CI/CD Pipelines Evolution and Restructuring: A Qualitative and Quantitative Study

Fiorella Zampetti,Salvatore Geremia
University of Sannio, Italy
{name.surname}@unisannio.it

Gabriele Bavota
Università della Svizzera Italiana, Switzerland
gabriele.bavota@usi.ch

Massimiliano Di Penta
University of Sannio, Italy
dipenta@unisannio.it

***Abstract*—Continuous Integration and Delivery (CI/CD) pipelines entail the build process automation on dedicated machines, and have been demonstrated to produce several advantages including early defect discovery, increased productivity, and faster release cycles. The effectiveness of CI/CD may depend on the extent to which such pipelines are properly maintained to cope with the system and its underlying technology evolution, as well as to limit bad practices. This paper reports the results of a study combining a qualitative and quantitative evaluation on CI/CD pipeline restructuring actions. First, by manually analyzing and coding 615 pipeline configuration change commits, we have crafted a taxonomy of 34 CI/CD pipeline restructuring actions, either improving extra-functional properties or changing the pipeline's behavior. Based on such actions, we have developed a metric extractor for Travis-CI pipelines, which extracts 16 indicators of how a pipeline evolves. The analysis of the pipeline evolution for 4,644 projects using Travis-CI and developed in 8 programming languages shows how some pipeline components, such as jobs and steps tend to change more often than others, but also the Docker adoption by the projects increases over time.**

***Index Terms*—Continuous Integration and Delivery; Restructuring; Empirical Study**

I. Introduction

Continuous Integration (CI) [18], [21] aims at automating the build process on dedicated servers, with the goal of early error discovery [19], [26], [46], as well as of an overall quality assessment and improvement. For such reasons, CI is quite adopted in industry [22] and open source [32]. Its evolution, Continuous Delivery (CD), helps to accomplish short cycle releases [16], [34]. Previous empirical research has indicated how CI/CD allows for early discovery of defects [32], increases developer productivity [48] and speeds up release cycles [26].

Previous work has also indicated challenges and barriers in CI/CD adoption [31] implying that setting up a CI/CD pipeline is not an easy task. Moreover, CI/CD can be wrongly applied, and this may limit its effectiveness and may introduce maintainability problems, as it has been highlighted in several catalogs of CI/CD bad practices [27], [34], [53].

Like any piece of software, CI/CD pipelines undergo evolutionary changes because of several reasons, for example:

- bad practices have been adopted when setting up the pipeline (detected also thanks to existing smell detection approaches [29], [49], [50]), and they need to be avoided;
- the pipeline becomes hard to maintain and understand, and therefore a restructuring activity is needed;
- some parts of the pipelines become unnecessary and can be removed, or some others (*e.g.,* testing environments) become obsolete and should be upgraded/replaced;
- performance bottlenecks need to be resolved, *e.g.,* by parallelizing or restructuring some pipeline jobs; or
- in general, the pipeline needs to be adapted to cope with the evolution of the underlying software and systems, including technological changes (*e.g.,* changes of architectures, operating systems, or library upgrades).

We report the results of an empirical qualitative and quantitative study investigating how CI/CD pipelines of open source projects evolve, and, in particular, the extent to which their configuration files undergo restructuring actions. We considered a set of 8,000 non-forked projects from GitHub relying on the usage of at least one CI/CD infrastructure (*i.e.,* AppVeyor, Bamboo, Circle-CI, GitLab-CI, Jenkins, Semaphore, Travis-CI, and Wercker), ordered by their number of forks.

From these projects, we first conduct a qualitative study on 615 commits, aimed at understanding what kinds of restructuring or refactoring actions are usually performed on pipelines. The study resulted in a 3-level catalog of CI/CD restructuring actions.

Based on the identified restructuring actions, we then develop a metric extractor for Travis-CI pipelines, which extracts 16 different metrics, including, for example, the total number of jobs, the number of jobs that are allowed to fail, the type of notification mechanism being explicitly set, or the phases and their configuration inside the pipeline. Then, we use the metric extractor to analyze the evolution (253,492 commits) of 4,644 projects out of the initial set of 8,000, limiting to those using Travis-CI and having at least a commit modifying the pipeline configuration file and 100 commits from the Travis-CI adoption. We report and discuss the extent to which CI/CD pipelines change over time in terms of the identified metrics.

Finally, we show how the extracted metrics can be used as possible indicators of the restructuring actions in our catalog. The monitoring of the pipeline evolution can be not only used by developers to monitor it and trigger possible restructuring activities, but also as training material for recommenders aimed at automatically restructuring the pipeline.

II. Study Design

The *goal* of this study is to qualitatively and quantitatively investigate the extent to which CI/CD pipelines of open source

TABLE I
DISTRIBUTION OF PROJECTS USING EACH CI/CD INFRASTRUCTURE BY PROGRAMMING LANGUAGE

Language	AppVeyor	Circle-CI	GitLab	Jenkins	Semaphore	Travis-CI	Wercker
C	197	103	72	–	1	898	1
C++	322	152	50	–	–	861	3
C#	655	104	18	–	1	399	2
Java	45	138	16	2	1	904	5
JavaScript	91	199	5	2	–	852	3
PHP	53	152	13	1	–	931	–
Python	97	180	17	4	1	895	4
Ruby	60	154	11	–	1	883	2
Total	**1,520**	**1,182**	**202**	**9**	**7**	**6,623**	**20**

projects undergo restructuring activities. The *perspective* is of researchers interested to understand the CI/CD evolution process with the long-term goal of automatically recommending restructuring actions. The *context* consists (i) for the qualitative analysis, of a sample of 615 commits out of 2,383 candidate ones coming from 1,235 GitHub open source projects and having at least a CI/CD configuration file; (ii) for the quantitative part, of 253,492 commits from 4,644 projects using Travis-CI, changing the pipeline configuration file at least once and having more than 100 commits from Travis-CI adoption.

The data and scripts of our study are publicly available [52].

A. Research Questions

Similarly to production and test code, CI/CD pipelines, *i.e.,* CI/CD configuration files, evolve. They follow the evolutionary needs of the underlying systems, not only from a technological perspective but also in terms of developers' needs. While previous work has studied CI smells and antipatterns [27], [29], [34], [49], [50], [53], it lacks a deeper understanding of what kinds of restructuring actions (which go beyond smell/antipattern avoidance or removal) are applied in CI/CD pipelines. Therefore, we formulate our first research question:

RQ_1: *What are the restructuring operations occurring in CI/CD pipelines?*

Knowing the restructuring actions applied to CI/CD configuration files, it is possible to derive a set of metrics for tracking their evolution over time as previously done for tracking the evolution of other software artifacts. For example, it is possible to trace the evolution in terms of how many build jobs are defined, how many of them are allowed to fail, the total number of (global) environment variables, etc.. Based on the evolution of such metrics, we answer our second research question:

RQ_2: *How do CI/CD pipelines evolve over time?*

Through such metrics it will be possible to understand what kinds of CI/CD pipeline changes occur more frequently than others. Also, the metrics could be a starting point to devise approaches to recommend pipeline restructuring actions.

B. Context Selection

The study *context* for answering our $\mathbf{RQ}_1$ is made up of 615 commits, sampled from an initial set of 2,383 candidate CI restructuring commits, which have been extracted from 1,235 open source projects hosted on GitHub and written in 8 different programming languages, *i.e.,* JavaScript, Python, Java, Ruby, C++, PHP, C# and C. These are among the top 10 most popular languages on GitHub[1].

The projects to analyze have been selected as follows. For each programming language, we used the GitHub API to sort the whole set of GitHub projects using the number of forks as a proxy for their popularity, whereas we excluded projects being forked from others. After that, we ordered the projects from the most to the least popular, filtering out those that did not use a CI/CD infrastructure. More specifically, we cloned the project repository locally and searched for the presence of a predefined set of configuration files indicating the adoption of one of the following CI/CD infrastructures: AppVeyor, Bamboo, Circle-CI, GitLab, Jenkins, Semaphore, Travis-CI, and Wercker. We stopped the projects' selection once we collected the top 1,000 projects using at least one CI/CD infrastructure for each programming language, ending up with a total of 8,000 projects.

Table I shows the number of projects using each CI/CD infrastructure framework by programming language (no project using Bamboo was found). Note that the sum in the "Total" row is greater than 8,000 (9,563) because some projects rely on multiple frameworks. As highlighted in Table I, and consistently with previous work [32], Travis-CI is by far the most adopted CI/CD infrastructure in GitHub, followed by AppVeyor and Circle-CI. Unsurprisingly, only 202 projects use GitLab, and only 9 projects rely on Jenkins.

Looking at the number of forks in the 8,000 projects, only for C# we ended up sampling projects with less than 100 forks (min=40, median=106.5), while for the other programming languages the minimum number of forks is greater than 105.

Having collected the projects, we used Perceval [24] to retrieve their change history. Specifically, for each commit we stored the path of the files it impacts (*i.e.,* modifies, adds, deletes) and the commit message. Through regular expressions applied to the file names, we filtered out all commits that do not change the CI/CD configuration files. Moreover, we excluded commits not explicitly mentioning restructuring actions in the commit message. This was done by matching in the commit message the following five keywords devised after a preliminary skimming of projects' logs: *refact*, *restruct*, *cleanup*, *clean up* and *remodul*. Finally, we filter out commits where the ratio between the number of impacted files related to the CI/CD infrastructure and the total number of impacted files is lower than 0.3. This because we wanted to focus on commits that are likely to implement restructuring actions for the CI/CD process, excluding commits impacting a large number of files and likely representing tangled changes. As a result, we obtained 1,235 projects containing 2,383 candidate commits to inspect.

Concerning the *context* for our quantitative study ($\mathbf{RQ}_2$), we started from the 6,623 projects relying on Travis-CI. We then selected those having at least one commit modifying the

[1]https://madnight.github.io/githut/\#/pull_requests/2021/1

CI/CD configuration file and excluded those having less than 100 commits after the Travis-CI introduction (4,644 projects).

C. Qualitative Analysis of CI/CD Pipeline Restructuring

To derive a catalog of CI/CD restructuring actions ($\mathbf{RQ}_1$), we manually analyzed a total of 615 randomly selected (in proportion among different programming languages) commits from the 2,383 candidate ones resulting from the regular expression matching described in Section II-B. As it will be clearer below, the 615 commits were incrementally coded in a pilot phase (30), a first round involving 122 commits, a second-round done on a statistically-significant sample of 283 instances (confidence interval of $\pm 5\%$ confidence level 95%), and a saturation phase (180). The overall set of 615 is statistically significant with a confidence interval of $\pm 3.4\%$.

The manual analysis was conducted to classify each commit as (i) *false positive* (*e.g.,* cases where the "clean up" keyword matched in the commit message refers to a specific phase that can be set in the CI/CD configuration file, while not referring to a possible restructuring action); (ii) *no*: the restructuring action is not referring to the modified CI/CD infrastructure; and (iii) *yes*: the restructuring action mentioned in the commit message involves the CI/CD process. All the cases labeled as *yes* have been additionally categorized with a label describing the kind of restructuring action applied. The labels were created by following a card-sorting procedure, and specifically a cooperative (multiple annotators) hybrid card-sorting (partial set of predefined categories) [45].

We started from a predefined list of categories derived running a pilot study in which three of the authors (raters in the following) analyzed, in a plenary session, a random set of 30 commits modifying a CI/CD configuration file and explicitly referring to a restructuring activity in the commit message. The labeling procedure was performed in two different steps. First, a set of 122 commits has been validated independently by two raters who could assign one of the previously defined categories or add a new one when needed. The category assignment was based on the inspection of the commit diff, message, and related discussions such as linked pull requests or issues. The commits to inspect were organized into a Google sheet. When one of the raters had to categorize a commit, the Google sheet provided the list of categories defined so far by both of the two raters. This was done to help in using consistent naming while not introducing a substantial bias. Indeed, given the goal of our manual validation (*i.e.,* classify the type of restructuring action), the number of possible categories may be extremely high. Note that, while the defined categories were visible as a list, it was not possible to see which category was assigned by the other rater to the inspected commits.

At the end of the first round of the labeling process, an open discussion was performed by adding a third rater. In the open discussion, we checked all the commits that had at least one *yes* assigned (*i.e.,* at least one of the two raters found the commit to be relevant for our study). Given that the process of defining categories was incremental, it was not feasible to estimate the reliability of the study using inter-rater agreement. To limit agreement by chance, also cases where there was an agreement were confirmed and discussed with the third rater. Finally, the three raters worked together to derive a first version of the catalog consisting of a set of 40 CI/CD restructuring actions.

The same procedure has been used to classify the remaining 283 commits. Based on the 8 newly emerged categories, the three raters worked together to refine the previous version of the catalog that accounted, after this step, of 34 types of restructuring actions.

As the last step, we randomly choose a different set made up of 180 commits to check for saturation. Such a number has been defined with the final goal of obtaining $\sim$100 commits classified with a *yes*. Based on our experience, $\sim$45% of commits we automatically select are either false positives or not relevant for our study (*no* classification). The above commits have been labeled by two raters independently using as predefined categories the ones obtained after the catalog refinement step (with the possibility of defining new categories). In the end, an open discussion between them was performed for solving conflicts and agreeing on the newly added restructuring actions (*i.e.,* 3). By discussing with a third rater we ended up combining multiple restructuring actions related to either enabling or disabling a feature (*e.g.,* allow failure). We can conclude that no new categories emerged from our saturation check.

The final version of the catalog features 34 CI/CD restructuring actions organized into two (1st-level) categories, in turn specialized into seven (2nd-level) sub-categories.

D. Metrics/Facts extractor

To quantitatively investigate how the CI/CD process changes over time, we derived a set of metrics aimed at tracking its evolution by looking at CI/CD configuration files. We started from the catalog of CI/CD restructuring actions obtained as a result of RQ_1, and focused on the Travis-CI infrastructure since, as shown in Table I, it is the most widely used among the 8,000 projects we analyzed. We defined a set of 16 metrics/facts as summarized in Table II. The metric extractor has been developed in Python, leveraging the *yaml* package.

Similarly to the evolution of production and test code, we look at the size of the CI/CD configuration files by counting the total number of lines of code by filtering out comments and empty lines (M_1), and we extract the comments they contain (M_2), that can be used to quantify the number of comments and to keep track of their changes.

Then, we consider build matrices. A build matrix [14] allows for defining values for different environment settings (*e.g.,* compiler/interpreter versions) or variables, so that jobs will be instantiated by combination of those values and run in parallel. We look at three different metrics and a fact aimed at characterizing the build matrix: (i) build matrix size, (ii) jobs excluded, (iii) jobs allowed to fail, and (iv) fast finishing. Concerning the build matrix size (M_3), Travis-CI gives the possibility to define them by using the matrix expansion feature or else by listing the configuration of each job. First, we identified the set of keys that can be used as

TABLE II
DESCRIPTION OF THE METRICS BEING EXTRACTED FROM CI/CD CONFIGURATION FILES

ID	Metric/Fact	Description
M_1	LOC	Number of lines of code without comments and empty lines
M_2	Comments	Set of comments
M_3	Build matrix size	Total number of jobs that run in parallel
M_4	Jobs excluded	Number of unwanted jobs from the ones identified by the build matrix expansion
M_5	Jobs allowed to fail	Number of jobs that are allowed to fail without causing the entire build to fail
M_6	Fast finishing	Whether or not, in presence of jobs allowed to fail, the build can be marked as finished even if not all of them are completed
M_7	Phases	Number of job-defining phases among the ones predefined by Travis-CI
M_8	Env. variables	Number and name of the public (*i.e.,* environment) variables defined in the `env` key
M_9	Global env. variables	Number and name of the env. variables global to the matrix (*i.e.,* considered for each job in the matrix expansion) as defined in the `global` key
M_{10}	Notification channels	Number and types of channels explicitly used to notify the build results (*e.g.,* Slack or IRC)
M_{11}	Use Docker	Whether or not the build relies on Docker as a service, together with the set of Docker commands used (within the pipeline configuration)
M_{12}	Use shell commands	Whether or not, the build configuration (*i.e.,* `.travis.yml`) run a shell script as part of the CI/CD process, together with the set of commands
M_{13}	Use stages	Whether or not the CI/CD process relies on build stages to group jobs and parallelize their execution in each stage while running stages sequentially
M_{14}	Caching	Whether or not the CI/CD process relies on the caching feature, together with its configuration (content of the `cache` key)
M_{15}	Retrying	Set of commands that use the `travis_retry` function for retrying the command up to a specific number of time if the return code is non-zero
M_{16}	Waiting	Set of commands using `travis_wait` function to extend the time a command has to complete

matrix expansion keys that take arrays of values (*i.e.,* env, rvm, jdk, gemfile, dist, python, arch, node_js, php, compiler, os, mono, and dotnet), and multiply their size. This gives us the number of jobs being defined by simply using the matrix expansion feature. Then, we add the number of jobs listed individually by counting the number of entries into the key `jobs.include` or `matrix.include` and remove the number of excluded jobs (M_4). Indeed, since the matrix expansion feature may produce unwanted combinations (*i.e.,* resulting in unneeded jobs), it is possible to exclude some jobs by relying on the `jobs.exclude` or `matrix.exclude` key. The number of entries for the above keys together with the number of jobs excluded through environment variables results in M_4. Furthermore, Travis-CI gives the possibility to define jobs that are allowed to fail in the build matrix (*e.g.,* jobs for non-stable environments) without impacting the overall build status by using the `jobs.allow_failures` or `matrix.allow_failures` key. The number of entries for the above keys results in M_5. Finally, to speed up the build process in presence of jobs that are allowed to fail, it is possible to mark the build as finished even if not all the jobs have been completed. M_6 looks at whether the `fast_finish: true` property has been set inside the jobs/matrix section.

Each job in Travis-CI can be seen as a sequence of different phases (*e.g.,* install, script). We extract the number of phases together with their characteristics (M_7) such as the list of commands invoked in each phase. Furthermore, using the `env` key we derive M_8. To avoid duplicated code and increase the maintainability of the build process, developers may rely on environment variables that are global to the matrix. The entries in the `env.global` key with their specifications are used to determine M_9.

The CI/CD notification mechanisms let anyone know about the build results. By counting the number of entries in the `notifications` key, excluding the ones set to *false*, together with their properties we obtain M_{10}. Also, we check whether or not the build process relies on Docker, by looking at whether the `.travis.yml` includes Docker inside the `services` key or the CI/CD process relies on Docker Compose as a tool. After that, we identified the set of Docker commands by searching for commands in phases having the `- docker` prefix (M_{11}). Note that it is out of our scope to look at changes in the Docker configuration files.

To improve the readability and understandability of the CI/CD configuration files it is possible to rely on the invocation of shell scripts as part of the CI/CD process. Through regular expression matching, M_{12} identifies the presence of shell script invocations.

We also capture if a CI/CD process relies on (i) build stages (`stages` key), *i.e.,* groups of jobs executed sequentially (M_{13}), or (ii) caching strategies (`cache` key with their properties) for storing the content that does not change and reuse them in subsequent builds (M_{14}) to speed up the overall build process.

Finally, the last two facts reported in Table II focus on the usage of two specific features provided by Travis-CI, namely `travis_retry` (M_{15}) and `travis_wait` (M_{16}) that can be used to impact the way the overall build outcome is determined.

E. RQ_2 Methodology

Starting from the 6,623 projects relying on Travis-CI as CI/CD infrastructure (see Table I), for each commit in their master branch modifying the CI/CD infrastructure, we downloaded the content of the `.travis.yml` file and computed the 16 metrics/facts previously introduced. For projects with multiple Travis-CI configurations, we analyzed all of them. After that, we computed the differences in terms of metrics/facts among two subsequent changes as described in the following:

- for each numerical metric (*e.g.,* LOC) we compute whether the value increases, decreases, or remains unchanged;
- for each fact (*e.g.,* comments or phases' content) we determine whether or not its value (*e.g.,* set content) changes;
- for each boolean metric (*e.g.,* Caching, Use shell commands) we compute whether the feature is being introduced or removed in a commit.

To address RQ_2, we report:

- the percentage of commits over the projects' CI history that affected Travis-CI configurations and those which increased/decreased its size;
- a Spearman's correlation [44] analysis between the number of changes to Travis-CI and the CI history length;

- results of Dunn's test [25] to determine whether the frequency of CI configuration file changes significantly varies between programming languages (p-values are adjusted using Benjamini-Hochberg correction [51]). We also compute Cliff's d effect size [30].
- the percentage of projects where each metric changed at least once, at least 25% or 50% of the commits modifying the Travis-CI configuration, where it increased/was added, decreased/was removed, or was changed.
- considering the 615 commits manually-classified in RQ_1, for each of the 34 restructuring actions, whether the 16 metrics increase or decrease.

III. Study Results

This section discusses the study results addressing the two research questions formulated in Section II.

A. What are the restructuring operations occurring in CI/CD pipelines?

As a result of the qualitative analysis, we identified 34 CI/CD pipeline restructuring actions. As shown in Table III, such actions have been organized in a three-level taxonomy. The first level of the taxonomy distinguishes between:

1) Extra-Functional (EF): actions that (in principle) should not modify the pipeline behavior (or should have a limited effect on it), and introduce an extra-functional improvement to the pipeline, *e.g.,* maintainability, security, or performance;
2) Pipeline-Behavior (PB): actions modifying the pipeline's behavior, that aim at restructuring the pipeline to cope with the system's or technology evolution, or to improve the pipeline's effectiveness.

The second level of the categorization groups restructuring actions based on their purpose. Table III also reports, for each type of action, the number of instances found in our manually-analyzed sample. In the following, we discuss the different categories of the taxonomy.

Maintainability (EF). This category groups restructuring actions aimed at making the pipeline easier to understand and maintain. The most common action, *Improve readability of code snippets*, concerns redesigning a `yaml` source code snippet (without altering its syntax) to make it more compact, or to ease its readability. For example, in *zeromq/zyre* a commit explicitly admits the removal of duplicated commands in the `.travis.yml` file: "Problem: .travis.yml contains a lot [of] duplication. Solution: Cleanup the travis file" [15], or in *yiisoft/yii2* we found a commit where the developer removes a command reported in each stage of the pipeline as already included in the `after_script` phase (*i.e.,* "cleaned cleanup (done in after_script)") [13]. From a different perspective, in *phpbb/phpbb* we encountered a commit in which the goal was to restructure the conditional commands being involved in the `before_script` phase to group the cases with the same objective (*i.e.,* "[task/travis] Refactor php version check for dbunit install") [11]. Related to readability, often developers improve the level of the `yaml` script documentation (*i.e., Improve code comments*).

Another relevant action is related to *Simplify build matrix*. The build matrix allows defining treatments for different dimensions under which a build must be run. These may include for example compiler/interpreter versions, virtual machines, or simply environment variable values. The *Use matrix expansion feature or list job configurations explicitly* groups change in the strategy used for specifying the pipeline environments, *i.e.,* listing jobs explicitly or using a matrix expansion.

Then, there are categories that are analogous to Fowler's refactoring [28], although they are performed on build-related assets. Specifically, we found: *Split build scripts and/or build jobs*, *Extract environment variables*, *Move variables to a different scope*, and *Rename steps/tasks/scripts/jobs*.

Other categories are related to changing the way a task is accomplished, *e.g.,* using a shell script or a dedicated plug-in. These include *Use shell scripts to avoid duplicated code*, or, on the contrary, *Remove shell invocations* and *Replace tools/methods for accomplishing a specific task*. Concerning the first one, we found a commit in *karelzak/util-linux* [5] where the developer explains the reason why it is important relying on specific shell scripts instead of simply relying on the Travis-CI features: "travis-ci: refactor and add .travis-functions.sh. Travis yaml syntax, where we can only use shell one-liners, is awful and ugly. We add a real shell script and source it from .travis.yml". While in principle this change should not affect the behavior, there is no guarantee that a plugin and the shell script behave exactly the same.

Performance (EF). These restructuring actions aim at reducing the build time (a slow build is one of the antipatterns advocated by Duvall [27] and monitored by the tool by Vassallo *et al.* [49]). In most cases (in our manual analysis sample) the build speedup is achieved by removing unneeded components from the build, either *unneeded environments, scripts, or tasks*, or through a *Cleanup build matrix*, *i.e.,* by removing unneeded entries from a build matrix (*e.g.,* a compiler/interpreter no longer used).

It is also possible to improve performance by adopting CI caching (*i.e., Change caching configuration*), *e.g.,* to perform incremental builds without refreshing and even recompiling some dependencies every time.

Some other performance-related actions have been advocated by existing literature, in particular by Duvall *et al.* [26]. These include *Introduce parallelization* by running more than one job in parallel or else by introducing staged builds. For example, in *gatsbyjs/gatsby* we found a commit reporting: "... ci: attempt to speed-up jobs" [3], or in *chef/knife-ec2* a commit reporting that the parallelization relies on threads (*i.e.,* "Speed up travis installs by running 7 threads") [2].

Security (EF). While identifying security-related concerns was not a specific goal of our work, and has been extensively treated in related literature [41] (concerning Infrastructure-as-Code and not specifically CI/CD pipelines), we found two types of restructuring that relate to security. One, pointed out by Rahman *et al.* [41], but also detected by the pipeline

TABLE III
CI/CD PIPELINE RESTRUCTURING ACTIONS

First-Level Category	Second-Level Category	Restructuring Action	# of Instances
Extra Functional	Maintainability	A_{01}: Improve readability of code snippets	24
		A_{02}: Extract environment variables	16
		A_{03}: Simplify build matrix (without removing jobs)	16
		A_{04}: Use shell scripts to avoid duplicated code	13
		A_{05}: Split build scripts and/or build jobs	11
		A_{06}: Use matrix expansion feature or list job configurations explicitly	11
		A_{07}: Improve code comments	9
		A_{08}: Remove shell invocations	7
		A_{09}: Move variables to a different scope	4
		A_{10}: Rename steps/tasks/scripts/jobs	4
		A_{11}: Replace tools/methods for accomplishing a specific task	4
	Performance	A_{12}: Remove unneeded environments/scripts/tasks	59
		A_{13}: Cleanup build matrix	41
		A_{14}: Change caching configuration	20
		A_{15}: Introduce parallelization	9
	Security	A_{16}: Introduce/Remove `sudo` in commands	3
		A_{17}: Remove credentials/tokens in clear	2
Changing the pipeline's behavior	Infrastructure	A_{18}: Introduce Dockerization/Containerization	3
	Build Policy	A_{19}: Change how the build outcome is determined	10
		A_{20}: Skip useless tasks/steps/environments	9
		A_{21}: Change build matrix introducing `allow_failure`	8
		A_{22}: Change the dependencies' installation policy	6
		A_{23}: Introduce/Remove nightly builds	3
		A_{24}: Deploy only after build success	1
		A_{25}: Move from manual to automatic tasks	1
	Dashboard/notifications	A_{26}: Improve readability of the build log	19
		A_{27}: Restructure the notification mechanism	13
	Build Process Organization	A_{28}: Update checks in the build process	43
		A_{29}: Reorganize build steps order of execution	27
		A_{30}: Restructure `install` and `script` phases	13
		A_{31}: Restructure jobs and/or stages	7
		A_{32}: Introduce/Change timeout/waiting time for tasks	2
		A_{33}: Introduce/Remove retry for commands	2
		A_{34}: Use parametrized builds	1

linter of Gallaba and McIntosh [29], is related to *Removing credentials/tokens in clear* from the pipeline configuration. The latter is highly important for open source projects since the pipeline configuration is accessible to anybody. Moreover, a different restructuring action is *Introduce/Remove `sudo` in commands*. For example, in *getsentry/sentry-ruby*, we found a commit which message mentions "... Don't tell people to sudo bundle install. That's Bad" [4].

Next, we discuss pipeline restructuring categories that aim at changing the pipeline behavior (PB), *i.e.,* those related to Infrastructure, Build Policy, Dashboard/Notifications, and Build Process Organization.

Infrastructure (PB). We found only three cases in which the infrastructure set to run the overall build process changed by adopting Docker to have a consistent environment, increase the repeatability and reduce the overall build time (*i.e., Introduce Dockerization/Containerization*).

Build Policy (PB). First, this category includes a set of restructuring actions dealing with the build triggering strategy, *e.g.,* the way the build process is enacted, namely *Introduce/Remove nightly builds*, *Deploy only after build success*, *Move from manual to automatic tasks* and *Change dependencies' installation policy*. The former entails both the introduction and removal as a possible improvement activity, since, as already reported by Zampetti *et al.* [53], depending on the case a nightly build can be better (for time-consuming tasks) or worse (it defeats the principle of continuous builds). Concerning the dependencies installation policy, developers often change it to reduce the overall build execution time. For example, in *mruby/mruby* we found a commit where the developer admits the reduced build time as a consequence of the change in the way dependencies are installed, *i.e.,* "Disable automatic update and clean up on brew install (install time 160 sec -> 5 sec)" [7].

This category also includes restructuring actions involving the strategy used for assigning the overall build status: *Skip useless tasks/steps/environments*, *Change build matrix introducing `allow_failure`*, and *Change how the build outcome is determined*. Concerning *Skip useless tasks/steps/environments*, even if previous literature highlights that skipping a task/stage/job to hide a failure instead of fixing it is a bad practice [49], [53], we found cases where skipping represents a safe operation. For example, in *navit-gps/navit* a commit motivates the circumstances under which the skip is mandatory: "update:CI: skip the build steps if the change is only for documentation" [8].

Some CI/CD infrastructures (*e.g.,* Travis-CI) permit to specify jobs that are allowed to fail without affecting the overall build status (*Change build matrix introducing* `allow_failure` action). While, in principle, this is useful to cope with unstable environments, once the environment reaches maturity, it is important to make the job affecting the build status [50]. The *Change how the build outcome is determined* includes cases where developers move commands from the `script` to the `after_script` phase in Travis-CI [6], or else cases where developers rely on the `fast_finish` feature provided by the CI/CD framework for reporting success/failure without waiting for jobs that are allowed to fail [1].

Dashboard/Notifications (PB). This category includes restructuring actions aimed at improving (i) the notification mechanism used to let developers know the build result, *i.e., Restructure the notification mechanism*, and (ii) the build outcome shown in the build log, *i.e., Improve readability of the build log*. For *Restructure the notification mechanism*, for example, a commit message mentions the need to replace the notification mechanism to fulfill a project policy "Remove Travis CI notification hook for IRC ...* Add Travis CI notification hook for gitter.im" [12]. Concerning *Improve readability of the build log*, we found a commit in *openmm/openmm* where a developer clearly states: "Eliminate extra warning about overriding the warning level [...] Remove annoying -V" [9]. While in the previous case better readability is achieved by pruning useless information, in other cases more details need to be added. For example, in *phalcon/incubator* a commit reports the need to "Be verbose on tests" [10].

Build Process Organization (PB). This category accounts for restructuring activities dealing with the overall CI/CD process organization in terms of commands involved in each phase, stage, or job, and their order of execution: *Restructure* `install` *and* `script` *phases*, *Reorganize build steps order of execution*, *Restructure jobs and/or stages*, *Use parameterized builds*, and *Update checks in the build process*. For the latter, we found cases in which the execution of specific phases is needed only when the build is triggered by a pull request or a change is done on a specific branch.

Introduce/Change timeout/waiting time for tasks includes changes aimed to avoid build termination because of a timeout, while *Introduce/Remove retry for commands* deals with running or not the same command multiple times before determining the overall build status. The latter may be useful to cope with flaky behavior (which is a known practice in CI/CD [42]), although its abuse is a bad practice [50].

RQ$_1$ summary: We devised a taxonomy of 34 CI/CD pipeline restructuring actions, coping with extra-functional properties, or with the adaptation of the pipeline to the system/environment evolution.

B. How do CI/CD pipelines evolve over time?

Fig. 1 shows boxplots of the percentage of commits, over the project CI history, where the `.travis.yml` file(s) was/were changed, as well as the cases in which LOC increased or decreased (M_1 in Table II). The numbers in parenthesis indicate the first, second (median), and third quartile, respectively. As it can be seen, the median percentage of changes is relatively low (2.22% of the commits), however, in absolute terms, the median number of pipeline-affecting commits is 20, even if 618 projects ($\simeq 10\%$ of the total) have more than 100 commits. The median percentage of changes increasing or decreasing LOC is below 1% of the total. Indeed, the pipeline configuration should not be modified for each change occurring in the project. Rather, this should happen only when, as found by Vassallo *et al.* [50], developers realize that the pipeline uses a bad practice or, in general, there is a need for cleanup, or when a radical change in the project/technology makes it necessary.

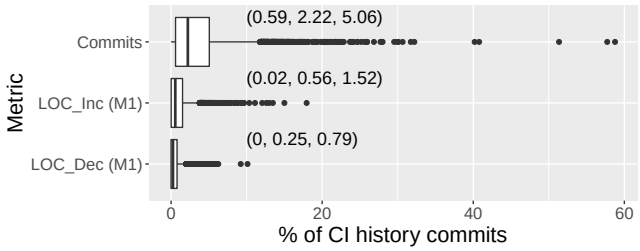

Fig. 1. % of commits, LOC increase and decrease impacting Travis-CI config.

TABLE IV
% OF PROJECTS UNDERGOING CHANGES TO DIFFERENT METRICS

Metric	% Proj Pr.	≥25% .yml changes	≥50% .yml changes	% Proj Incr.	% Proj Decr.	% Changes
Comments (M_2)	61.95	32.28	11.65	58.10	51.08	61.95
Jobs (M_3)	82.90	35.51	6.48	78.53	71.25	–
Excl. Jobs (M_4)	11.52	0.84	0.11	11.24	10.12	–
Allow Fail (M_5)	27.61	1.98	0.22	26.59	23.79	–
Fast Finish (M_6)	15.96	–	–	15.48	7.80	–
Phases (M_7)	91.97	75.43	42.87	75.28	55.34	91.97
Glob. Var. (M_8)	30.86	6.57	1.72	29.52	19.53	30.86
Env. Var. (M_9)	32.00	9.28	3.53	29.18	27.02	32.00
Notif. Ch. (M_{10})	22.65	1.98	0.65	20.63	0.00	22.65
Docker (M_{11})	10.59	–	–	9.41	3.66	4.69
Shell (M_{12})	27.84	0.28	0.04	27.84	16.97	24.85
Stages (M_{13})	7.86	0.15	0.04	7.80	2.26	–
Cache (M_{14})	41.24	1.46	0.22	38.20	15.93	16.04
Retry (M_{15})	10.68	0.34	0.06	10.16	7.11	–
Wait (M_{16})	5.23	0.11	0.00	5.19	3.66	–

We also investigated whether projects with a longer history change their pipeline more than others. We found a positive yet moderate Spearman's ($\rho = 0.22$), and Kendall's ($\tau = 0.17$) correlation, implying that the number of changes to the pipeline configuration only partially depends on the history length.

We also looked at whether CI/CD pipeline configurations belonging to certain programming languages are more change-prone than others. We found that Ruby and Python projects are significantly more change-prone than others (Dunn's test p-value< 0.01, effect size medium in most cases, further data in the appendix [52]). Python and Ruby exhibit subtle incompatibilities across versions [43], and we found that pipelines frequently adapt the build matrix adding/replacing interpreter versions. This affects metric M_3 in Table II.

Table IV provides an overview of the extent to which the different indicators described in Table II (except for LOC, discussed above) change across the analyzed 4,644 projects.

Table IV reports (i) the % of projects where each metric changes at least once, (ii) changes at least 25% of the `.travis.yml` commits, (iii) changes at least 50% of the `.travis.yml` commits (iv) has at least an increase/introduction change, (v) a decrease/removal change, or (vi) a modification change (no increase nor decrease).

As it can be noticed from the table, Phases (M_7, 91.97%) and Jobs (M_3, 82.90%) are those changed more. Even when considering 25% or 50% of the changes, the percentage of projects affected by such changes remains very high. This is not surprising, because, as conjectured in the introduction, one emerging reason for pipeline change is to adapt to changing environments, *e.g.,* compilers, interpreters. Other changes affecting these variables are related to restructuring too complex scripts into job matrices, or, in general, to simplifying matrices.

Other changes occurring quite often are related to documentation by relying on comments being added to configuration scripts (M_2, 61.95%), which, as we see, both increase and decrease. In our qualitative analysis, we noticed how comment removals were either related to clean-ups, or to uncommenting existing code. On the contrary, we found cases for which comment additions were related to excluding code or else to adding the reasons behind a specific configuration setting.

As for Cache (M_{14}, 41.24% of the projects), we have observed more introductions than removals, yet there were cases in which developers found that it was no longer needed to have an incremental build. Moreover, once introduced, caching specification may evolve as a consequence of the evolutionary history of the project ($\simeq 16\%$).

About 30% of the projects exhibit restructuring related to defining global environment variables (M_8) or extracting environment variables (M_9). We observed a similar proportion of cases in which variables were added or removed.

The abuse of shell scripts may negatively impact the maintainability of CI configurations [53]. However, we found this to be controversial. In our study the percentage of projects where the use of shell scripts (M_{12}) increases (27.85%) is larger than when it decreases (16.97%). This because, as explained by the examples discussed in Section III-A, an external script may help to simplify the build.

Unsurprisingly, notification mechanisms/channels (M_{10}, 22.65%) are also customized quite a bit (*e.g.,* because the team changes), although we found no removals of notification.

Other observed changes partially relate to antipatterns handled by the tool by Vassallo *et al.* [50], *i.e.,* allowing a build to fail (M_5, 27.61%) for which, in practice, we observed both additions and removals, and, with a lower percentage (10.68%), retry jobs/tasks (M_{15}).

Last, but not least, we observed a small, yet non-negligible percentage (10.59%) of projects where changes in the configuration concerned the introduction (in most cases) or removal of Docker (M_{11}), *e.g.,* to cope with complex execution environments. However, this may generate both maintenance [23] or security [40], [41] issues in Docker configuration files.

Fig. 2 shows the extent to which metrics change in the 615 manually analyzed commits of RQ_1, that involve `.travis.yml` files and are performed in the project's master branch (including the merger of other branches). Two action types (A_{24} and A_{25}) only occur elsewhere and are not reported in this figure. Specifically, we report, for each refactoring action, the proportion of commits (1=100%) where the metric changes (a darker color corresponds to a higher proportion). Near each action name, we report the number of considered commits (the number may be smaller than in Table III, as the latter includes commits in all branches as well as other CI/CD infrastructures beyond Travis-CI). Cases where the number is small must be interpreted with caution.

10 commits in our RQ_1 dataset did not induce any change to metrics. These were related to some fine-grained changes (*e.g.,* change build matrix order of execution, minor changes to notification mechanisms, or changes to conditionals) not captured by the current set of metrics.

By looking at Fig. 2, we can state that the 16 metrics detailed in Table II very often reflect the type of restructuring action being applied, even if, for many of them it is possible to see a change in more than one metric, because restructuring actions may occur together with other changes to the configuration files, *e.g.,* the addition of a new environment or the addition of commands in previously defined phases. For example, M_{11} changes only when a project introduces or removes Docker as a service (A_{18}), or when it changes the way of using Docker by modifying the commands being used. Each time a developer moves a variable to a different scope (A_{09}), *e.g.,* introduce a global variable from an environment variable, both M_8 and M_9 change. When moving/extracting commands from the configuration file to a shell script (A_{04}) or splitting jobs (A_{05}) we observe a change to M_7 (100% and $\simeq 80\%$ respectively). The second percentage is less than 100% as we currently miss cases where script fragments were moved into other, existing script files. Other missed cases here, as in *Remove shell invocations*, may depend on the presence of shell scripts having extensions we did not capture.

Concerning the adoption or change of the CI caching feature (A_{14}), 8 out of 10 commits show a change in M_{14}, for the other two cases we found a limitation in our extractor since we only focus on the `cache` directive without considering the `before_cache` key.

Even if our current set of metrics does not include the analysis of conditionals yet, in $\simeq$ 90% of the cases in which checks were updated (A_{28}) there is also a change in the specification of the phases being included in the CI/CD configuration (M_7). Focusing on the replacement of tools (A_{11}), it seems that developers tend to apply the replacement in isolation since that M_1 does not change, and the same occurs for the other 14 metrics.

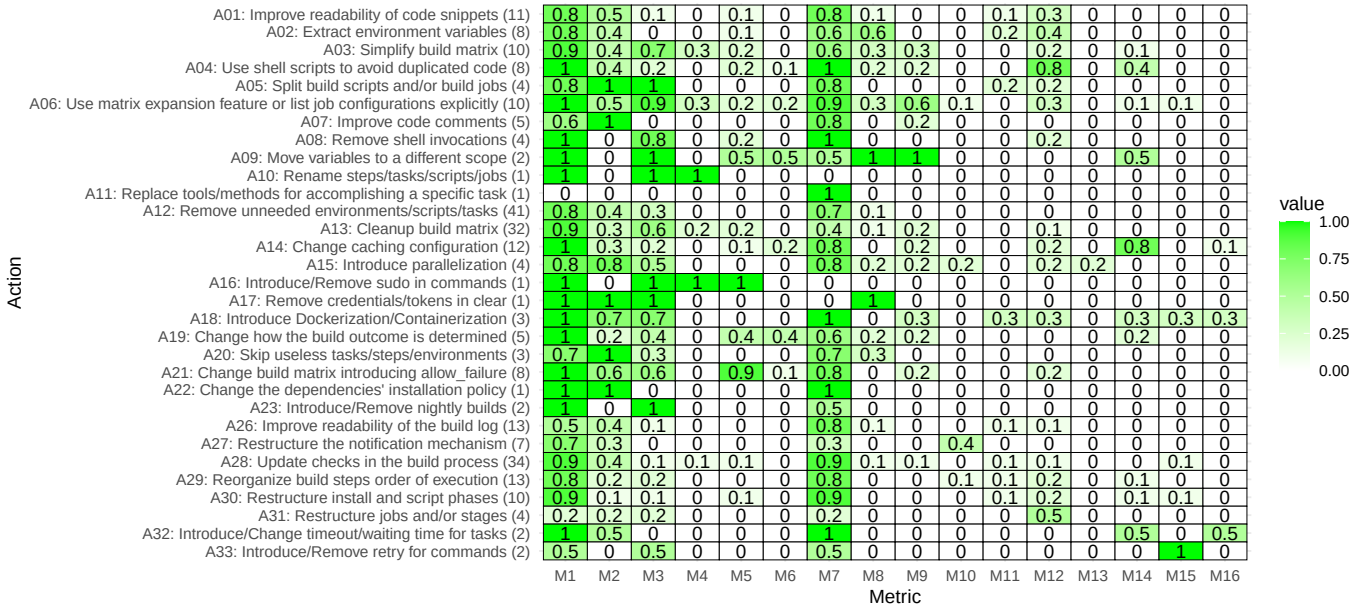

Fig. 2. Percentage of metric changes for commits containing CI/CD restructuring actions.

RQ_2 summary: While the percentage of changes to Travis-CI files is relatively low, some projects may undergo several changes to the pipeline, in most cases concerning build/phase restructuring. Within their limitations, our 16 metrics tend to reflect the performed restructuring actions. Therefore, they can be useful features to suggest pipeline restructuring actions as well as smell removals.

IV. Threats to Validity

Construct validity (relationship between theory and observation). The choice of candidate commits in RQ_1 may be biased by the presence of certain keywords in commit messages. We mitigate this threat by performing the RQ_2 analysis on all commits involving CI configurations. The manual analysis of RQ_1 might be insufficient to understand the developers' intent in restructuring CI configurations. We mitigated this threat by observing all available sources (diffs, commit messages, linked issues, and pull requests). For RQ_2, one threat could be related to the extent to which the 16 metrics of Table II reflect restructuring changes to CI/CD pipelines. While we are aware that those metrics are far from being complete, we derived them from the qualitative observations of RQ_1.

Internal validity (possible factors that could influence the observed variables). We analyzed the extent to which factors such as history length and programming language influence CI configuration change-proneness.

Conclusion validity (relationship between treatment and outcome). Such threats are due to the reliability of our measurements. We have limited subjectiveness and error-proneness in RQ_1 by having multiple raters cooperatively assessing the sample of commits. Given the complexity of the task (with many categories that need to be incrementally restructured), we preferred a cooperative card sorting [45] over independent coding and inter-rater agreement computation. As for RQ_2, we limited problems due to bugs in our metrics extractor by letting an author (who did not implement the tool) manually checking the metrics extracted for a sample of 30 projects. Nevertheless, as discussed in Section II-D and Section III-B, our metric analysis is light-weight, therefore approximations and imprecision occurred anyway.

External validity (generalization). As for the qualitative analysis, we used the third step of manual tagging to verify that saturation was reached. Besides that, we are aware that our results are valid for open source projects, although the sample of projects is relatively large, and although they are related to 8 programming languages and, for RQ_1, for 7 CI/CD frameworks. As for RQ_2, the metric extractor only works for Travis-CI, the most popular in the open source [33]. Other CI/CD infrastructures will be supported in our future work.

V. Related Work

This section discusses the literature related to CI/CD practices and to the evolution of software builds.

A. CI/CD barriers and bad practice

Different authors studied barriers/challenges in adopting CI/CD. These were initially identified by Duvall *et al.* [26], and related to the need for maintaining a fully automated build process, handling dependencies, having different levels of builds, and coping with different target environments.

Hilton *et al.* [31] studied barriers developers encounter when moving toward CI. These are related to dimensions such as quality assurance, security, and flexibility. Challenges in the migration towards CD were then studied by Olsson *et al.* [39].

Once CI/CD is in place, it may be applied improperly, making, for example, the pipeline less effective, slow, or more difficult to maintain. To this extent, Duvall defined a comprehensive set of 50 patterns and antipatterns regarding

different phases of the CI/CD process [27]. Zampetti *et al.* [53] conducted an empirical investigation involving interviews with developers, analysis of Stack Overflow posts, and online surveys to identify CI smells that developers actually encounter in the practice. They defined a catalog of 79 bad smells belonging to 7 different categories.

Differently from Zampetti *et al.* [53], we do not investigate restructuring actions by interviewing developers or by looking at Stack Overflow. Instead, we analyze how CI/CD configurations change over time. For this reason, our work is complementary to what done by Zampetti *et al.*.

B. CI/CD smell detectors

Gallaba *et al.* [29] proposed an approach to detect and remove antipatterns in Travis-CI configuration scripts. Such antipatterns are specifically related to problems such as (i) redirecting scripts into interpreters, (ii) bypassing security checks, (iii) having unused properties in `travis.yml` files, or (iv) unrelated commands in build phases.

Deviations from good CI principles have also been investigated in a work by Vassallo *et al.* [49], however, by observing the pipeline in its execution (*i.e.,* through its log) rather than analyzing its configuration scripts. Their tool, CI-Odor, detects antipatterns such as a build becoming slow, developers working on feature branches for a longer period, broken release branches, or skipped tests to make the build passing.

Vassallo *et al.* [50] proposed a linter for GitLab configurations, and conducted a six-month study on over 5k projects hosted on GitLab. They monitored CI/CD bad practices and automatically opened issues when such bad practices occurred. They reported how a majority of the opened issues was fixed.

All the aforementioned work deals with the identification and resolution of CI/CD bad practices. This is one of the reasons (but not the only one, as we observed in our study) for which a CI/CD pipeline is being restructured. Once pipeline restructuring actions have been identified (RQ_1), the facts extracted and observed in RQ_2 can be used to learn from previous changes or from changes in other projects, and recommend CI/CD smell removals.

Abdalkareem *et al.* [17] optimize the build process by CI-skipping commits where the build outcome is clear. Dynamically-skipping commits is one of the possible CI pipeline optimizations. In other cases, as the ones studied in this paper, a CI configuration restructuring would be necessary.

The work by Rahman *et al.* [41] is instead less related to CI/CD configurations, while being related to security smells in Infrastructure-as-Code scripts, *e.g.,* in Docker image configurations. As shown in our study, these may also affect the evolution of a pipeline when Docker images are used within it.

C. Evolution and quality of builds

In previous work, researchers studied the evolution of builds, which, unavoidably, interact with the evolution of CI/CD configurations.

In an early study, McIntosh *et al.* [37] studied the effort in build maintenance. They found that such an effort is similar to the maintenance effort of production or test code and that nearly 80% of the software developers are involved in such changes. In a follow-up work [35], the need for maintaining build files as a consequence of source code changes was also predicted through random forest classifiers, and by using both programming language-specific and language-agnostic features.

McIntosh *et al.* [38] studied the maintenance of build scripts in programs using old (*e.g.,* ant-based) and new, framework-based (*e.g.,* Maven) build technologies. The study suggests how the latter requires more maintenance, also tightly coupled to the evolution of the source code structure. Sometimes developers also migrate between different build systems, as it has been studied by Suvorov *et al.* [47] in the context of KDE and the Linux kernel. They found how the build migration follows phases similar to the spiral model life-cycle.

In the build process of C/C++ programs, frequently changing header files can slow down the build. For this reason, McIntosh *et al.* [36] propose the analysis of header file dependency graphs to identify hotspots that can cause performance degradation.

Bezemer *et al.* study unspecified dependencies in Makefile-based builds [20], in which the analysis of explicit dependencies is relevant to entail incremental builds which, as found in our study, may also be a key component of a CI/CD process.

The aforementioned research has highlighted how build systems' evolution plays a paramount role in software evolution. In our paper, we shift the focus on the evolution of CI/CD configuration scripts, to understand what features of such scripts are typically subject to changes during restructuring operations.

VI. Conclusion

As production code, and as build automation scripts [37], [35], [38], Continuous Integration (CI) and Delivery (CD) pipelines evolve to cope with the system's evolution and to fix antipatterns/smells [27], [29], [49], [50], [53].

This paper reports an empirical study on the evolution of CI/CD configuration files. With an open coding of 615 commits from open source projects written in 8 programming languages and relying on 7 CI/CD infrastructures, we devised a taxonomy of 34 pipeline restructuring actions grouped into two-top level categories (extra-functional changes, and changes affecting the pipeline's behavior) and 7 sub-categories.

The taxonomy supported us to identify 16 metrics and implement a metric extractor for Travis-CI. The analysis of pipelines' evolution for 4,644 projects indicates that, although pipeline's changes do not occur as frequently as for production/test code, they can happen several times. The study also shows how the metrics can be used to monitor CI/CD pipelines' evolution.

In future work, we plan to improve the analyzer to cover further metrics and fine-grained changes, and to automatically detect restructuring actions. Also, we plan to leverage it to support automated recommendations for pipeline improvements.

Acknowledgments

Fiorella Zampetti and Massimiliano Di Penta are supported by the EU Horizon 2020 project *COSMOS* (DevOps for Complex Cyber-physical Systems), Project No. 957254-COSMOS.

References

[1] "ankidroid/Anki-Android," Travis-CI configuration change: https://github.com/ankidroid/Anki-Android/commit/7ab84731, accessed: 2021-04-28.

[2] "chef/knife-ec2," Travis-CI configuration change: https://github.com/chef/knife-ec2/commit/9f8b76c7, accessed: 2021-04-28.

[3] "gatsbyjs/gatsby," Travis-CI configuration change: https://github.com/gatsbyjs/gatsby/commit/4226693f, accessed: 2021-04-28.

[4] "getsentry/sentry-ruby," Travis-CI configuration change: https://github.com/getsentry/sentry-ruby/commit/29290b5e, accessed: 2021-04-28.

[5] "karelzak/util-linux," Travis-CI configuration change: https://github.com/karelzak/util-linux/commit/dd68764c, accessed: 2021-04-28.

[6] "livewire/livewire," Travis-CI configuration change: https://github.com/livewire/livewire/commit/6dd064c7, accessed: 2021-04-28.

[7] "mruby/mruby," Travis-CI configuration change: https://github.com/mruby/mruby/commit/92d8012b, accessed: 2021-04-28.

[8] "navit-gps/navit," Travis-CI configuration change: https://github.com/navit-gps/navit/commit/39f745bc, accessed: 2021-04-28.

[9] "openmm/openmm," Travis-CI configuration change: https://github.com/openmm/openmm/commit/23143e82, accessed: 2021-04-28.

[10] "phalcon/incubator," Travis-CI configuration change: https://github.com/phalcon/incubator/commit/03aaa4df, accessed: 2021-04-28.

[11] "phpbb/phpbb," Travis-CI configuration change: https://github.com/phpbb/phpbb/commit/841d11c6, accessed: 2021-04-28.

[12] "sebastianbergmann/phpunit-mock-objects," Travis-CI configuration change: https://github.com/sebastianbergmann/phpunit-mock-objects/commit/35cb1273, accessed: 2021-04-28.

[13] "yiisoft/yii2," Travis-CI configuration change: https://github.com/yiisoft/yii2/commit/4e29c44c, accessed: 2021-04-28.

[14] "zeromq/zyre," Travis-CI Build Matrix: https://docs.travis-ci.com/user/build-matrix/, accessed: 2021-06-21.

[15] "zeromq/zyre," Travis-CI configuration change: https://github.com/zeromq/zyre/commit/c250599d, accessed: 2021-04-28.

[16] "What is continuous delivery?" 2017. [Online]. Available: https://aws.amazon.com/devops/continuous-delivery/

[17] R. Abdalkareem, S. Mujahid, E. Shihab, and J. Rilling, "Which commits can be CI skipped?" *IEEE Trans. Software Eng.*, vol. 47, no. 3, pp. 448–463, 2021.

[18] K. Beck, *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2000.

[19] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub," in *International Conference on Mining Software Repositories*, 2017.

[20] C. Bezemer, S. McIntosh, B. Adams, D. M. Germán, and A. E. Hassan, "An empirical study of unspecified dependencies in make-based build systems," *Empir. Softw. Eng.*, vol. 22, no. 6, pp. 3117–3148, 2017.

[21] G. Booch, *Object Oriented Design: With Applications*. Benjamin Cummings, 1991.

[22] L. Chen, "Continuous delivery: Overcoming adoption challenges," *Journal of Systems and Software*, vol. 128, pp. 72 – 86, 2017.

[23] J. Cito, G. Schermann, J. E. Wittern, P. Leitner, S. Zumberi, and H. C. Gall, "An empirical analysis of the docker container ecosystem on github," in *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, 2017, pp. 323–333.

[24] S. Dueñas, V. Cosentino, G. Robles, and J. M. Gonzalez-Barahona, "Perceval: software project data at your will," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018, pp. 1–4.

[25] O. J. Dunn, "Multiple comparisons among means," *Journal of the American Statistical Association*, vol. 56, no. 293, pp. 52–64, 1961.

[26] P. Duvall, S. M. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007.

[27] P. M. Duvall, "Continuous delivery: Patterns and antipatterns in the software life cycle," *DZone refcard #145*, 2011. [Online]. Available: https://dzone.com/refcardz/continuous-delivery-patterns

[28] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[29] K. Gallaba and S. McIntosh, "Use and misuse of continuous integration features: An empirical study of projects that (mis) use travis ci," *IEEE Transactions on Software Engineering*, vol. 46, no. 1, pp. 33–50, 2018.

[30] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach*, 2nd ed. Lawrence Earlbaum Associates, 2005.

[31] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "Trade-offs in continuous integration: Assurance, security, and flexibility," in *Proceedings of the 25th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2017*, 2017.

[32] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. 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.

[33] ——, "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.

[34] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[35] S. McIntosh, B. Adams, M. Nagappan, and A. E. Hassan, "Mining co-change information to understand when build changes are necessary," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, 2014, pp. 241–250.

[36] ——, "Identifying and understanding header file hotspots in C/C++ build processes," *Autom. Softw. Eng.*, vol. 23, no. 4, pp. 619–647, 2016.

[37] S. McIntosh, B. Adams, T. H. D. Nguyen, Y. Kamei, and A. E. Hassan, "An empirical study of build maintenance effort," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, 2011, pp. 141–150.

[38] S. McIntosh, M. Nagappan, B. Adams, A. Mockus, and A. E. Hassan, "A large-scale empirical study of the relationship between build technology and build maintenance," *Empir. Softw. Eng.*, vol. 20, no. 6, pp. 1587–1633, 2015.

[39] H. H. Olsson, H. Alahyari, and J. Bosch, "Climbing the "stairway to heaven" – a mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software," in *Proceedings of the 2012 38th Euromicro Conference on Software Engineering and Advanced Applications*, ser. SEAA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 392–399.

[40] A. Rahman, "Characteristics of defective infrastructure as code scripts in devops," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 476–479.

[41] A. Rahman, C. Parnin, and L. Williams, "The seven sins: Security smells in infrastructure as code scripts," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, pp. 164–175.

[42] T. Rausch, W. Hummer, P. Leitner, and S. Schulte, "An empirical analysis of build failures in the continuous integration workflows of java-based open-source software," in *International Conference on Mining Software Repositories (MSR)*. ACM, 2017.

[43] P. Sharma, B. T. R. Savarimuthu, N. Stanger, S. A. Licorish, and A. Rainer, "Investigating developers' email discussions during decision-making in python language evolution," in *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE'17. New York, NY, USA: Association for Computing Machinery, 2017, p. 286–291.

[44] C. Spearman, "The proof and measurement of association between two things," *American Journal of Psychology*, vol. 15, pp. 88–103, 1904.

[45] D. Spencer, *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[46] D. Staahl and J. Bosch, "Modeling continuous integration practice differences in industry software development," *J. Syst. Softw.*, vol. 87, pp. 48–59, Jan. 2014.

[47] R. Suvorov, M. Nagappan, A. E. Hassan, Y. Zou, and B. Adams, "An empirical study of build system migrations in practice: Case studies on KDE and the linux kernel," in *28th IEEE International Conference on Software Maintenance, ICSM 2012, Trento, Italy, September 23-28, 2012*, 2012, pp. 160–169.

[48] B. Vasilescu, Y. Yu, H. Wang, P. T. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in github," in *ESEC/SIGSOFT FSE*. ACM, 2015, pp. 805–816.

[49] C. Vassallo, S. Proksch, H. Gall, and M. Di Penta, "Automated reporting of anti-patterns and decay in continuous integration," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, Canada, May 25 - 31, 2019*. IEEE, 2019.

[50] C. Vassallo, S. Proksch, A. Jancso, H. C. Gall, and M. 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*, 2020, pp. 327–337.

[51] B. Yoav and H. Yosef, "Controlling the false discovery rate: A practical and powerful approach to multiple testing," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 57, no. 1, pp. 289–300, 1995.

[52] F. Zampetti, S. Geremia, G. Bavota, and M. Di Penta, "CI/CD Pipelines Evolution and Restructuring: A Qualitative and Quantitative Study - Dataset and scripts," Apr. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4727934

[53] F. Zampetti, C. Vassallo, S. Panichella, G. Canfora, H. C. Gall, and M. Di Penta, "An empirical characterization of bad practices in continuous integration," *Empirical Software Engineering*, vol. 25, no. 2, pp. 1095–1135, 2020.