Universiteit Antwerpen

Empirical Analysis on Continuous Integration Anti-patterns in JavaScript Repositories

Freek De Sagher

Principal Advisor: Serge Demeyer
Assistant Advisor: John Businge

August 25, 2022

AnSyMo
Antwerp Systems & Software Modelling
University of Antwerp

Contents

Acknowledgments

This thesis would not have been possible without the help and advise of my advisors Dr. John Businge and professor Serge Demeyer. With their help and advice I have been able to put this work to a good end. For that, I am extremely grateful.

Additionally, I would like to thank Dr. Alexandre Decan and professor Tom Mens of the University of Mons for providing valuable feedback that helped to give form to the performed research and for proposing the topic of the research.

I am also grateful to my classmates, friends and family for supporting me throughout the journey, especially for encouraging me when things were going less than planned.

Abstract

Continuous Integration, referred to as CI, is implemented in a large share of git repositories in order to help automating the building and testing cycle. CI only yields great benefits if it is implemented right. Due to the high availability of different tools and the high flexibility of implementing a certain goal, it is harder to make the right choices in practice. Earlier research identified lists of CI anti-patterns: ineffective solutions to recurring problems that should generally be avoided. Furthermore, developers often tend to change from one CI tool to the other over time. Earlier research found that developers mostly change from Travis CI to GitHub Actions in particular. It has not been observed in earlier work what the effects of changing from one CI tool to the other are on the CI pipeline with respect to CI anti-patterns. Therefore, we have performed an empirical analysis on CI anti-pattern data of 110 active GitHub repositories that changed from the popular Travis CI tool to GitHub Actions or vice versa. Our research focuses on quantifying three important CI anti-patterns: Slow Build, Broken Release and Late Merging. A custom CI anti-pattern detection and reporting tool called CIAN is written to help solve this problem. We observed that repositories that changed to GitHub Actions have a less variable build duration compared to Travis CI. On the other hand, an increase of approximately 5% of broken builds is observed

Nederlandstalige Samenvatting

Continuous Integration, ook wel gekend als CI, is steeds vaker terug te vinden in populaire git repositories. CI helpt ontwikkelaars om code automatisch te bouwen en testen. Ontwikkelaars hebben meeste baat bij CI als het op een juiste manier toegepast is in hun werk. Aangezien er een groote waaier aan tools bestaat die CI kunnen toevoegen aan een project en dat CI op verscheidene flexibele manieren kan geïmplementeerd worden, is dit in de praktijk minder het geval. Eerdere onderzoeken stelden lijsten samen van antipatronen in CI. Dit zijn wederkerende, ineffectieve oplossingen voor vaak voorkomende problemen en kunnen best vermeden worden waar mogelijk. Verder is het zo dat ontwikkelaars regelmatig opteren om van de ene CI tool te veranderen naar een andere. Een ander onderzoek quantificeerde dit, en bemerkte dat ontwikkelaars het meest veranderen van de Travis CI tool naar de GitHub Actions tool. Wat de effecten zijn op het project en op de CI pipeline met betrekking tot anti-patronen is nog niet eerder onderzocht. In dit onderzoek hebben we een empirische analyze uitgevoerd naar de manifestatie van CI antipatronen voor en na een verandering in CI tool. De focus lag hierbij op projecten die veranderden van Travis CI naar GitHub Actions en vice versa. Met ons onderzoek leggen we de nadruk op het quantificeren van drie belangrijke CI antipatronen: Slow Build, Broken Release en Late Merging. Hiervoor ontwierpen we een nieuwe analysatie tool genaamd CIAN die in staat is om deze antipatronen te analyzeren in git repositories. Data werd verzameld van in totaal 110 projecten die gebruik maakten van CI, geïmplementeerd aan de hand van Travis CI en/of GitHub Actions. We zagen dat projecten die veranderden van Travis CI naar GitHub Actions minder variabiliteit vertoonden in de Slow Build data na de verandering van tool. Anderzijds bemerkten we eveneens dat meerdere instanties van het Broken Build antipatroon zich manifesteerden in deze projecten net na de verandering. Deze verhoging bedroeg ongeveer 5%.

1 Introduction

For this master thesis, I opted to write a paper based thesis instead of a thesis in book format. This allows us to submit the work to a relevant conference. The paper is included in this document under Section 2.

For the time being, the paper is not submitted to any conferences yet, but we are planning to submit it to ...

The work itself is introduced in the paper and thus will not be recited here. The references that were consulted to create this work are included in the paper as well.

Finally, a couple of appendices are added to this document. Appendix A provides some additional details on the proof-of-concept tool CIAN. Appendix B lists the dependencies for both CIAN and the empirical analysis. Appendix C gives an insight in how the work was distributed among the authors and co-authors of this work. Finally, Appendix D shows the relationship with Research Project I and II.

2 Scientific Paper

This section includes the scientific paper starting on the next page.

Empirical Analysis on Continuous Integration Anti-patterns in JavaScript Repositories

Freek De Sagher
dept. Computer Science
University of Antwerp
Antwerp, Belgium
freek.desagher@student.uantwerpen.be

John Businge
dept. Computer Science
University of Antwerp
Antwerp, Belgium
john.businge@uantwerpen.be

Serge Demeyer
dept. Computer Science
University of Antwerp
Antwerp, Belgium
serge.demeyer@uantwerpen.be

***Abstract*—Continuous Integration, referred to as CI, is implemented in a large share of git repositories in order to help automating the building and testing cycle. CI only yields great benefits if it is implemented right. Due to the high availability of different tools and the high flexibility of implementing a certain goal, it is harder to make the right choices in practice. Earlier research identified lists of CI anti-patterns: ineffective solutions to recurring problems that should generally be avoided. Furthermore, developers often tend to change from one CI implementation to the other over time. Earlier research found that developers mostly change from Travis CI to GitHub Actions in particular. It has not been observed in earlier work what the effects of changing from one CI tool to the other are on the CI pipeline with respect to CI anti-patterns. Therefore, we have performed an empirical analysis on CI anti-pattern data of 110 active GitHub repositories that changed from the popular Travis CI tool to GitHub Actions or vice versa. The research focuses on quantifying three important CI anti-patterns: Slow Build, Broken Release and Late Merging. A custom CI anti-pattern detection and reporting tool called CIAN is created to help solve this problem. We observed that repositories that changed to GitHub Actions have a less variable build duration compared to Travis CI. On the other hand, an increase of approximately 5% of broken builds is observed.**

***Index Terms*—continuous integration, anti-patterns, GitHub Actions, Travis CI**

I. Introduction

Continuous Integration (CI) is the process of automatically building and testing code whenever a pulse is received on a repository [1]. These pulses can be receiving commits, the creation of issues or pull requests, whenever someone makes a comment and so on. CI is a development practice where members of a team integrate their work frequently [2]. These integrations are verified by the automated CI pipeline to detect errors more rapidly. CI is commonly found in a large share of git repositories [3]. It fits perfectly in the ecosystem of social coding facilitated by git and social coding platforms like GitHub [4], [5]. By automating steps when project receive updates, the intensive workload of integrating changes or the effort of managing the repetitive tasks to check whether code builds, is working correctly and follows a certain style guide is reduced significantly [6]. CI can be expanded with Continuous Deployment (CD) which facilitates tasks such as automatically publishing code or code artifacts to places where users can access and use it [7]. Generally, the entire CI/CD process improves developer productivity and increases development cycle releases [8], [9].

CI works best if the developers of the software use best practices while coding such as committing often and by using git branches [10], [11]. This reduces conflicts and ensures that the build is continuously executable [9], [12], [13]. Since CI can be implemented in a flexible way, it is challenging to adhere to these standards and problems may arise [10]. Common, but ineffective solutions can be used to solve a recurring problem. These CI ineffective solutions are called anti-patterns and should generally be avoided. Various studies list and categorize patterns (and related anti-patterns) that can occur while integrating and using CI in projects [7], [14]. The catalog of CI patterns/anti-patterns created by Duvall [14] is referenced by most of the related work.

A large amount of tools exist to facilitate the implementation of CI in repositories hosted on platforms such as GitHub [15], [16]. For example GitHub Actions, Travis CI or CircleCI can be used to implement CI. These tools essentially perform the same task: a build server is hosted which can be used to perform the CI activities, such as, building, testing and reviewing code [17], [18]. In addition, a specification can be written that instructs the tool on how and when the entire process of building, testing... should run. Tools such as Travis CI, GitHub Actions or similar offer a huge flexibility in how CI processes can be created in repositories. Large flexibility improves the chance that developers misuse the tools and introduce anti-patterns in their workflows [19]. For example in [10], a tool is created called CI-Odor which can identify some of the more common anti-patterns and report them to help the developers spot these misuses.

Golzadeh et al. conducted a empirical study on the use of CI tools in the repositories from the JavaScript programming language ecosystem that are hosted on GitHub [20]. The authors report that Travis CI and GitHub Actions are the two most prominent tools used to implement CI. They also observed that, while some repositories maintain the same CI tool, others migrate from one CI tool to another. Specifically, developers mostly migrated from Travis CI to GitHub Actions ever since the latter was released in 2019 [21]. Although the study of Golzadeh et al. reports interesting results about the use and migration of CI tool in GitHub repositories, the

study does not give an indication for the possible reasons why developers migrate from one CI tool to another. By leveraging the dataset used in the work of Golzadeh et al. [20], we analyze the CI build logs of data mined from repositories of the JavaScript ecosystem. We hypothesize that the results of the CI anti-pattern quantification in the logs would shed light on the reasons why developers migrate from one CI tool to another.

We do so by answering the following Research Questions (RQs):

RQ1 Which differences are there between CI usage before and after a developer switched from one CI to another with respect to anti-patterns?
An answer to this question can be useful for developers to decide which CI tool to use. If less anti-patterns occur in one tool compared to the other, it gives an additional insight in how easy it can be to set it up and use correctly.

RQ2 Can the difference in anti-patterns explain why developers exchange one CI tool for the other?
An answer to this question is useful to open up further research on anti-patterns in CI. It can show whether or not anti-patterns are an important factor when developers decide to substitute one CI tool for the other.

Our contributions are twofold. For our first contribution, we developed a proof-of-concept tool named CIAN (Continuous Integration Antipattern aNalyzer) that is capable of analyzing CI build logs of repositories in order to extract information on CI anti-patterns. CIAN is strongly based on the CI Odor tool created by Vasallo et al. [10]. Although the goals of both tools are similar, there are some capabilities missing in CI Odor that were required for our purposes. In addition to the repositories written in Java programming language that CI-Odor can analyze, CIAN can analyze repositories written in the programming languages Python, C++ and JavaScript. Furthermore, CIAN is designed to extract and analyze build data from both Travis CI and GitHub Actions, yet CI-Odor only extracts and analyzes build data from Travis CI. Our second contribution is an empirical study. We leveraged the dataset of Golzadeh et al. [20] and explored a sample of 110 actively maintained repositories from JavaScript ecosystem by mining CI data from them. We analyzed the antipatterns present the CI builds of the repositories and report that the results are not sufficient in explaining why developers migrate from one CI build tool to another.

The remainder of this paper is structured as follows. In Section II we give some background information. This includes the anti-patterns that were used and how previous work relates to this work. Furthermore, a motivation for the CIAN proof-of-concept is given including a comparison with existing tools (Section II-B). Then we give an overview of the method for our research in Section III. This includes a section that provides some details on CIAN (Section III-A) and an overview of the steps taken in the empirical analysis. After this, Section IV presents the results we obtained from our empirical analysis. Finally, we present a discussion in Section V with some pointers to future work and conclude the work in Section VI.

II. Background and Related Work

In this section gives an overview of important background concepts, mainly the anti-patterns we considered for our analysis. Furthermore, we also link these concepts with related literature. A motivation for our new proof-of-concept CIAN is given in II-B. This section compares CIAN with the CI-Odor tool which we have used as the main source of inspiration. A focus on the major differences between the two tools is presented.

A. Continuous Integration anti-patterns

Similar to regular software engineering, anti-patterns can emerge when dealing with CI. An anti-pattern is a common, ineffective solution to a recurring problem. Some of these anti-patterns are related to the way in which developers use the pipeline, while others are created by misconfiguring a CI workflow [14], [19].

An exhaustive list of patterns and their anti-pattern counterparts related to CI (and CD) is compiled by Paul Duvall, an active member in the DevOps community [14]. All these (anti-)patterns are combined in categories such as *Configuration Management* or *Testing*. Other work related to CI/CD anti-patterns often reference this list.

Based on the type of anti-pattern and the way it is manifested in the CI pipeline, a detector can be created. The tool used in this work is largely inspired by a tool, CI Odor, [10] that is capable of detecting four interesting anti-patterns in CI.

In our study, we explore four anti-patterns that were reported by the study of Vasallo et al. to be significant importance to most developers [10]. Below we define the four anti-patterns:

- *Slow Build:* This anti-pattern relates to Duvall's *Fast Build* pattern. A *Slow Build* may be caused by a misconfiguration of the CI workflow, for example by specifying too many jobs [19]. In that case it is created by the developer. It can also be caused by the workload on the build server or by the billing plan of the CI tool that is used [22]. Detecting "slow builds" is useful for developers as it gives an insight into how well the CI workflow is designed and how well the build server is behaving. Furthermore, build duration is something that should be monitored as it can become the bottleneck for the entire CI process [23], [24].
- *Broken Release Branch:* This anti-pattern relates to Duvall's *stop the line* pattern. Builds that are triggered on the main or a release branch are called release builds. A build is considered broken if it failed or errored during the CI build procedure. This anti-pattern shows how stable a repository is over time. A period with many broken release builds indicates a period of instability. Further, the downtime can be analyzed. Downtime is the time between a broken build and the first subsequent successful build For developers, the *broken release branch* anti-pattern is useful since it can be used to measure the downtime of a

release. It gives an insight into how long the project was broken.

- *Late Merging:* This anti-pattern relates to Duvall's *merge daily* pattern. Often, features are created using the so-called feature branches. The longer such a branch exists in comparison to the mainline branch, the harder it becomes to integrate it. Similarly, a branch should be kept up to date with the main branch to avoid duplicate, or missed activity. *Late Merging* is subdivided in a couple sub-metrics. First we have *missed activity*. Missed activity quantifies the amount of activity on other branches since the current branch was last merged with the main branch. Missed activity can on its turn be divided in two other components. First there is *branch deviation*. Branch deviation quantifies the amount of changes on other branches since the last update of the current branch. Finally, we have the *unsynced activity* component. Unsynced activity quantifies the amount of activity on the current branch since the last update with the main branch. For developers, the *Late Merging* anti-pattern is useful since it can be utilized to detect their merge strategy. If this anti-pattern is prominently present in a repository, it indicates that the developers might want to change their strategy to reduce integration efforts.
- *Skip Failing Tests* This anti-pattern relates to Duvall's *automated tests* pattern. In some cases, tests are skipped to make a previous broken build pass or to save resources [25]. Although the effect of applying this strategy seems to be positive, it does not solve the cause for a broken build. Making the pipeline pass, just to make it pass defeats the purpose of creating one in the first place. The goal of CI is to detect issues early such that developers can act upon it rather than ignoring it. Detecting this anti-pattern for developers is useful since it indicates places where they might have skipped over important problems.

B. CIAN Tool Motivation

CIAN stands for *Continuous Integration Antipattern aNalyzer* and is pronounced as 'cyan'. It is strongly based on the CI-Odor tool created by Vasallo et al. [10]. CIAN is created from scratch using a recent version of Python 3 (Python 3.10 [26]). The goal of CIAN is similar to the goal of CI-Odor: detecting and reporting anti-patterns in CI. CIAN mines anti-pattern data from the repositories and analyzes this data. More information on the working of the CIAN tool is given in Section III-A.

Although there exist some other tools that are capable of analyzing CI (for example this[1] one which analyzes build times of repositories using the CircleCI tool), none of them are usable for our specific goals. We specifically want to look at anti-patterns in CI, but very little tools exist that perform this kind of analysis. In fact, we have only found CI-Odor that is able to analyse CI anti-patterns in repositories. Although CI-Odor would have been the best fit for our purposes, it was not usable for our specific use case. While CI-Odor is build to analyse repositories written in the Java programming language only, CIAN is able to analyse the programming languages of JavaScript, Java, Python and C++ and can easily be extended for other programming languages.

[1]https://github.com/ashishb/citool

TABLE I
SUPPORTED TESTING FRAMEWORKS PER SUPPORTED LANGUAGE

Repository Language	Supported Testing Frameworks
C++	Google Test, ctest
Java	JUnit
JavaScript	Jest, Mocha, QUnit
Python	pytest

In the Table I and the text that follows, we present other key differences between CI-Odor and CIAN.

- *Repository Language*: as mentioned before, our analysis requires a tool that is capable of analyzing anti-patterns in repositories written in JavaScript programming language. CI Odor does not support these repositories. CIAN is built to handle repositories of different languages and is easily extensible to support for new languages. As an example, support for the popular languages Python and C++ [27] are added to support this claim.
- *Testing Frameworks*: the difference in repository language also implies a difference in supported testing frameworks. Test reports should be mined and analyzed to gather information for the *Skip Failing Tests* anti-pattern. As CI Odor only supports Java, it supports the JUnit testing framework only, as this is one of the most used testing frameworks in the Java ecosystem [28]. Since CIAN supports other repository languages, it should also support other testing frameworks. We have tested CIAN with repositories written in C++, Java, JavaScript and Python and have implemented support for at least one (popular) testing framework per language. The supported frameworks are summarized in Table I.
- *CI tools*: CI-Odor focuses on builds generated by Travis CI only. It utilizes the Travis API to gather information from the builds. This is a limiting factor to the power of the tool as more repositories are shifting towards using GitHub Actions over Travis CI [20]. Furthermore, we want to make a comparison between GitHub Actions and Travis CI in our analysis. Therefore, CIAN supports both these CI tools.
- *Late merging difference*: CI Odor collects a metric called branch age for this anti-pattern. With the GitHub API it is not possible to mine historical data on branches, it is not possible to analyze this metric with CIAN.
- *Broken Release difference*: CI Odor generates warnings for weeks where there are substantially more broken releases compared to the other weeks. This kind of warnings is not useful for our analysis, so this was not implemented in CIAN.

- *Availability*: The source code of CI Odor is not hosted on a public git repository. CIAN has its own public MIT licensed repository[2] and is extensible by anyone that want to extend upon it.
- *Collection date range*: CI Odor automatically collects information for the last three months of a repository. In our analysis, it is important to gather information from specific periods in time. With CIAN it is possible to select a range of dates in which we want to analyze data.

CIAN Goals and Tasks We set a few goals for the CIAN and decided which tasks it should be able to perform.. First, the tool should be capable of taking one or more git repositories as input. As a first task, the tool should collect information on the CI build tools that are used in those repositories. This incorporates both detecting whether or not a certain tool is implemented in a repository and whether or not the tool has been used (builds are generated). If builds exist, the tool should be able to mine important information from them. It should also be possible to select a date range in which to collect build information. For "skipped failing tests" anti-pattern, it is required to collect test result information. Test results are stored in build logs which should also be mined and analyzed. Then, analysis should be performed on the mined information of the builds and build logs such that anti-patterns can be detected in the information. Finally, the tool should output its results both in a graphical way and in a textual way. Graphs can be used to immediately analyze results, while textual output can be used for further processing or analysis.

All functionality of the tool should be easy to use using a command line interface. Furthermore, the source code of it is kept open source to allow for future improvements, fixes and adjustments if required.

C. CI Tools

Our research focuses on two prominent tools in the CI landscape: Travis CI and GitHub Actions. Travis CI the first CI tool that provided free CI services to open source projects [29]. This explains why it became one of the most popular CI tools of its time. With the advent of GitHub Actions in 2019 [21], the popularity of Travis CI started to decline in favor of GitHub Actions [20]. Due to the popularity of these two tools, and due to the fact that this shift in popularity is so prominent, we decided to use these two CI tools to mine build information from for our analysis.

III. Study Design

This section explains our approach to solve the research questions posed in Section I. First, we discuss the proof-of-concept CIAN we created which helped us to analyze the CI builds (Section III-A). Then, we discuss the methods used for the empirical analysis on CI anti-patterns in git repositories.

The methods of the empirical analysis is summarized in Figure 1. Four distinct steps can be identified in the process. First, there is the raw data collection, then CI workflow build collection, followed by anti-pattern detection and finally the results are aggregated and interpreted. Each of these steps is explained in Sections III-B - III-D. Finally, in Section III-E, we discuss the limitations we have encountered in our research.

A. CIAN Tool

The following section presents a high level overview of the CIAN tool. It reuses formulas used in CI Odor to collect similar information on antipatterns. In Section III-A1 we give a broad overview of how the tool works. For a more elaborate description of the workings of the tool, Appendix A and the GitHub repository[3] can be consulted.

1) Architecture Overview: Figure 2 gives a broad overview of the architecture of the tool. It consists of three main layers: a data collection layer (with the build info collection and CI tool detection as components), a detection layer and an output layer. We used one of the newest stable versions of Python (Python 3.10) to build everything.

The data collection layer utilizes the GitHub API [30] for information about the repository. Furthermore, the GitHub Actions API is used to mine build information from. For Travis CI repositories, the Travis CI API v3.0 [31] is used. In order to save resources, collected data is stored in JSON cache file.

The second layer is the anti-pattern detection layer. It is the heart of the tool; the most important calculations happen in this layer. The anti-pattern detection components contain the same formulas used by CI Odor to extract anti-patterns from the mined data.

Finally, the output layer is capable of generating graphics that can be used to immediately analyze a certain repository. Furthermore, it can output textual (JSON) files that can be used for further analysis.

B. Repository collection

We reused the dataset assembled by Golzadeh et al [20]. This dataset contains a huge amount of repositories from the JavaScript ecosystem on GitHub. This data inhabits information on these repositories (eg. the used CI tools, the time at which a CI tool was first used, the time it was last used, the first and latest commit of the repository and the gap time at which no CI was used). By using this data it is possible to analyze the evolution of CI in the repositories, as performed by the authors of the paper.

Based on this immense data set, four data classes were extracted. First, repositories that used GitHub Actions (`GHA Only`) only or Travis CI only (`Travis CI Only`) were collected. Second, repositories that evolved from GHA to Travis CI (`GHA→TravisCI`) and the other way around (`TravisCI→GHA`) were collected. The former two classes were collected to give a broader perspective to the latter two classes. Note that besides GHA or Travis CI, many other CI tools were present in the dataset. Since most data points used one of these tools, the research is limited to GHA and Travis CI.

[2]https://github.com/FreekDS/CIAN

[3]https://github.com/FreekDS/CIAN

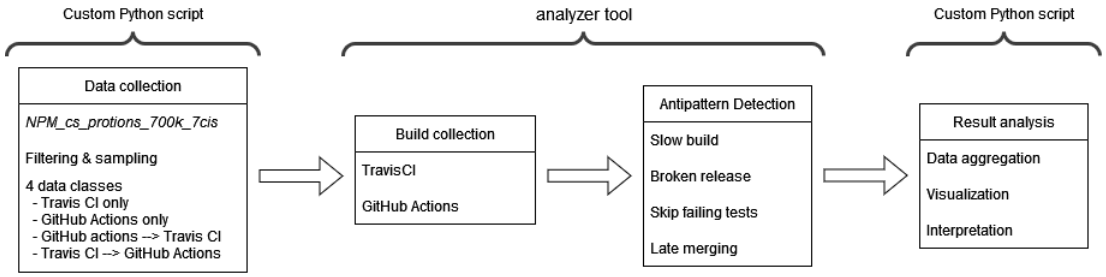

Fig. 1. The entire flow of the conducted research

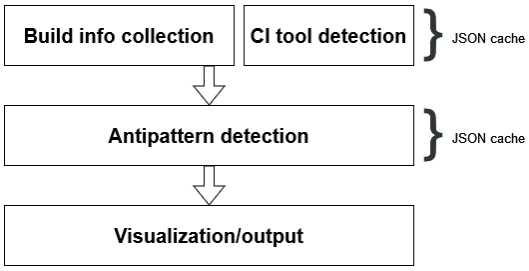

Fig. 2. Large overview of CIAN architecture

A total of 110 repositories were analyzed among the four different data classes. 30 for each data class except for `GHA → TravisCI`. For this class 20 repositories were analyzed.

C. Build Collection and Anti-Pattern Detection

After the repositories are determined, it is possible to collect data from them using the appropriate APIs. GitHub Actions build information is gathered from the GitHub API [30] while Travs CI build information is gathered from the Travis CI API [31]. When builds are collected, they can be analyzed to extract anti-patterns from them. These two steps were performed by the custom written analyzer tool CIAN. It is responsible for collecting builds from GitHub Actions and Travis CI and for analyzing them to extract the anti-patterns per repository.

CIAN is able to extract the four anti-patterns described in Section II: *late merging*, *broken release*, *skip failing tests* and *slow suild*. The tool is executed with all repositories of each of the four data classes (110 repositories in total). Collecting all builds of the entire lifetime of each of these repositories would not be feasible as mining builds from Travis CI and GitHub APIs takes a significant amount of time and resources. To mitigate this, CIAN was instructed to only collect builds in the range of 90 days after a specified date.

D. Result Analysis

The final step in the analysis is to aggregate, analyze and interpret the results collected by the previous steps. For each data class, all anti-pattern data was combined and visualized in box plots using Python scripts[4]. By utilizing the generated plots, the results of the anti-patterns were interpreted. The results of the analysis are described in Section IV.

[4]https://github.com/FreekDS/Git-CI-Antipattern-Analysis

E. Limitations

There are a few important limitations that popped up while working on the analysis. This section describes them and indicates the way in which they were handled.

The first limitation is the size of the entire dataset. The entire dataset was too large to be used in the analysis, even after filtering out repositories that did not use Travis CI or GitHub Actions. To mine information on all repositories, a large amount of network bandwidth and access tokens should be used. Furthermore, it would take hours, if not days to collect all data. To solve this issue, a smart sample is taken from the dataset. While sampling, a few filtering criteria were used to collect the most interesting repositories:

- Repositories that were not active anymore at the time of sampling were not used. A repository is considered inactive if the date of the latest commit (t_{lc}) is more than 30 days before the analysis date (t_a)

$$t_a - t_{lc} > 30 \tag{1}$$

- Repositories that do not have builds from the CI tool(s) that they use over their entire lifetime are removed from the sample.

The sampling method ran until it collected the required sample size. For each data class, a sample size of 30 was picked. This is possible for all data classes, except for `GHA→TravisCI`. As it turns out, it does not happen often that repository developers move from GitHub Actions to Travis CI. Only a total of 20 repositories are present in the immense dataset. This is due to *"the fall of Travis CI"* [20].

A second limitation is related to the age of the repository dataset. At the time of the analysis, the dataset reached the age of approximately 6 months. The CI landscape is ever-changing at a rapid rate. This has as consequence that some repositories are not up to date anymore. For example, it is possible that a repository that only used Travis CI decided to change to another tool in the last 6 months. Sometimes it is even the case that a repository is removed entirely. To overcome this limitation, a time range is set for the build collection. CIAN is instructed to only mine builds from a specific timespan. First the newest date of the repository dataset is collected. For this, the newest 'latest commit' entry of the dataset is taken (t_{latest}). The analyzer tool is executed to only collect data

starting from 90 days before this latest date until the latest date for both the GHA only and Travis CI only data classes.

$$range = [t_{latest} - 90, t_{latest}] \tag{2}$$

In order to make a comparison from before and after the CI change, CIAN was instructed to mine builds from 45 days before the change in CI and 45 days after the change (t_{change}). This gives us a total of a 90 day timespan as well.

$$range_{before} = [t_{change} - 45, t_{change}] \tag{3}$$

$$range_{after} = [t_{change}, t_{change} + 45] \tag{4}$$

A third limitation is imposed by the age of the repository dataset as well. Information on the *Skip Failing Tests* anti-pattern cannot be collected. To detect this anti-pattern, build logs should be analyzed of the repositories. Unfortunately, GitHub does not keep build logs longer than 90 days by default [32]. For that matter, the Skip Failing Tests anti-pattern is not considered in the empirical analysis that tries to find differences among the different data classes. This is one of the biggest limitations to our research.

A final limitation is related to the *Late Merging* anti-pattern. The *Late Merging* anti-pattern uses branch information to gather metrics that tell us useful information about the repository. Unfortunately, GitHub does not store historical data on branches, which means that for this anti-pattern, only the present can be mined and analyzed. The anti-pattern is not removed from the empirical analysis as it gives an insight in how the repositories are managed among the different data classes.

IV. Empirical Analysis Results

The following sections describe the results that we collected after aggregating and analyzing the data gathered by the analyzer tool. Section IV-A gives a broader insight in the repository dataset by providing some additional, general information on the repositories and by using the data aggregated from the *Late Merging* anti-pattern. Sections IV-B and IV-C describe the variation in the *Slow Build* and *Broken Release* anti-patterns respectively. In order to compare `Travis CI→GHA` and `GHA→Travis CI` with each other, box plots are generated for each metric of each anti-pattern. Similarly, a comparison of the `Travis only` and `GHA only` classes is made.

A. Repository Information

In order to sketch an overview of how the repositories look like and how they are managed, we gathered some general information on the repositories of each class. This information does not only include the amount of issues, contributors, pull requests and stars, but also the data gathered by the *Late Merging* anti-pattern. The *Late Merging* anti-pattern is separated into a couple submetrics [10]. All of them are discussed in this section.

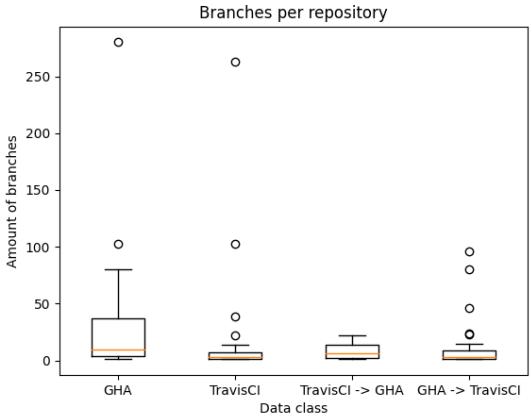

Fig. 3. The amount of branches for each repository in each data class

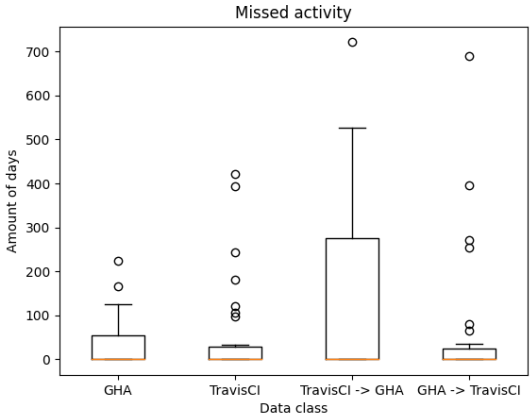

Fig. 4. The average amount of days for the missed activity in each repository in each data class

1) Late Merging Data: First, we take a look at the total amount of branches in the repositories (see Figure 3). This gives us an insight in how the code is managed over different source branches among the data classes. Overall, the amount of branches is similar among all repositories in all data classes. This is indicated by the median value of the boxes which is around 10 or lower. The `GHA only` data class contains the repositories that have the most branches. This is shown by the size of the box plot and the outliers that can reach values higher than 100. Furthermore, the repositories that are using Travis CI at the moment (the `TravisCI only` and `GHA → TravisCI` data classes), generally have less branches than the repositories managed by GitHub Actions. The amount of branches in a repository could have a direct influence on the other metrics gathered by the *Late Merging* detector.

Secondly, we can take a look at the data distributions of the missed activity sub-metric (Figure 4). Missed activity is a quantifier for the integration effort of feature branches. The larger the value is, the more effort is required to merge the branches into the mainline branch of the repositories.

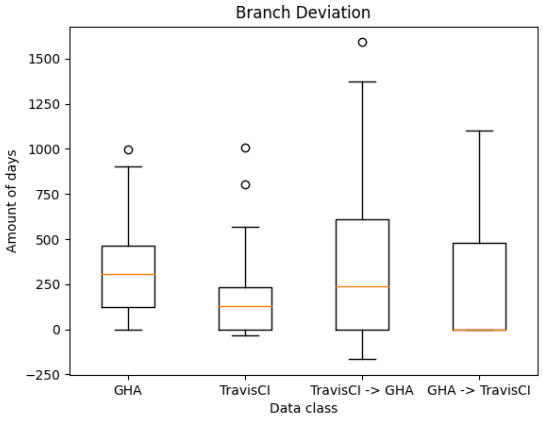

Fig. 5. The average amount of days of branch deviation in each repository in each data class

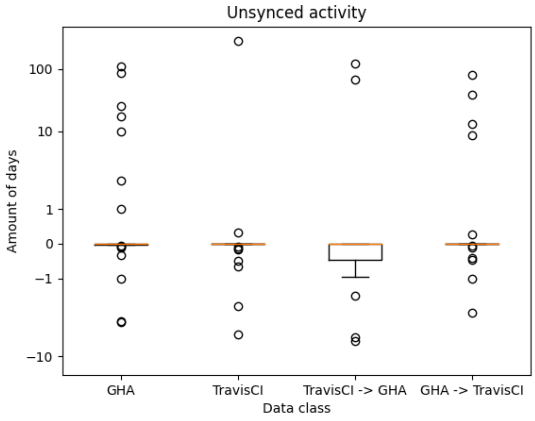

Fig. 6. The average amount of days of unsynced activity in each repository in each data class in logarithmic scale

The trend shown in Figure 4 shows that in general, most repositories have 0 days of missed activity among all data classes. This is indicated by the median values which are all at 0. Furthermore, a similar trend can be seen as with the branch count: the repositories managed by GitHub Actions have the tendency to have a higher missed activity and thus a higher integration effort. This conclusion is drawn from the box sizes of `GHA only` and `TravisCI` $\rightarrow$ `GHA` which are larger than their Travis CI counterparts.

A third analyzed metric is the branch deviation (Figure 5). This is a sub-metric extracted from the missed activity metric and indicates the amount of changes in the branches since the last update. Again, the repositories managed by GitHub Actions seem to exhibit a larger deviation in the branches than the repositories that have Travis CI implemented. The median values for these classes are higher than the `TravisCI only` and `GHA` $\rightarrow$ `TravisCI` data classes. Another observation is that only for the `TravisCI` $\rightarrow$ `GHA` data class, negative branch deviation is reported. This means that in such a repository, there are branches that are quite far ahead of the others.

The final metric analyzed by the *Late Merging* detector is the unsynced activity (Figure 6). Unsynced activity indicates the integration effort that is required to merge a branch with the main branch. Important to mention is that in Figure 6 a logarithmic scale is used. What immediately is noticeable is the fact that most data classes do not generate a box. This means that in most cases, branches are merged with the mainline relatively often. Negative unsynced activity is an indication of how far branches are ahead of the mainline. We notice that only for the `TravisCI` $\rightarrow$ `GHA` data class, a box is shown with a value that goes to -1. This means that in this data class, there are slightly more repositories with branches that are ahead of the mainline branch.

2) Other Repository Information: Besides the metrics extracted from the *Late Merging* anti-pattern, a few additional metrics are extracted from the repositories that indicate the traffic and popularity of the repositories. For this, we have looked at the amount of contributors, stars, commits and issues (see Figures 7, 8, 9 and 10). Besides that, we have looked at the amount of CI workflows that are implemented in each data class (Figure 11).

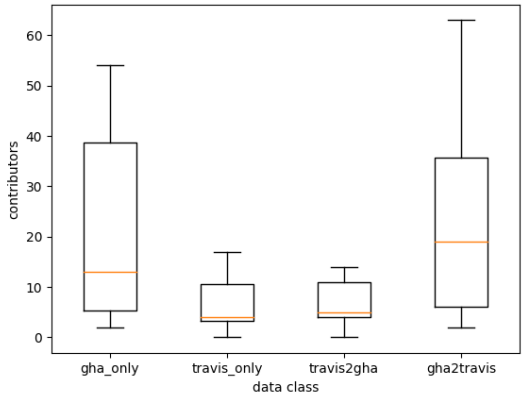

Fig. 7. Amount of contributors per repository

From this data, we notice that repositories in the `GHA Only` and `GHA` $\rightarrow$ `TravisCI` are the most popular among the public. This is indicated by the amount of stars being larger than with the other classes. This also explains why these two classes have more issues: more people that use the code leads to more people that might have a problem, which ultimately leads to more issues created in the repository.

Furthermore, it can be seen that the `GHA only` and `TravisCI` $\rightarrow$ `GHA` data classes generally have more commits than the other classes. This is important as continuous integration often runs based on some commits that are pushed to a repository. This means that there is more opportunity for CI data in these two classes compared to `TravisCI only` and `GHA` $\rightarrow$ `TravisCI`. Opportunity for CI data is influenced by the amount of different workflows that are

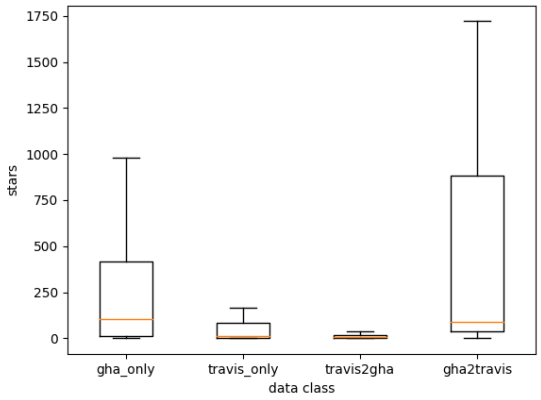

Fig. 8. Amount of stars per repository

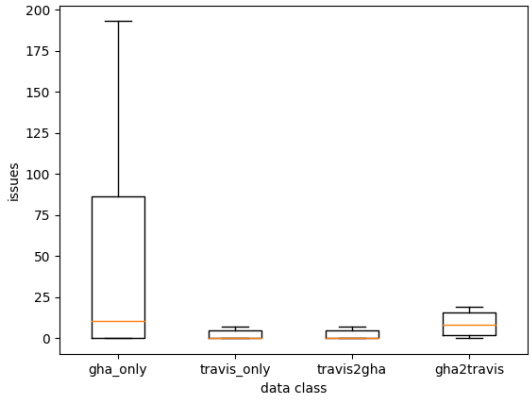

Fig. 10. Amount of issues per repository

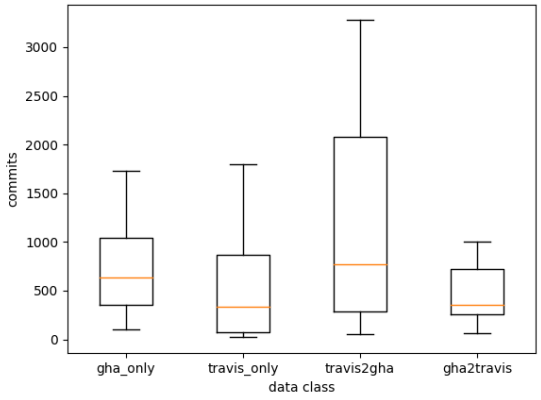

Fig. 9. Amount of commits per repository

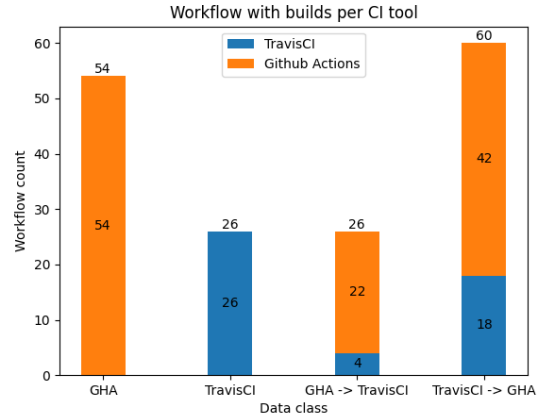

Fig. 11. Active workflow count for each data class

present in the data. Figure 11 shows these amounts in a stacked bar plot. A workflow is only counted if it has at least generated one build for the repository. Based on this information, we see that for the `TravisCI only` class, only 26 out of 30 repositories have an active workflow. Four repositories that were sampled, did not have a build in the specified time frame. Furthermore, for the `GHA` $\rightarrow$ `TravisCI` class, only 4 repositories of the 20 that were available have data generated by Travis CI. This observation enforces the fact that the data in this class is extremely sparse. Finally, we see that for the `TravisCI` $\rightarrow$ `GHA` data class, not all repositories have generated builds with Travis CI before they changed to GitHub Actions.

B. Slow Build Results

Two different comparisons are made, one that compares the build duration of `Travis only` with `GHA only` repositories and one where `Travis CI`$\rightarrow$`GHA` build durations are compared with the durations of `GHA`$\rightarrow$`Travis CI`. These two comparisons are highlighted in the subsequent sections. Box plots are generated for each data class allowing them to be compared easily. A single datapoint is represented as the average duration of almost all workflows combined. Not all workflows are used to calculate this average. Using the Inter Quartile Range (IQR) formula [33] for detecting outliers, workflows that are categorized as outliers (in comparison to other workflows present in a repository) are not utilized in the calculation of the average. This is important as in some cases, a workflow that performs heavy tasks such as building all documentation, exceeds the average duration of other workflows by a lot.

1) Single CI results: Figure 12 displays the distribution of the repositories that use a single CI tool during their lifetimes with respect to the *Slow Build* anti-pattern. It shows outliers as circles and the average as a green triangle. The orange line in the box plot indicates the median value.

Based on this plot, a few observations can be made. The difference between `GHA only` and `TravisCI only` repositories is not extremely large. We do notice that both quartile group 1 and quartile group 4 of the `GHA only` repositories is smaller than with `TravisCI only` repositories. This is indicated by the length of the whiskers

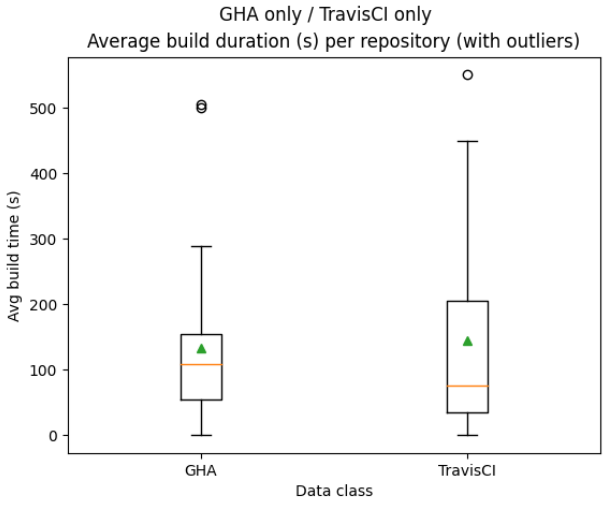

Fig. 12. Box plots representing the distribution of the repositories with respect to average build duration (in seconds). The green triangle indicates the mean value.

of the boxes. Furthermore, the box size of `GHA only` is smaller than the other. This indicates that the distribution of the slow build averages of `GHA only` repositories is less variable than with `TravisCI only` repositories. The average of both distributions is almost equal (approximately 120 seconds), but the median value of *Slow Build* in the `GHA only` class is slightly higher with approximately 20 seconds. This means that there are more repositories with (slightly) higher build times in this class compared to `TravisCI only`. In addition, we can see that there are more outliers with the former in comparison with the latter. Finally we can see that in both classes, the minimal average build time is 0s. That is because there are repositories that have no generated workflow builds in the timespan that was selected for the analysis.

2) Repositories that changed CI results: Here, combining the two classes does not make much sense. Better is to compare right before the change with right after the change to notice any difference.

a) GHA → TravisCI: An example of a repository in this class and the distribution box plots are shown in Figure 13. Recall that the data in this class is extremely sparse: there were only 4 repositories that have data on Travis CI builds after the change. This results in the box plot after the change to be completely squashed to a single line with a median duration of 0s. It makes comparing the state of the repository before the CI change with after the CI change impossible. The only metric that can be compared is the outliers. There we see that after the change, the outliers are slightly higher than before the change with values ranging from 75 seconds to more than 200 seconds.

The provided example strengthens this claim, since it clearly shows an increase in build duration after the developers changed to Travis CI. It shows the *Slow Build* results of the WebSSH2 repository before and after the change to Travis CI. It is clear that GitHub Actions was faster in building the project than Travis CI did in this particular case.

b) TravisCI → GHA: Similarly, Figure 14 shows an example of slow build results for a repository (bhovhannes/additween) and the box plots that represent the distribution. Based on the box plots, a few conclusions can be drawn. There is no great difference in the median value (before: 60s, after: 50s). Although he median is slightly higher for the repositories before the change to GHA, the difference with after the change is negligible. Furthermore, the box size of the box plot before the change is larger than after the change. This is the same trend as observed in the comparison of `TravisCI only` with `GHA only`. Finally, we see that there are only a couple outliers after the change and no from before the change.

The presented example displays *Slow Build* data right before and right after the change in CI tool. In this case, GHA achieves lower build durations than the previously used workflows created and managed by Travis CI.

C. Broken Release Results

For the analysis of broken release builds among all repositories, some additional processing is performed to allow for a better comparison. First, repositories that do not have release builds (repositories that have no workflow that builds the main branch) are not considered. Rather than comparing absolute values, we opted for a comparison of percentages. The amount of failing release builds is divided by the total amount of release builds to obtain a percentage.

Each data point in the box plots simply represents the total broken release percentage of a repository. The analysis is separated in two parts: first the repositories of the `GHA only` and `TravisCI only` classes are compared. Secondly, a comparison is made before and after the change of CI tool of both the `GHA` → `TravisCI` and `TravisCI` → `GHA` classes.

1) Single CI: Figure 15 shows the distribution of the repositories in the `TravisCI only` and `GHA only` classes in a box plot.

From this graphic, a few observations can be made. First, we see that the general trend among the repositories is to have a low percentage of broken release builds. This is indicated by the median value which remains (close to) zero. When the median is at 0%, it means that at least 50% of the repositories in that class have a broken release percentage of 0%. Do note that with `GHA only` the median value is slightly higher than with `TravisCI only`. This means that in the former, there are more repositories with a broken release percentage higher than 0% than in the latter.

Furthermore, it can be seen that with the `GHA only` class, the box size is significantly smaller. Thus, among `GHA only` repositories, there is less variability in broken release percentage than with `TravisCI only`.

In addition, we can notice that the box of `TravisCI only` has a way longer upper whisker indicating that there

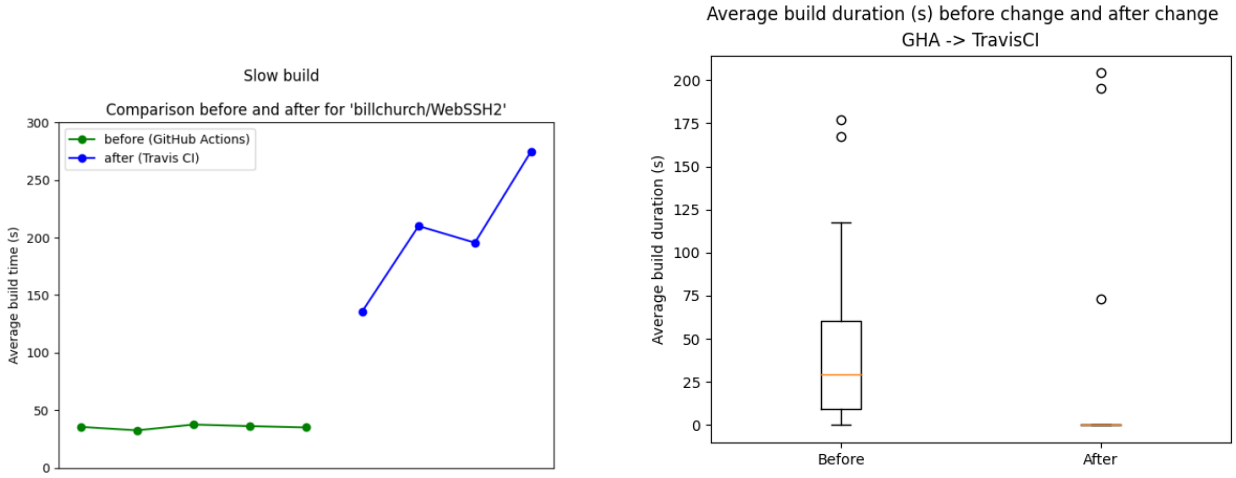

Fig. 13. An example of *Slow Build* in a repository (billchurch/WebSSH2) before and after the CI change (left) and distribution of all repositories before and after change (right), all from the `GHA → TravisCI` class

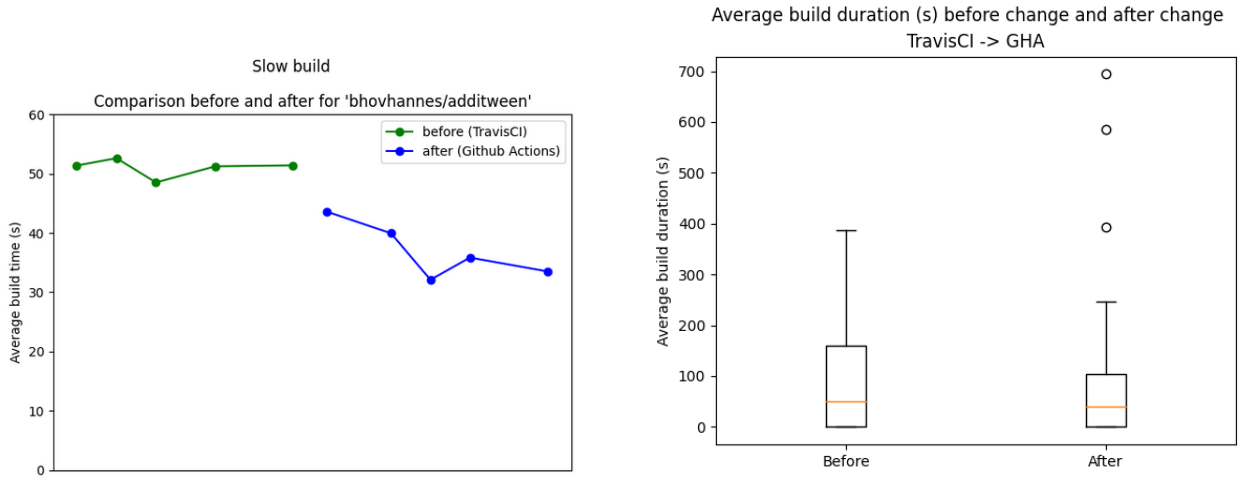

Fig. 14. An example of *Slow Build* in a repository (bhovhannes/additween) before and after the CI change (left) and distribution of all repositories before and after change (right), all from the `TravisCI → GHA` class

are more repositories in the 4th quartile (with a relatively high broken release percentage).

Each class has a few outliers. For `GHA only` outliers vary between 60 and 100% while for `TravisCI only` outliers all have a broken release percentage of 100%.

2) Repositories that changed CI results: Results are separated in the two classes. For each class, we look at the state of the repositories before the change in CI tool and compare it with the state of the repositories in the dataset after the change in CI tool.

a) GHA → TravisCI: Remember that in this class the data is extremely sparse so comparisons are harder to make. The results for this data class are shown in Figure 16 represented as a box plot. From this data, before the change, 50% of the repositories had a broken release percentage lower than 50% and the other halve has more than 50% broken releases. This is indicated by the median value being centered perfectly in the middle of the box and by the box spreading from 0 to 100%. After changing to TravisCI, this is entirely different. The median value is significantly lower than 50% and the highest outlier of this data even is lower than 50%.

b) TravisCI → GHA: Figure 17 displays the distribution of this class before and after the change in CI. First, we notice that before the change in CI, the repositories are all relatively stable. The median value is at 0 indicating that most repositories have no broken release builds. There are some outliers, but none of them have a higher percentage than 40%. After the change however, this is different. Now the median value appears at a significantly higher value (around 8%). This means that after changing to GHA, the repositories became more unstable with respect to the broken release branches. This is emphasized by the larger box plot size and by the

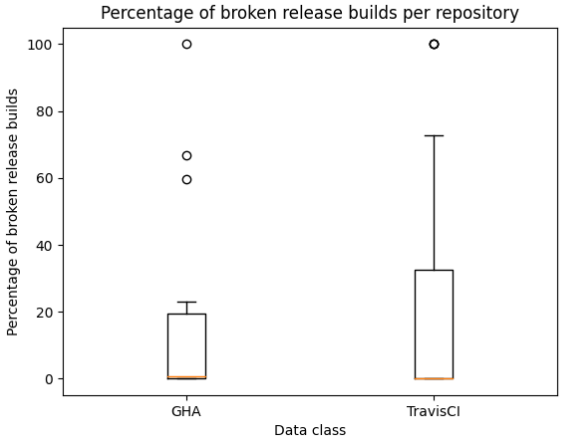

Fig. 15. Distribution of broken release percentages per repository for repositories that used a single CI tool

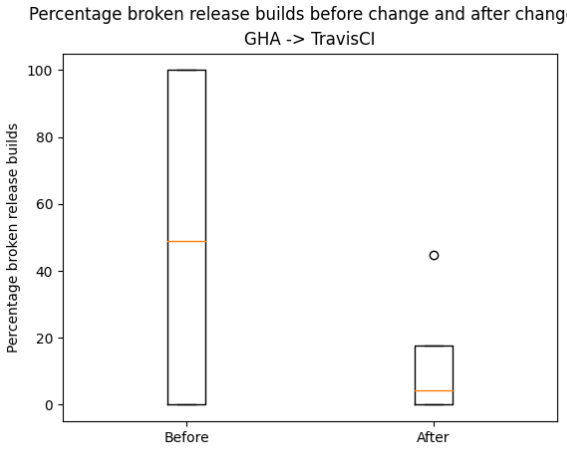

Fig. 16. Distribution of broken release percentages before the change in CI tool (from GHA) and after the change in CI tool (to Travis CI)

outliers having reaching up to 80%.

V. Discussion

The results presented in section IV allow for some additional interpretation and discussion. The following sections provide further insight in the obtained results. Section V-A tries to find answers among why some results were obtained. Section V-B points to potential problems that might have influenced the observations made and section V-C gives some pointers for future work.

A. Interpretation of Results

Based on the obtained results, a few additional interpretations can be given to the data.

From the general repository information, we have seen that repositories using GitHub Actions usually implement multiple workflows in the same repository. This is not the case with Travis CI since it is impossible to define multiple workflows

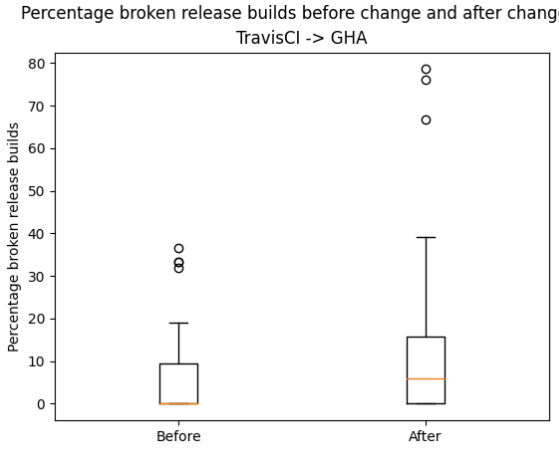

Fig. 17. Distribution of broken release percentages before the change in CI tool (from Travis CI) and after the change in CI tool (to GHA)

in a single repository using Travis CI [34]. By looking at the amount of stars, commits and contributors of repositories we have seen that the more popular repositories are favoring GitHub Actions over Travis CI. This can perfectly be tied to the *fall of Travis CI* and the *rise of GitHub Actions* [20].

Furthermore, we noticed that the anti-pattern results for both *Slow Build* and *Broken Release* showed that GHA has less variable data than Travis CI. This is especially true for the single CI repository classes. It indicates that among the repositories of our sampled dataset, GHA works more stable and reliable than Travis CI. This variability may be explained by the limiting factors that Travis CI imposes on their free plan, as mentioned in [20].

Do note that this observation is not true for the Broken Release results in the `TravisCI` $\rightarrow$ `GHA` data class. Here, we noticed that GHA had more broken release among the repositories than Travis CI had before changing. In that light, it is strange that developers decide to change from this tool to the other. Future research can focus on this observation to see whether this is true on larger datasets. Furthermore, it is possible that developers need some time to get accustomed to the new CI tool and are prone to make more mistakes when they start using GHA.

B. Threats to Validity

This section summarizes possible threats that could possibly have an influence on the obtained results.

- Construct validity: although the analysis tool is thoroughly tested, it is still possible that some bugs are in the code. The obtained results where manually verified for some repositories to make sure that the analysis data obtained from the tool are reliable and correct. This does not exclude any unforeseen problems. To mitigate this threat, unit tests for the important components of the tool are written to make sure that they behave as they were intended to.

- API availability: during the development of CIAN, problems were encountered while using the various APIs used to gather the data (eg. GitHub API or Travis CI API). In some cases the API did not behave as expected resulting in issues with the data collection. By manually monitoring the gathered data, the effects of this threat are minimized.
- Developer experience: in the analysis, a comparison is made between before and after changing to a particular CI tool. We found that after changing, more broken releases were encountered. This may be influenced by the developer experience with the new CI tool. It is likely that developers try out this new CI tool and make more mistakes in the beginning as a part of their learning curve. This may or may not influence the future behavior of the CI workflows in the repositories. To find out whether or not this is the case, additional research should be performed.
- Comparison bias: comparing workflows of two different CI tools directly is hard. When a repository changes from one CI tool to another, the developers are not obligated to follow the same CI structure as they had before. It is entirely possible that large workflows are separated into many smaller workflows, or the other way around. Furthermore, it is possible that more CI tasks are performed before or after a change. There is no formal way to compare workflows directly before and after a change in CI tool. We can only guess that the same tasks are performed before and after changing tools. The way in which this problem is managed in this research is by taking smart averages among the different workflows that are present in a single repository. Smart means that workflows that exhibit outlier behavior are simply not used in the analysis. Averaging out the values of the different workflows in a repository only works if we make the assumption that the behavior of the implemented CI does not change significantly when a new, different CI tool is picked.

C. Future Work

The observations we made with this empirical analysis open up opportunities for future research. It may be interesting to compile a larger dataset for the `GHA` $\rightarrow$ `TravisCI` class in order to see whether our hypothesis still holds that GHA behaves less variable compared to Travis CI with respect to the discussed anti-patterns.

Furthermore, our research focused on the time frame right before a change in CI tool and right after a change. Additional research can be performed to see whether the manifestation of CI anti-patterns changed further down the line. For example, as projects tend to become larger, their CI pipelines have the tendency to run slower [35]. This hypothesis can be verified by looking at the *Slow Build* anti-pattern at a later time after the CI change. On the other hand, developers might become more experienced and enhance efficiency of the CI pipeline over time.

Another possible point of research may be related to the *Broken Release* anti-pattern. We noticed that an increase in broken release occurrences happens right after the change to GitHub Actions. Whether or not this still is the case after developers gathered additional experience with GHA is not clear from our data.

It might also be interesting to look at the time it takes for developers to fix a main branch after a broken release occurred. Our hypothesis is that this time decreases as CI evolves in a repository.

Finally, our research does not find a concise answer to the 'why' question: why do developers pick another CI tool over the other? To find a possible answer to this question, a survey could be performed among developers of projects that changed from one CI to the other. It would be interesting to see if such a survey confirms the observations we made in this research.

VI. Conclusion

With this paper we have performed an empirical study on anti-patterns in CI implementations in repositories that used multiple tools throughout their lifetime. In particular we looked at three prominent CI anti-patterns: *Slow Build, Broken Release* and *Late Merging*. To achieve this, we have built a proof-of-concept called CIAN based on the CI-Odor tool built by Vasallo et al. [10]. Initially we also wanted to analyze the *Skip Failing Tests* anti-pattern, but due to limitations regarding the GitHub API it was not feasible to collect data on this anti-pattern.

By utilizing CIAN and by analyzing the CI build data from 110 git repositories, we compared anti-pattern data of GitHub Actions based implementations with Travis CI based implementations. We saw that in repositories that use GitHub actions, or repositories that changed to use GitHub Actions, the anti-pattern data for *Slow Build* was observed to be less variable compared with Travis CI implementations. Furthermore, we have observed that the most popular repositories are more likely to use GitHub Actions rather than Travis CI. This confirms the rise of GitHub Actions and fall of Travis CI as mentioned by [20].

Finally, we noticed that after changing from Travis CI to GitHub Actions, a slight increase in *Broken Release* builds was present. This may be explained by the inexperience of developers with the new GitHub Actions CI tool. This requires additional research to be confirmed.

References

[1] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in github," in *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, 2015, pp. 805–816.

[2] M. Fowler and M. Foemmel, "Continuous integration," 2006.

[3] B. Vasilescu, S. van Schuylenburg, J. Wulms, A. Serebrenik, and M. G. van den Brand, "Continuous integration in a social-coding world: Empirical evidence from github," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 401–405.

[4] T. Kinsman, M. Wessel, M. A. Gerosa, and C. Treude, "How do software developers use github actions to automate their workflows?" in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 420–431.

[5] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social coding in github: transparency and collaboration in an open software repository," in *Proceedings of the ACM 2012 conference on computer supported cooperative work*, 2012, pp. 1277–1286.

[6] T. Kinsman, M. Wessel, M. A. Gerosa, and C. Treude, "How do software developers use github actions to automate their workflows?" in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, 2021, pp. 420–431.

[7] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[8] A. Poth, M. Werner, and X. Lei, "How to deliver faster with ci/cd integrated testing services?" in *European Conference on Software Process Improvement*. Springer, 2018, pp. 401–409.

[9] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[10] C. Vassallo, S. Proksch, H. C. 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*, ser. ICSE '19. IEEE Press, 2019, p. 105–115. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00028

[11] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu, "The impact of continuous integration on other software development practices: A large-scale empirical study," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 60–71.

[12] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An explorative analysis of travis ci with github," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 356–367.

[13] C. Chandrasekara, "Hands-on github actions."

[14] P. M. Duvall and M. Olson, "Continuous delivery: Patterns and antipatterns in the software life cycle," *DZone refcard*, vol. 145, 2011.

[15] M. Meyer, "Continuous integration and its tools," *IEEE Software*, vol. 31, no. 3, pp. 14–16, 2014.

[16] M. Shahin, M. Ali Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017.

[17] C. Chandrasekara and P. Herath, "Introduction to github actions," in *Hands-on GitHub Actions*. Springer, 2021, pp. 1–8.

[18] GitHub, "What is GitHub Actions?" https://resources.github.com/downloads/What-is-GitHub.Actions_.Benefits-and-examples.pdf.

[19] 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.

[20] M. Golzadeh, A. Decan, and T. Mens, "On the rise and fall of ci services in github," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 662–672.

[21] T. Chen, Y. Zhang, S. Chen, T. Wang, and Y. Wu, "Let's supercharge the workflows: An empirical study of github actions," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2021, pp. 01–10.

[22] D. G. Widder, M. Hilton, C. Kästner, and B. Vasilescu, "A conceptual replication of continuous integration pain points in the context of travis ci," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 647–658.

[23] E. Bisong, E. Tran, and O. Baysal, "Built to last or built too fast? evaluating prediction models for build times," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017, pp. 487–490.

[24] T. A. Ghaleb, D. A. Da Costa, and Y. Zou, "An empirical study of the long duration of continuous integration builds," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2102–2139, 2019.

[25] X. Jin and F. Servant, "Which builds are really safe to skip? maximizing failure observation for build selection in continuous integration," *Journal of Systems and Software*, vol. 188, p. 111292, 2022.

[26] "Python Release 3.10.0 - Python.org," https://www.python.org/downloads/release/python-3100/, accessed: 2022-08-19.

[27] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, and L. Réveillere, "Popularity, interoperability, and impact of programming languages in 100,000 open source projects," in *2013 IEEE 37th annual computer software and applications conference*. IEEE, 2013, pp. 303–312.

[28] P. Stefan, V. Horky, L. Bulej, and P. Tuma, "Unit testing performance in java projects: Are we there yet?" in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, 2017, pp. 401–412.

[29] Wikipedia, "Travis CI — Wikipedia, the free encyclopedia," http://en.wikipedia.org/w/index.php?title=Travis%20CI&oldid=1094591640, 2022, [Accessed: 2022-08-23].

[30] "GitHub REST API - GitHub Docs," https://docs.github.com/en/rest, accessed: 2022-08-19.

[31] "Travis CI APIs - Travis CI," https://docs.travis-ci.com/user/developer/, accessed: 2022-08-19.

[32] "Configuring the retention period for GitHub Actions artifacts and logs in your organization - GitHub AE Docs," https://docs.github.com/en/github-ae@latest/organizations/managing-organization-settings/configuring-the-retention-period-for-github-actions-artifacts-and-logs-in-your-organization, accessed: 2022-08-19.

[33] S. Walfish, "A review of statistical outlier methods," *Pharmaceutical technology*, vol. 30, no. 11, p. 82, 2006.

[34] B. Lopatin, "Automating," in *Django Standalone Apps*. Springer, 2020, pp. 145–150.

[35] T. A. Ghaleb, D. A. Da Costa, and Y. Zou, "An empirical study of the long duration of continuous integration builds," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2102–2139, 2019.

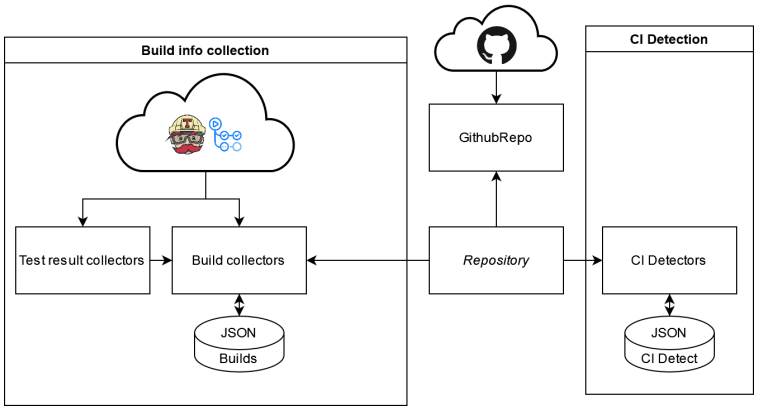

Figure 1: Data collection schematically

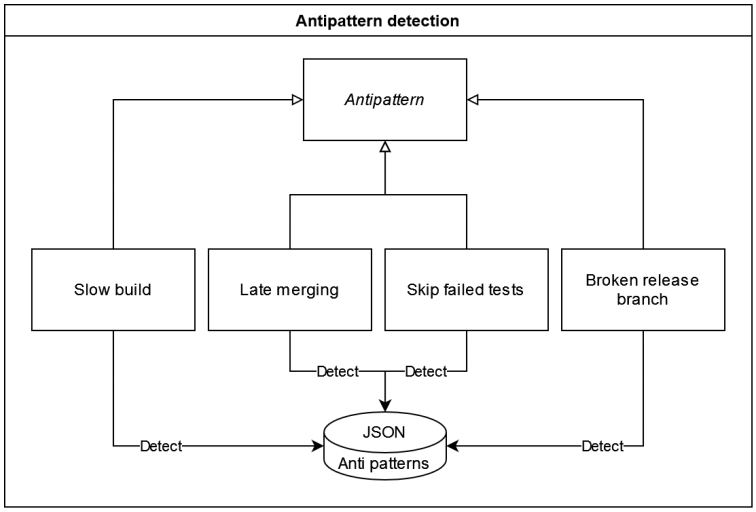

Figure 2: Antipattern detection layer schematically

Appendix

Appendix A - Details on CIAN

This appendix gives additional information on the inner workings of the CIAN tool. Each layer of the tool is explained in greater detail. In addition, examples of output are given and explained.

Data Collection Layer

The data collection layer is responsible for collecting the appropriate build data from the various CI tools and GitHub. Figure 1 schematically pictures how each component in this layer functions. There are build collector components that make a connection with the Application Programming Interfaces (APIs) of GitHub Actions and Travis CI to fetch both the build information and the test log information. To use this data in further layers, it is serialized in a JSON scheme. Furthermore, a Repository component is present to hold git repository information. This object is used together with a CI Detector component to detect which frameworks are implemented.

Antipattern Detection Layer

The second layer of the tool is responsible for extracting antipatterns from the acquired data in the previous layer. Each antipattern has its own dedicated detector component that uses the JSON serialized data from the data collection layer. Figure 2 shows how the layer looks like schematically. The following subsections describe how each antipattern is

detected and how information on them is collected. The way in which this is done is strongly based on how the CI Odor detects information on these antipatterns.

Slow Build

In order to mine information for Slow Builds, the duration of builds is collected from the mined data. The duration of specific jobs in a build, or the amount of jobs in a build is not taken into account, only the total duration matters for this antipattern. In order to represent this data graphically and in order to compare it in a more general way, the average build duration of all builds of one week is taken. In doing so, outliers have less impact on how the data is reported. This is exactly the same approach as used in CI Odor. The only difference is that our tool is capable of looking beyond the last three months and a date range can be set to collect data from.

CI Odor also reports warnings for builds that exceed the usual build duration by a significant amount. A medium severity warning is issued to builds for which the build duration exceeds the third quartile. A high severity warning is issued to builds that have a duration higher than the formula for outlier detection based on the Inter-Quartile Range (IQR) (see equation 1, Q3 represents the third quartile).

$$duration > Q3 + 1.5 \times IQR \tag{1}$$

Our tool reports the same warnings using the same formulas.

Late Merging

The Late Merging antipattern can be refined into a couple sub-metrics. Each of these metrics tells us something about how the repository is managed with respect to the different branches. Branches can be feature branches, hotfix branches or other. CI Odor proposed and implemented 4 submetics: branch age, unsynced activity, branch deviation and missed activity. All submetrics, except branch age are implemented in our tool. CI Odor also generates warnings for each of these metrics in a similar fashion to explained earlier (see 2).

Missed Activity (t_{MA}) This metric quantifies the amount of activity on other branches since the current branch was synced with the main branch. The larger this value is, the more integration effort is required to merge the branch with the mainline. Missed activity is calculated using the formula in equation 2.

$$t_{MA} = t_{LO} - t_{Sync} \tag{2}$$

Where t_{LO} is the date of the latest commit on the other branches, while t_{Sync} is the date of the latest merge commit.

Branch Deviation (t_{BD}) Branch deviation is a component of missed activity. It quantifies the amount of changes in other branches since the last update of the current branch. Branch deviation is calculated using formula 3

$$t_{BD} = t_{LO} - t_{LC} \tag{3}$$

Again, t_{LO} is the last commit date on other branches. t_{LC} on the other hand is the last commit date of the current branch. Note that branch deviation can become negative. This means that this branch is ahead of the others.

Unsynced Activity (t_{UA}) Unsynced activity is a quantification of the amount of activity in the current branch since the last update with the main branch.

$$t_{UA} = t_{LC} - t_{Sync} \tag{4}$$

Broken Release Builds

The tool considers a build broken when its final state is either a failure or an error. When such a build is triggered on the main branch in the git repository, it is a broken release build. By default builds on the branch called 'main' are considered release builds, but this can be changed using a command line option. Similar to the Slow Build antipattern, the average amount of broken release builds is taken for each week to show the evolution better, similar to CI Odor. For this anti-pattern, no medium or high severity warnings are generated by our tool.

Skip Failing Tests

For the tool to be able to detect this antipattern, it requires an analysis of the test logs of the CI builds. A build is considered to have skipped tests if in a build following a failing build, less tests are executed with the sole reason of making the pipeline pass. CI Odor mentioned a formal formula that can be used to identify Skip Failing Tests (see equation 5)

$$\Delta_{breaks} < 0 \wedge (\Delta_{runs} < 0 \vee \Delta_{skipped} > 0) \tag{5}$$

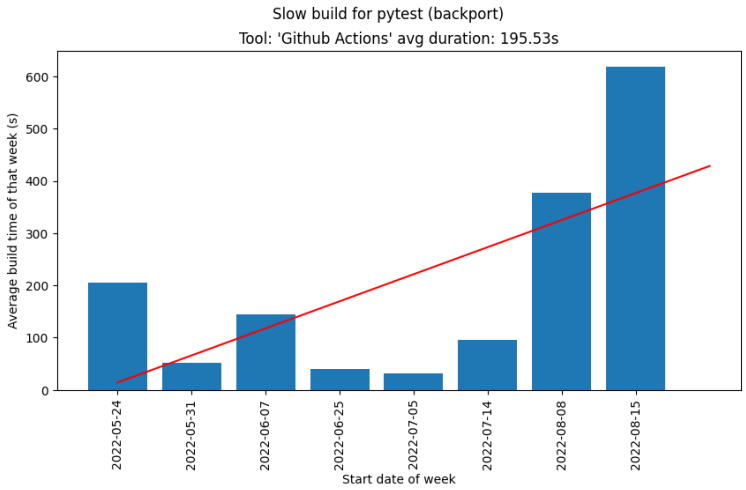

Figure 3: Slow Build output example generated by CIAN for the pytest-dev/pytest repository hosted on GitHub

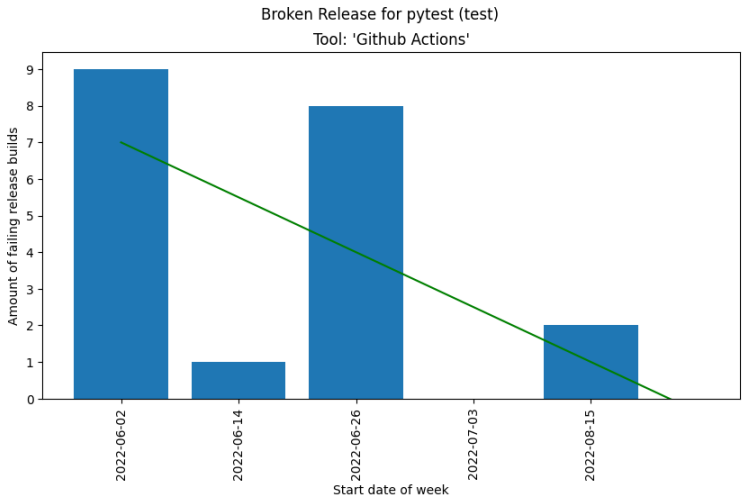

Figure 4: Broken Release output example generated by CIAN for the pytest-dev/pytest repository hosted on GitHub

The deltas represent the change in tests with a particular outcome on the same branch. If the formula yields a true value, the build that followed the broken build is marked as a skip failing tests instance. Our tool reuses this formula to implement Skip Failing Tests in the same way.

Output and Visualization Layer

The final layer is responsible for generating useful output. It takes the antipattern data generated by the previous layer and creates files based on that data. This layer contains two main components: one is responsible for creating graphics while the other is responsible for creating text based files like JSON.

Some anti-patterns allow generating graphs better than others. Figures 3 and 4 show examples of how the tool represents the slow build and broken release anti-patterns in the Pytest repository[1] respectively.

Figure 3 shows the results of the slow build anti-pattern manifested in the Pytest repository. Each bar in the bar chart represents the average build duration of a week. The date that labels the bar is the start date of the week. A linear regression line is plotted on the figure showing the general trend among. When an increasing trend is detected, this line is colored red, while with a decreasing trend, it is colored green. The Pytest example shows 8 weeks of slow build data. An obvious increasing trend can be noticed, indicated by the red line.

[1] https://github.com/pytest-dev/pytest

Figure 4 shows a similar graph for the broken release results. Here, an obvious decreasing trend is present, indicated by the green regression line.

For the Late Merging antipattern it is not possible to generate similar graphs. For this antipattern a text file is generated that contains the results of each submetric (branch deviation, missed activity and unsynced activity).

Finally, the Skip Failing Tests antipattern is also outputted as a text file. In CI Odor, a similar bar chart is presented as with Broken Release or Slow Build. Since the Skip Failing Tests antipattern does not occur much, we decided to not generate graphs as with CI Odor, but to print the results to a text file instead. If required, the tool can be easily extended to generate similar graphs for this antipattern.

Appendix B - Dependencies

Both the proof-of-concept tool CIAN and the analysis are built on a couple of Python libraries. In this section, we will list all the dependencies for both the tool and the empirical analysis.

- PyTest: used for the unit tests of CIAN.
- Flake8: a linter tool for Python projects.
- Requests and aiohttp: libraries used to mine information from the GitHub API and Travis CI API.
- Matplotlib and Seaborn: used to visualize the output of the tool and to create the box plots of the analysis.
- Numpy: used to perform calculations in the tool and empirical analysis.

Furthermore, the tool strongly depends on the GitHub and Travis CI APIs to mine the CI build data. Finally, the empirical analysis is built using a sample of the dataset created by Golzadeh et al. for the paper *"On the rise and fall of CI services in GitHub"*.

Appendix C - Distribution of work

The topic of this thesis was first proposed by professor Tom Mens and Dr. Alexandre Decan of the University of Mons. Initially, they wanted us to develop a tool that was capable of analysing continuous integration in git repositories. By performing research on the topic, I quickly stumbled across the topic of anti-patterns in continuous integration. The advisors and I decided that this was the direction to go for this project.

Most of the programming work for the CIAN tool was performed by me. John provided code snippets that helped me in mining information from the GitHub and Travis APIs. Furthermore, John was always ready to provide feedback and suggestions during our regular meetings.

During an intermediate presentation throughout the year with professor Tom Mens, Dr. Alexandre Decan professor Serge Demeyer and Dr. John Businge, the idea for the empirical analysis of anti-patterns in repositories that substituted CI implementation popped up. From there on, we decided to work towards an analysis on this topic accompanied by a scientific paper that presents the results.

I have written most of the paper with the advice of John. His experience in writing scientific literature significantly helped me to structure the work as good as possible. Especially in critical sections such as the introduction, he provided the resources to write it as best as possible.

Finally, professor Serge Demeyer provided useful feedback on the drafts of this work. He mentioned two larger points that could improve my work besides typos and weird sentence constructions. First, he asked a bit more on the motivation for the choice of experiment. For that, I have included a few sentences that explained why we would collect data from Python and JavaScript repositories in the *CIAN Tool Motivation* section. Second, the professor mentioned that the overview of the popular solution techniques was a bit lacking. I have made this part more explicit in the *CIAN Tool Motivation* section as well.

Appendix D - Relation with Research Projects

Both research project I (Reverse Engineering Google WiFi) and research project II (Spatial Audio in VR - A Comparison of Ambisonic Sound with Regular Sound) have no relation with this project. This project and corresponding research is built from scratch.