify the path where it is possible to examine the versioning history of the repository under analysis, together with the location where the detected CI/CD misuses must be stored.

Finally, as shown in Listing 2, there are also two optional arguments that could be provided. On the one hand, it is important to specify what is the name of the branch that is used as the "release" branch by the organization. Our tool uses as default the "master" branch, but it can be customized based on the organizational needs. On the other hand, since most of the CI/CD misuses we are detecting do not look at the whole history, while focusing on a specific time window, BUILD LOG ANALYZER gives the possibility to properly specify what is the size of the time window to look at for the analysis. The value must be passed in terms of the number of previous months to look at, so the TIME argument only takes as value positive integer values. If the user omits to specify this value, by default, the analysis is conducted considering the build history of the last month.

5.3.1 Running Example

In this section we report a simple running example showing how the BUILD LOG ANALYZER works. Specifically, we start detailing the open-source project hosted on GitLab we have used to check what are the outputs

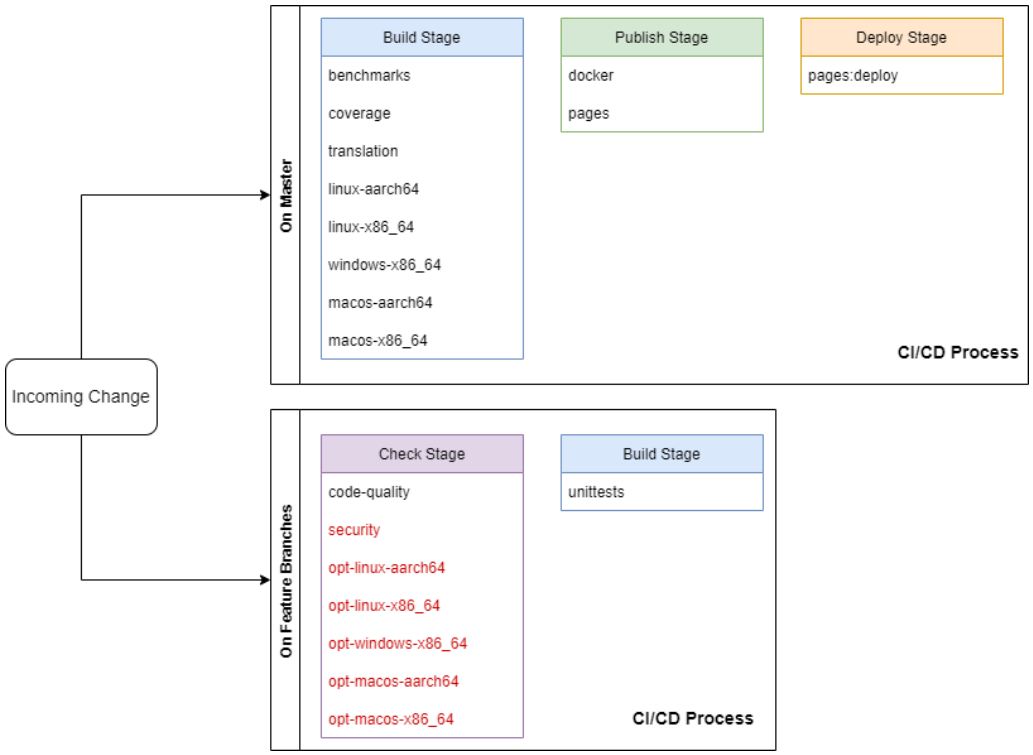

Figure 16: High-level description of the CI/CD process adopted by `veloren/veloren`

generated by our tool. After that, we will describe the structure of the `.gitlab-ci.yml` file adopted by the project at a specific snapshot the running example refers to, together with a summary of the last pipelines with the related jobs occurring in the time window of interest (the one used as default by our tool), and finally we will provide the textual file being generated by our tool. Since the detection strategies currently available in the BUILD LOG ANALYZER are enabled or not based on the outcome of the last build being processed by the GitLab CI/CD framework, we show the output considering two different builds having two different build outcomes, i.e., failed and success, respectively, as the build that will enact the activation of the tool.

As regards the project, we used `veloren/veloren`[5]: a multiplayer voxel RPG written in Rust, taking inspiration from games such as Cube World, Minecraft and Dwarf Fortress. As specified by the project's owners, the game is currently under heavy development, however, it is playable. The CI/CD process in place for the `veloren/veloren` project is highly complex, and it is summarized in Figure 16. As shown in the figure, there are two different CI/CD processes that will be executed based on the type of change applied to the repository. Specifically, if the change occurs in any of the feature branches inside the repository (i.e., bottom part of Figure 16), different from the release (master) branch, the process involves two different stages, i.e., **check** and **build**. As regards the former, it is made up of seven different jobs, of which all except **code-quality** are allowed to fail (highlighted in red in the figure). The build stage instead is only aimed at running regression tests to check for the absence of possible mis-behavior introduced by the last change. As regards the changes occurring on the stable release branch, first of all, there are no jobs that are allowed to fail. Furthermore, there are three stages being sequentially executed there, i.e., **build**, **publish** and **deploy**. It is also important to note that, in this specific case, for the **build** stage we have different jobs, each one aimed at building the application into a different environment, i.e., different operating systems so that it is possible to have the guarantee that the game is playable on multiple platforms.

Let's assume that (i) the local version of the repository to be analyzed is stored in the same directory where the user has downloaded the source code of the BUILD LOG ANALYZER, (ii) in the same directory, the user wants to store the results generated by the tool in a textual file having as name the identifier of the build that

[5] `https://gitlab.com/veloren/veloren`

```
....

benchmarks:
  extends: .release
  stage: build
  image: 'registry.gitlab.com/veloren/veloren-docker-ci
      /cache/bench:${CACHE_IMAGE_TAG}'
  tags:
    - veloren/veloren
    - check
    - benchmark
  script:
    - unset DISABLE_GIT_LFS_CHECK
    - ln -s /dockercache/target target
    - cat ./.gitlab/scripts/benchmark.sh
    - source ./.gitlab/scripts/benchmark.sh
    - >-
      TAGUUID="Z$(cat /dev/urandom | tr -dc 'a-zA-Z0-9' |
          fold -w 16 | head -n
      1)" || echo "ignore this returncode, dont ask me why,
          it works"
    - echo $TAGUUID
    - >-
      echo 'SET veloren.timestamp = "'"$(git show --no
          -patch --no-notes
      --pretty='%cd' HEAD)"'";' > upload.sql
    - >-
      echo "SET veloren.branch =
      \$${TAGUUID}\$${CI_COMMIT_REF_NAME}\$${TAGUUID}\$;"
          >> upload.sql
    - >-
      echo "SET veloren.sha =
          \$${TAGUUID}\$${CI_COMMIT_SHA}\$${TAGUUID}\$;" >>
      upload.sql
    - >-
      find target/criterion -wholename "*new/*.csv" -exec
          echo '\copy benchmarks
      ("group", "function", "value", throughput_num,
          throughput_type,
      sample_measured_value, unit, iteration_count) from
          '"'{}' csv header" >>
      upload.sql \;
    - cat upload.sql
    - >-
      PGPASSWORD="${CIDBPASSWORD}" PGSSLROOTCERT="./.gitlab
          /ci-db.crt" psql
      "sslmode=verify-ca host=grafana.veloren.net port
          =15432 dbname=benchmarks"
      -U hgseehzjtsrghtjdcqw -f upload.sql;
  retry:
    max: 2

.release:
  stage: build
  rules:
    - if: >-
        $CI_PIPELINE_SOURCE != "merge_request_event" && (
            $CI_PIPELINE_SOURCE ==
      "schedule" || $CI_COMMIT_BRANCH ==
          $CI_DEFAULT_BRANCH || (
      $CI_COMMIT_TAG != null && $CI_COMMIT_TAG =~
          $TAG_REGEX ) )
      when: on_success
    - when: never
  retry:
    max: 1


.tlinux-aarch64:
  image: >-
    registry.gitlab.com/veloren/veloren-docker-ci/cache
        /release-linux-aarch64:${CACHE_IMAGE_TAG}
  script:
    - ln -s /dockercache/target target
    - cat ./.gitlab/scripts/linux-aarch64.sh
    - source ./.gitlab/scripts/linux-aarch64.sh
    - >-
      cp -r target/aarch64-unknown-linux-gnu/release
          /veloren-server-cli
      $CI_PROJECT_DIR
    - >-
      cp -r target/aarch64-unknown-linux-gnu/release
          /veloren-voxygen
      $CI_PROJECT_DIR
  artifacts:
    paths:
      - veloren-server-cli
      - veloren-voxygen
      - assets/
      - LICENSE
    expire_in: 1 week

.twindows-x86_64:
  image: >-
    registry.gitlab.com/veloren/veloren-docker-ci/cache
        /release-windows-x86_64:${CACHE_IMAGE_TAG}
  script:
    - ln -s /dockercache/target target
    - cat ./.gitlab/scripts/windows-x86_64.sh
    - source ./.gitlab/scripts/windows-x86_64.sh
    - >-
      cp -r target/x86_64-pc-windows-gnu/release/veloren
          -server-cli.exe
      $CI_PROJECT_DIR
    - >-
      cp -r target/x86_64-pc-windows-gnu/release/veloren
          -voxygen.exe
      $CI_PROJECT_DIR
    - >-
      cp /usr/lib/gcc/x86_64-w64-mingw32/7.3-posix
          /libgcc_s_seh-1.dll
      $CI_PROJECT_DIR
    - >-
      cp /usr/lib/gcc/x86_64-w64-mingw32/7.3-posix/libstdc
          ++-6.dll
      $CI_PROJECT_DIR
    - cp /usr/x86_64-w64-mingw32/lib/libwinpthread-1.dll
        $CI_PROJECT_DIR
  artifacts:
    paths:
      - veloren-server-cli.exe
      - veloren-voxygen.exe
      - assets/
      - LICENSE
      - libgcc_s_seh-1.dll
      - libstdc++-6.dll
      - libwinpthread-1.dll
    expire_in: 1 week

linux-aarch64:
  extends:
    - .tlinux-aarch64
    - .release
  tags:
    - veloren/veloren
    - build
    - publish
    - trusted

windows-x86_64:
  extends:
    - .twindows-x86_64
    - .release
  tags:
    - veloren/veloren
    - build
    - publish
    - trusted


......
```

Figure 17: Excerpt of the `.gitlab-ci.yml` file adopted by `veloren/veloren`

the detection results refer to, and (iii) in the same directory there is also the folder where, for each build in the history of the project, it is possible to find the log traces of the jobs belonging to it. Moreover, as detailed in Section 5.3, to properly run the tool, it is mandatory to provide the file storing the relevant information in terms of what are the previous builds executed by GitLab CI/CD, together with the textual file storing the relevant information in terms of the single jobs being executed by each previous build.

Before going deeper into the example, it is important to specify what is the relevant information for the builds and the related jobs that we are going to store, focusing more on the type of data used by the detection strategies currently implemented. As regards the builds (i.e., pipelines in GitLab CI/CD), we will store the pipeline identifier, the identifier of the commit that has triggered the build, the name of the branch to which the change belongs, the overall build outcome (e.g., failed, success or errored), the event that has triggered the whole CI/CD process (e.g., push, schedule, or merge_request_event), the overall build execution time in seconds, the time between the creation and the start of the process (i.e., how much longer the build has been in a waiting status), the percentage of code coverage, together with how many test cases have ended with a success, error, failure and also how many of them have been skipped. As regards the jobs, instead, we will store their identifier together with the identifier of the pipeline to which the job belongs, the outcome of their execution (e.g., failed, success, or errored), the name of the stage to which the job belongs (e.g., build, cleanup, check, deploy or publish), together with the name of the job, the percentage of code coverage if the job executes some testing tasks, whether or not the job has been allowed to fail, its duration in seconds, and the number of seconds within which the job has been in a waiting status.

At this point, let's run our detection tool assuming as last pipeline being executed the one having as identifier 557256345[6], without specifying the values for the two optional parameters. The latter means that we will use as time window the last month and as stable release branch the master branch:

```
python ./build_log_analyzer.py ./../veloren_builds_summary.csv
./../veloren_jobs_summary.csv ./../veloren_logs/ 5 5 ./../veloren/
./../557256345.txt
```

The pipeline has been triggered by a push event occurred on the stable release branch (i.e., master) and ended with a success status. For this reason, only three out of six detectors are enabled. Specifically:

- *Pipeline steps/stages are skipped arbitrarily*. In this case, by looking at the previous build occurring on the same branch and being triggered by the same event (pipeline_ID = 556108787), the BUILD LOG ANALYZER does not find any change in the build outcome, indeed the previous build was already successful. For this reason the tool ends up finding that there are no skipped tests neither environments;
- *Build Time too long*. There are 38 builds being triggered on the master branch with a push event in the month before June 5, 2022 that results in a median execution time of 3,374 seconds. Since the last pipeline triggered on master as an overall execution time less than the median execution time, the tool ends up finding that there is no increasing trend in terms of the overall duration;
- *Inappropriate cache handling*: there are 11 jobs belonging to the pipeline which duration varies in the interval [3-2,179] seconds. Table 1, reports, for each stage and for each job the overall duration and also, for the stages with more than one job the median execution time computed considering the set of jobs belonging to it. Based on this information and considering the tolerance percentage of 5% given as input when running the detection tool, we obtain that the **pages** job in the publish stage, and the **windows-x86_64**, **linux-aarch64** together with the **benchmarks** jobs in the build stage have a duration that is greater than the median jobs duration belonging to the same stage in the pipeline plus the percentage tolerance. For these four jobs (highlighted in red in Table 1), the BUILD LOG ANALYZER will parse the `.gitlab-ci.yml` file and verifies whether or not their configurations rely on the `cache` feature. Figure 17 shows an excerpt of the `.gitlab-ci.yml` file, from which it is possible to state that the CI/CD process in place does not rely on the caching feature.

Based on what being said before, the BUILD LOG ANALYZER provides as outcome a textual file which content is shown in Figure 18. As it can be seen from the figure, the tool correctly identifies the presence of the four jobs, namely **pages**, **windows-x86_64**, **linux-aarch64** and **benchmarks** which duration could be probably decreased by relying on the `cache` feature.

[6]https://gitlab.com/veloren/veloren/-/pipelines/557256345

Table 1: Jobs duration for pipeline_ID = 557256345

Stage	Job Name	Duration [s]
deploy	pages:deploy	3
publish	pages	852
	docker	157
Median publish		504.5
build	macos-aarch64	1,161
	macos-x86_64	1,144
	windows-x86_64	2,179
	linux-aarch64	2,130
	linux-x86_64	2,031
	coverage	1,859
	benchmarks	2,066
	translation	443
Median build		1,945

```
Think about using the caching to reduce the duration of the following jobs:
================================================================================:
pages
linux-aarch64
windows-x86_64
benchmarks
====================================
```

Figure 18: Outcome of the BUILD LOG ANALYZER for the pipeline_ID = 557256345

Let's now consider what is the outcome of the BUILD LOG ANALYZER when it is enacted on a build ending with a failing status (i.e., pipeline_ID = 551736549[7]). Similarly to the previous case, we will run the tool relying on the default value for the two optional parameters:

```
python ./build_log_analyzer.py ./../veloren_builds_summary.csv
./../veloren_jobs_summary.csv ./../veloren_logs/ 5 5 ./../veloren/
./../551736549.txt
```

The pipeline has been triggered by a push event occurred on the stable release branch (i.e., master) ending with a failing status. For this reason, five out of six detectors are enabled, two of which have also been presented in the previous running example and for this reason are not detailed in the following, i.e., *Build Time too long* and *Inappropriate cache handling*. Specifically:

- *A stable release branch is missing*. In this case, the tool selects the total number of builds occurring on the master branch in the month before May 31, 2022 and discriminates between the ones failing/errored and the ones ending with a success. In the time window considered there are a total of 67 builds of which 24 end with a failure and the remaining 43 are successful. Based on the previous data, the tool points out that *"35.8% of failures have been experienced on the master branch. At most 5 consecutive failures have been experienced"* (see Figure 18);
- *Different jobs consistently give a different outcome*. In this case, the detection strategy starts seeing whether all the jobs involved in the pipeline are failing or else there are only some of them that fail. For the pipeline_ID = 551736549, among the 10 jobs only one is failing, i.e., **linux-x86_64** with a specific

[7]https://gitlab.com/veloren/veloren/-/pipelines/551736549

```
A stable release branch could be missed:
=======================================:
35.8\% of failures have been experienced on the master branch

At most 5 consecutive failures have been experienced
=======================================
```

Figure 19: Outcome of the BUILD LOG ANALYZER for the pipeline_ID = 551736549

compilation error: *"error: could not compile veloren-voxygen"*. For this reason, the tool compares the current build outcome with the one related to the build being executed immediately before by the same event, i.e., pipeline_ID = 551666887[8]. By comparing the build outcome the tool recognizes that the previous build was successful meaning that the root cause of the failures is less likely to be related to the presence of an unstable environment;

- *Build stages are not properly ordered*: Based on the CI/CD process in place for the *veloren/veloren* project, see Figure 16, this specific CI/CD misuse cannot be experienced, specifically if we look at how the stages are sequentially ordered on the master branch (the one we are considering in this example).

Based on what being said before, the BUILD LOG ANALYZER provides as outcome a textual file which content is shown in Figure 19. The only symptom of a CI/CD misuse being reported is related to the presence of a "high" percentage of build failures on the release branch which might hinder its stability.

6 Fat Docker Detection and De-Bloating

This section details an automated tool able to identify unused dependencies in a Docker image. It is also able to produce a de-bloated version of the original image, i.e., a new image with all unneeded dependencies removed. This tool is provided standalone, thus it can run locally, and then a developer, based on the output, can decide to update the docker image used in the project with the new one produced by our tool.

6.1 Which bad practice to detect?

Cyber-Physical Systems (CPSs) are composed of heterogeneous software and hardware units. A peculiar characteristic of CPSs is that they receive inputs from hardware components, e.g., from sensors, and, in turn, send their output to other pieces of hardware, e.g., actuators.

Applying CI/CD in the context of CPSs can be particularly useful, given that many of those systems can have safety-critical properties, and given their interaction with hardware or simulators during the development phase.

However, due to the complex nature of the system, developers could encounter challenges, barriers, and bad practices during CI adoption in the context of CPSs. Previous studies investigated challenges and bad practices faced when applying CI in CPSs development, and also specific solutions to the encountered problems [35].

For instance, simulators are a key asset for CPSs development but their usage and integration could be challenging. Indeed, if they are not designed to work within a CI pipeline, their integration could get more challenging. Thus, this could have an impact on setting up the build run-time environment, which set may become difficult to perform and maintain, but also overly slow.

[8] https://gitlab.com/veloren/veloren/-/pipelines/551666887

In these cases, the preferred solution is containerization and it can be easily introduced by setting up families of predefined images specifically tailored for CPS run-time/simulation needs. One of the most popular containerization tools in current DevOps practice is Docker [1, 39]. It allows encapsulating software packages into containers, which can run on any system. Docker containers can be considered the standard de-facto for the industry. During the last years, Docker has been widely used by developers. This growing use is due to the fact that Docker technology supports a convenient process able to create and build containers. It promotes close cooperation between developers and operations teams, allowing continuous software delivery.

Docker images have multiple layers, each one originates from the previous layer but is different from it. The layers speed up Docker builds while increasing reusability and decreasing disk use. Image layers are also read-only files. Once a container is created, a writeable layer is added on top of the unchangeable images, allowing a user to make changes.

In this way, it is really simple to build a new image, but this can lead to adding unnecessary bloat to it. For instance, images created by adding a new layer, could often include unneeded dependencies that increase the container size. Furthermore, developers usually do not clean the dependencies (that are not useful anymore) as our deliverable D3.2 pointed out [27]. This bad practice leads to huge packages that are time-consuming to deploy.

Thus, our aim is to automatically find all unneeded dependencies increasing the image size, and remove them to produce a de-bloated version of it. To achieve this goal we develop a tool that takes as input a Docker image and is able to identify and remove all unused packages.

6.2 Infrastructure

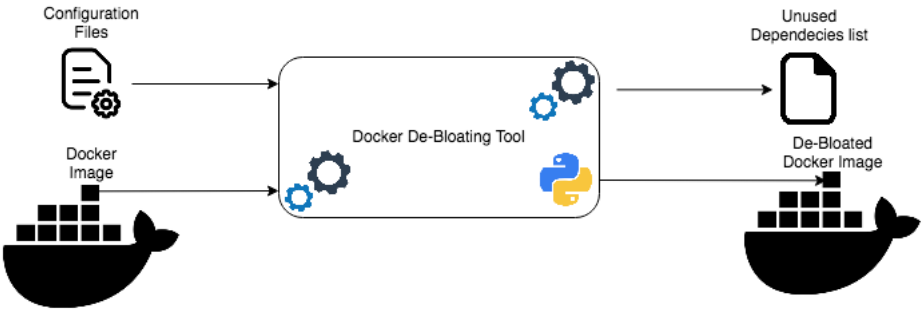

Figure 20: Workflow of the Docker image debloating approach.

Figure 20 shows a high-level view of the proposed tool. It is named Docker De-Bloating and takes as input a Docker image and a configuration file. The outputs are a file containing all unused dependencies list, and a de-bloated version of the original image. Our tool is provided standalone and developers can run it when needed. Docker De-Bloating tool is written in Python, and it needs Docker demon installed on the machine.

To find all unused dependencies and to produce a de-bloated Docker image, our tool goes through several consecutive and iterative steps summarized in the following:

1. retrieve Docker image;
2. collect all installed packages into a file by running a command inside the container;
3. prune the resulting installed packages;
4. check into application source code if some of the installed packages are used and return the list of used packages;

5. update the list of installed packages removing all used packages resulting from the previous step;
6. the complete list is used to create the packages dependencies graph;
7. determine reachability with transitive closure;
8. for any package installed and taking into account the related dependencies the tool tries to remove the packages;
9. after removing, the tool tries to run the application. Specifically, if it runs the tool will remove the package and all related dependencies, otherwise discard the packages, and put them into a list of needed dependencies;
10. when the removing actions are all performed the tool provides as output a file containing the list of removed dependencies and the corresponding de-bloated image.

More in detail, the first step performed by our tool is **Retrieve the Docker image**. Most of the available images are created using base images hosted in the Docker Hub registry. Developers can easily pull an image and try to define and configure it for their own purposes. Then, they can share the modified image pushing it to the Docker Hub registry. Thus, to find and share container images developers can use the hosted repository service provided by Docker, i.e., Docker Hub. Since the main functionality of our tool is to de-bloat Docker images, the first step aims at downloading from Docker Hub a particular image using the command `docker pull IMAGE_NAME`. Thus, through this command, our tool first checks if the Docker image (IMAGE_NAME) exists locally. If so, it checks for updates and downloads a newer version of the image. Instead, if the image does not exist locally then the Docker daemon connects to the public registry and pulls the image.

Once the image is retrieved our tool goes to the second step, i.e., **Collect all installed packages**, which retrieves and stores the list of all installed tools, packages, and software into a text file, following the alphabetical order. Specifically, the docker run command first creates a writeable container layer over the specified image and then starts it using the specified command. The complete command is shown in the following:

```
docker run --name RUNNING-NAME -i -u0  IMAGE-NAME STRING-COMMAND
```

where RUNNING-NAME is the name assigned to the running image, IMAGE-NAME is the name of the image of which we want to know the complete list of all installed packages and STRING-COMMAND is exactly the `apt list -installed` command showing all installed packages. The option `-name` assigns a name to the container, the option `-i` keeps the standard input open even if not attached, finally the option `-u0` defines the user id as zero. At the end of the execution of the previous commands, a file containing the packages list is created. It should be underlined that the file has the following format:

```
acl/now 2.2.53-10 amd64 [installed,local]
adduser/now 3.118 all [installed,local]
apt/now 2.2.4 amd64 [installed,local]
base-files/now 11.1+deb11u3 amd64 [installed,local]
base-passwd/now 3.5.51 amd64 [installed,local]
...
```

To obtain the pure name of a package, a cleaning step is needed, i.e., **Prune the list installed packages**. The main goal of this step is to simply isolate the name of any installed packages, by removing all unneeded information, e.g., the package version or its location.

Once obtained this pruned list of packages, the tool continues its execution with the next step, which is the **Search for packages** within the source code of the application. We start from the assumption that: if an

installed package is used by the application, it is not suitable for removal. To do so, our tool uses a regular expression to check if the name of the package is used somewhere in the source code or configuration files. In affirmative cases, our tool creates a list of un-touchable packages. A the end of this step our tool can proceed with its execution taking into account two lists of packages: a list with un-touchable dependencies, and a list with all installed packages that are suitable for removal.

Starting from these two lists, our tool runs the sixth step: **Create dependencies graph**. The latter is needed since dependencies are supporting packages required for the proper working of an application. For instance, when downloading an application on Ubuntu, `apt` will install some additional packages, in addition to the primary packaged application. Hence, a package can also have other packages as its dependencies in the form of a hierarchical structure. Our tool uses `apt` to get a list of dependencies associated with a package. The basic syntax of the command is the following:

```
apt depends PACKAGE-NAME
```

where PACKAGE-NAME is replaced each time with the name of a package contained in the list of packages suitable for removal. The above command provides the list of dependencies and the output also includes recommended and suggested packages that someone can install alongside PACKAGE-NAME. For any installed package and the list of its dependencies, our tool starts to build the (directed) dependencies graph, i.e., a matrix nxn containing 1 in position i, j if exists a dependency between the packages i and j respectively. Using this algorithm, we are able to create the dependencies graph represented as an adjacency matrix. After that, considering the previously created directed graph, our tool finds out if a vertex j is reachable (i.e., there is a path from node i to j) from another node i for all node pairs (i, j) in the graph. This step ends up with the definition of the reachability matrix, i.e., the transitive closure of a graph.

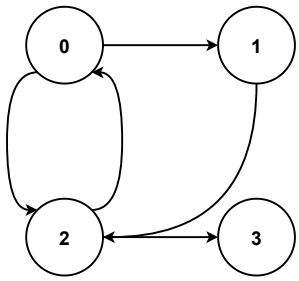

Figure 21: A simple graph example.

As a simple example, consider the graph in Figure 21. More precisely, every arrow indicates that exist a path between nodes A and B, e.g., 0 and 1, and it is direct. In our scenario, every node is a package and an arrow indicates a dependency between two nodes. The adjacency matrix of the graph in Figure 21 is shown in Table 2.

Table 2: Adjacency matrix for the graph in Figure 21

	Node-0	**Node-1**	**Node-2**	**Node-3**
Node-0	0	1	1	0
Node-1	0	0	1	0
Node-2	1	0	0	1
Node-3	0	0	0	0

This adjacency matrix has boolean values. When there is a value 1 for node i to node j, it means that there is a direct connection from i to j. These binary relations tell us only that node A is connected to node B and that node B is connected to node C, etc. Starting from these relations, it is possible to determine reachability, i.e., if

it is possible to reach a node A starting from node B using a finite number of steps. To **determine reachability** it is necessary to compute the **transitive closure** (the 7th step of our tool). Transitive closure constructs the output graph from the input graph. After the transitive closure building, it is possible to determine that node D is reachable from node A. In our specific case, this means that given a package and through transitive closure we are able to know all dependencies belonging to that package, i.e., how packages are dependent on each other. Table 3 shows the transitive closure of the graph in Figure 21.

Table 3: Transitive Closure of the graph in Figure 21

	Node-0	**Node-1**	**Node-2**	**Node-3**
Node-0	1	1	1	1
Node-1	1	1	1	1
Node-2	1	1	1	1
Node-3	0	0	0	0

Once this transitive closure is computed, our tool starts removing dependencies. To identify the best candidate for the removal process our tool uses the following heuristic:

- it searches isolated nodes and starts to remove them. Isolated nodes are nodes without incoming and outgoing arcs, i.e., packages without any dependency;
- once isolated nodes list is empty, it looks to the columns containing only 0 values and starts to remove them (such nodes have only outgoing arcs, so there is no package that depends on it) ;
- finally, it tries to remove all remaining dependencies, i.e., the columns being not zero.

A node with only outgoing arcs (like root) can be easily identified by looking at the columns of the matrix: the column contains only cells having “0” values.

Now, we look in detail at the steps involved in the **effective removal of the package**, step number 8. The removal function runs the following workflow:

1. remove the package given its name;
2. create a new image applying this change;
3. run new container with application start command;
4. check if the image is running;
   - if yes, stop container and put the removed package into removable list;
   - if no, put the package into not-removable list and restore the image with that having the package installed.

More precisely, to remove a package, our tool executes the following command:

```
docker run --name RUNNING-NAME -i -u0  IMAGE-NAME STRING-COMMAND
```

It is the same as above and allows to run a command ($STRING-COMMAND$) inside a running container. Differently, the required command, in this case, is the apt related command allowing to remove a package given its name. The string ($STRING-COMMAND$) used for the command is shown in the following:

```
apt-get --purge remove PACKAGE-NAME -y
```

where PACKAGE-NAME simply is the name of the package that we want to remove. The option $-y$ is needed since we want to automatically give "yes" as the answer to all prompts and run non-interactively.

After that, our tool creates a new image from the changed container, i.e., the container without the package. This is done using the Docker command:

```
docker commit IMAGE-NAME NEW-IMAGE-NAME
```

where IMAGE-NAME is the image with the package removed and NEW-IMAGE-NAME is the new name that we want to assign to the new image. Then, we have to verify if the encapsulated application works also without the package. To do so, our tool runs the application through the following command:

```
docker run --name RUNNING-NAME  -d  NEW-IMAGE-NAME CMD-APP -force
```

The arguments are the name assigned to the running image (RUNNING-NAME), and the option $-d$ forces the container to run in the background. CMD-APP is the command to start the application, i.e., the command reported into Dockerfile and NEW-IMAGE-NAME is the name of the image that we want to run. Finally, we also use the option `-force` to force the overwrite of the target image.

Finally, to confirm the removal we have to check if the image works also without the package. To verify its correct working we have to check if the container is running. To do so, we list all the running images with the Docker command `docker ps -a` and then, check if our de-bloated `RUNNING-NAME` image status is equal to Up. If yes, our tool stops the container through `docker stop` command and stores the removed package into the removable list. If the application status is not equal to Up, it means that the removed package is fundamental for the correct working of the application. In this case, our tool restores the package into the application and puts its name into the not-removable list.

Removal attempts will be performed until the list of removable packages is empty. Thus, at the end of the execution of our tool, we provide two different lists of packages: one containing the removable dependencies and the other containing the needed ones. Furthermore, our tool provides also a de-bloated image with all removable dependencies removed using docker-squash tool (available on GitHub[9]).

6.3 User Manual

The aim of Docker De-bloating is to automatically find all unneeded packages installed into an encapsulated application increasing the image size. Docker De-bloating removes all unneeded packages and produces a de-bloated version of the image. The tool takes as input a Docker image and a configuration file. Furthermore, it is able to identify and remove all unused packages and as output provides three files: one containing all removable dependencies, another containing all needed dependencies, and the file of the de-bloated image.

Installation

Docker De-bloating tool must run as root since it runs docker commands and Docker demon needs root privilege.

So, type in the terminal the command `sudo su` to obtain root privilege. Then, go to the directory where `debloating.py` file is located and type the command described into Section Usage.

Regarding dependencies, to correctly run the Docker Debloating tool you have to install:

- Python

[9] `https://github.com/goldmann/docker-squash`

- Docker
- `docker-squash` tool, available on GitHub: `https://github.com/goldmann/docker-squash`

Furthermore, you have to install Python requirements by running:

```
python3 -m pip install -r requirements.txt
```

Usage

The command to run Docker De-bloating tool has the following syntax:

```
debloating.py [-h] -i <imageFile> -c <configurationFile>
```

where `Required Arguments` are:

```
-i, -image        Docker image name
-c, -conf_file    Path to configuration file
```

Instead, the `Optional Argument` is:

```
-h, -help    Show the help message and exit
```

Note that, the `Configuration File` has the following format:

```
PRJ-PATH=    followed by the path of the source code of the application
     CMD=    followed by the command to start the application
    TEST=    followed by command to run test cases
```

6.3.1 Running Example

In this section, we report a simple running example to show how our De-Bloating tool works. We provide two sequence diagrams following the execution of our tool. They show a simple usage and running scenario. For simplicity, where possible we aggregate the execution steps (explained in Section 6.2) of our tool to better explain its workflow.

The running example tries to reduce the size of the Apache Solr image[10]. To run our tool a user has to go to the directory where it is located and type the following command:

```
debloating.py -i solr -c solr-conf.txt
```

where `-i solr` indicates the image name and `-c solr-conf.txt` is the configuration file needed for correct running of our tool. The `solr-conf.txt` has the following parameters:

```
PRJ-PATH=solr/
CMD=solr-foreground
TEST=./gradlew check
```

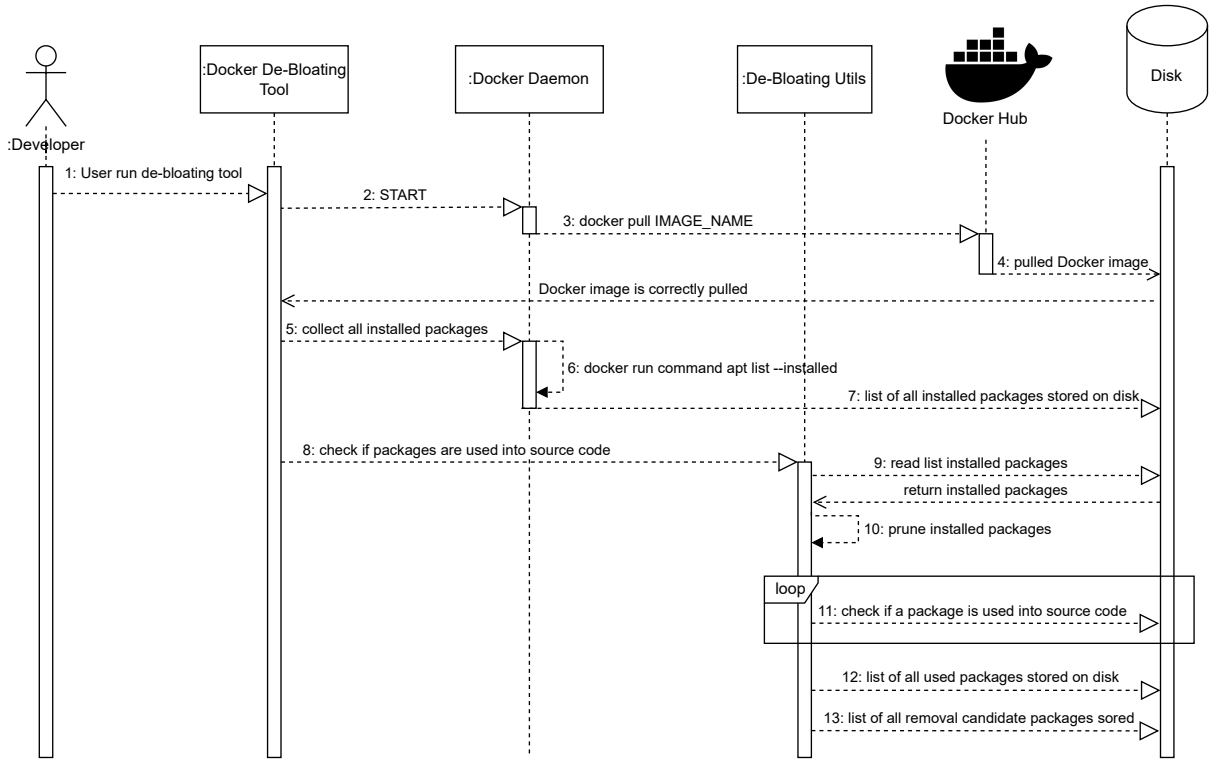

Figure 22: Sequence Diagram collecting all installed packages into Docker Image and checking if some of them are used in the source code of the application.

This file specifies where the source code is located, which is the command to run the application inside the container and the command to run application tests.

Figure 22 shows the initial steps performed by our tool. More in detail, the sequence diagram shows how the step from 1 to 5 are performed by the Docker De-Bloating tool (see Section 6.2).

In this case, the main interacting actors are:

- A developer: the user running the command to use the Docker De-Bloating tool;
- Docker De-Bloating: the main core of our tool;
- Docker Daemon: the Docker service allowing to run Docker commands;
- De-bloating Utils: support functions useful to perform some tasks, e.g., file pruning;
- Docker Hub: Docker hosted repository service to find and share container images;
- Disk: user space where the tool saves all required information.

Then the tool execution proceeds as depicted in the sequence diagram of Figure 23. It shows the steps involved in the package removing process and steps involved in de-bloating. Basically, Figure 23 details the step from 6 to 10 performed by Docker De-Bloating tool (see Section 6.2).

At the end of the execution, our tool provides two different files, one containing needed packages for the correct work of the application and another one containing all removed packages. Results show that Solr image has installed 155 packages. After the run of our tool, we successfully remove 61 packages, and the de-bloated image still works with only 94 packages.

[10] `https://hub.docker.com/_/solr`

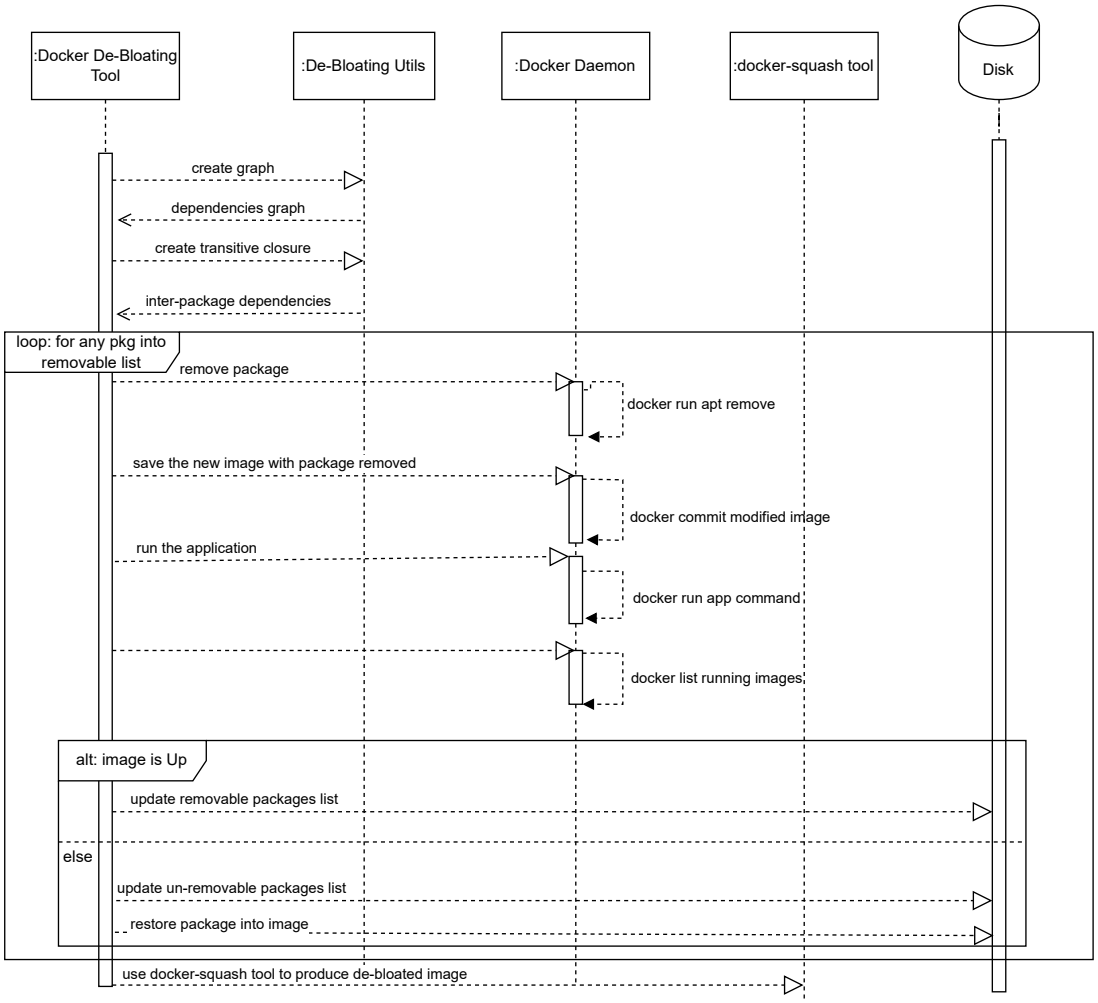

Figure 23: Sequence Diagram removing all installed packages into Docker Image and producing the de-bloated image with un-needed packages removed.

7 Conclusion

In this deliverable we have proposed a set of tools for the automated detection of bad practices in Continuous Integration (CI) and Delivery (CD) pipelines. More specifically, the deliverable describes:

1. A *CI/CD Configuration Analyzer*, determining bad practices in the configuration files of CI/CD pipelines;
2. A *Build Log Analyzer*, analyzing the CI/CD build log and determining the possible presence of bad practices and pipeline decay by observing the build execution history;
3. A *Fat Docker detection and de-bloating tool*, which identifies the occurrence of unused dependencies in Docker images and generated a flattened, reduced image.

The tools, and in particular the *CI/CD Configuration Analyzer* and the *Build Log Analyzer*, have been conceived so to be easily extended with further detection rules. For what concerns the *Fat Docker detection and de-bloating tool*, it can be extended in the future, by integrating (i) a test generator tool, and (ii) heuristics to reduce further types of dependencies not considered in the current implementation as well as legacy data and other redundancies.

All tools will be released as open-source projects under the MIT license.

Future work aims at complementing the developed CI/CD bad practice detectors with tools aimed at supporting the developers in the improvement of their pipelines, and/or in the development of pipelines for achieving certain CI/CD goals.

References

[1] Docker. `https://www.docker.com`; accessed June-13-2022.

[2] GitLab CI/CD Documentation. `https://docs.gitlab.com/ee/ci/`, Accessed Jun 13 2022".

[3] Afsoon Afzal, Claire Le Goues, Michael Hilton, and Christopher Steven Timperley. A study on challenges of testing robotic systems. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 96–107. IEEE, 2020.

[4] Jürgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C Gall. An empirical analysis of the docker container ecosystem on github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 323–333. IEEE, 2017.

[5] Paul M. Duvall. Continuous integration. patterns and antipatterns. *DZone refcard #84*, 2010.

[6] Paul M Duvall, Steve Matyas, and Andrew Glover. *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[7] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. Understanding flaky tests: the developer's perspective. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, pages 830–840, 2019.

[8] Keheliya Gallaba and Shane McIntosh. Use and misuse of continuous integration features: An empirical study of projects that (mis) use travis ci. *IEEE Transactions on Software Engineering*, 46(1):33–50, 2018.

[9] Joshua Garcia, Yang Feng, Junjie Shen, Sumaya Almanee, Yuan Xia, and Qi Alfred Chen. A comprehensive study of autonomous vehicle bugs. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 385–396, 2020.

[10] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. Trade-offs in continuous integration: assurance, security, and flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 197–207, 2017.

[11] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016.

[12] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[13] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, 23(1):384–417, 2018.

[14] Wing Lam, Patrice Godefroid, Suman Nath, Anirudh Santhiar, and Suresh Thummalapenta. Root causing flaky tests in a large-scale industrial setting. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 101–111. ACM, 2019.

[15] Wing Lam, Stefan Winter, Anjiang Wei, Tao Xie, Darko Marinov, and Jonathan Bell. A large-scale longitudinal study of flaky tests. *Proc. ACM Program. Lang.*, 4(OOPSLA):202:1–202:29, 2020.

[16] Zhigang Lu, Jiwei Xu, Yuewen Wu, Tao Wang, and Tao Huang. An empirical case study on the temporary file smell in dockerfiles. *IEEE Access*, 7:63650–63659, 2019.

[17] Christian Macho, Shane McIntosh, and Martin Pinzger. Automatically repairing dependency-related build breakage. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 106–117. IEEE, 2018.

[18] Torvald Mårtensson, Daniel Ståhl, and Jan Bosch. Continuous integration applied to software-intensive embedded systems–problems and experiences. In *International Conference on Product-Focused Software Process Improvement*, pages 448–457. Springer, 2016.

[19] Helena Holmström Olsson, Hiva Alahyari, and Jan Bosch. Climbing the" stairway to heaven"–a mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In *2012 38th euromicro conference on software engineering and advanced applications*, pages 392–399. IEEE, 2012.

[20] Gustavo Pinto, Breno Miranda, Supun Dissanayake, Marcelo d'Amorim, Christoph Treude, and Antonia Bertolino. What is the vocabulary of flaky tests? In *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, pages 492–502, 2020.

[21] Akond Rahman, Chris Parnin, and Laurie Williams. The seven sins: Security smells in infrastructure as code scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 164–175. IEEE, 2019.

[22] Akond Rahman and Laurie Williams. Characterizing defective configuration scripts used for continuous deployment. In *2018 IEEE 11th International conference on software testing, verification and validation (ICST)*, pages 34–45. IEEE, 2018.

[23] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. Cimplifier: automatically debloating containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 476–486, 2017.

[24] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. Does your configuration code smell? In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 189–200. IEEE, 2016.

[25] Daniel Ståhl and Jan Bosch. Continuous integration flows. In *Continuous software engineering*, pages 107–115. Springer, 2014.

[26] Martin Törngren and Ulf Sellgren. Complexity challenges in development of cyber-physical systems. In *Principles of Modeling*, pages 478–503. Springer, 2018.

[27] University of Sannio, ZHAW. D3.2 - Catalog of good and bad practices of DevOps for CPS., 2021. COSMOS Delivery, Released on December 2021.

[28] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar T. Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in github. In *ESEC/SIGSOFT FSE*, pages 805–816. ACM, 2015.

[29] Carmine Vassallo, Sebastian Proksch, Harald C Gall, and Massimiliano Di Penta. Automated reporting of anti-patterns and decay in continuous integration. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 105–115. IEEE, 2019.

[30] Carmine Vassallo, Sebastian Proksch, Harald C. Gall, and Massimiliano Di Penta. Automated reporting of anti-patterns and decay in continuous integration. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 105–115. IEEE / ACM, 2019.

[31] Carmine Vassallo, Sebastian Proksch, Anna Jancso, Harald C Gall, and Massimiliano Di Penta. Configuration smells in continuous delivery pipelines: a linter and a six-month study on gitlab. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 327–337, 2020.

[32] Carmine Vassallo, Sebastian Proksch, Timothy Zemp, and Harald C Gall. Un-break my build: Assisting developers with build repair hints. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 41–4110. IEEE, 2018.

[33] Dinghua Wang, Shuqing Li, Guanping Xiao, Yepang Liu, and Yulei Sui. An exploratory study of autopilot software bugs in unmanned aerial vehicles. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 20–31, 2021.

[34] Fiorella Zampetti, Salvatore Geremia, Gabriele Bavota, and Massimiliano Di Penta. CI/CD pipelines evolution and restructuring: A qualitative and quantitative study. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2021, Luxembourg, September 27 - October 1, 2021*, pages 471–482. IEEE, 2021.

[35] Fiorella Zampetti, Vittoria Nardone, and Massimiliano Di Penta. Problems and solutions in applying continuous integration and delivery to 20 open-source cyber-physical systems. In *MSR '22: 19th International Conference on Mining Software Repositories, Pittsburgh, PA, USA, 18-20 May - Virtual, 2022*, page to appear, 2022.

[36] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald C. Gall, and Massimiliano Di Penta. An empirical characterization of bad practices in continuous integration. *Empir. Softw. Eng.*, 25(2):1095–1135, 2020.

[37] Ahmed Zerouali, Valerio Cosentino, Tom Mens, Gregorio Robles, and Jesus M Gonzalez-Barahona. On the impact of outdated and vulnerable javascript packages in docker images. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 619–623. IEEE, 2019.

[38] Ahmed Zerouali, Tom Mens, Gregorio Robles, and Jesus M Gonzalez-Barahona. On the relation between outdated docker containers, severity vulnerabilities, and bugs. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 491–501. IEEE, 2019.

[39] Hong Zhu and Ian Bayley. If docker is the answer, what is the question? In *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pages 152–163. IEEE, 2018.

[40] Behrouz Zolfaghari, Reza M. Parizi, Gautam Srivastava, and Yoseph Hailemariam. Root causing, detecting, and fixing flaky tests: State of the art and future roadmap. *Softw. Pract. Exp.*, 51(5):851–867, 2021.