GitHub Mining

The Implementation of Continuous Integration Pipelines

Bram de Vries[1]

Supervisor(s): Sebastian Proksch[1]**, Shujun Huang**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 25, 2023

Name of the student: Bram de Vries
Final project course: CSE3000 Research Project
Thesis committee: Sebastian Proksch, Shujun Huang, Fenia Aivaloglou

An electronic version of this thesis is available at http://repository.tudelft.nl/.
Parts of this document were generated using AI tools; see Appendix C

Abstract

While continuous integration has already been proven to positively affect software development, little is known about how it should be implemented based on project context. This paper investigates how CI pipelines are configured by analysing data mined from software projects on GitHub. This research has shown the continued rise of the CI platform GitHub Actions, which enables developers to broaden CI pipelines' functionality due to great integration into GitHub. Moreover, key differences between how jobs within pipelines are structured in Travis CI and GitHub Actions are outlined. These results can be used in future research, which will be aimed at connecting project context to CI setup with the goal of informing developers on maturing their CI configuration.

1 Introduction

Continuous Integration (CI) is a software development practice that involves frequently merging code changes from multiple developers into a single shared repository. It has been shown that CI can improve the productivity of project teams [17] by automating the processes of building, testing, and validating software changes. However, little is known about patterns in the implementation of CI [14] based on contextual factors [6].

Many services are available which offer cloud-hosted continuous integration solutions, with Travis CI[1] and GitHub Actions[2] being the most popular as of May 2021 [12]. While some differences exist between these platforms' pricing and features, they share the core ability to configure CI pipelines in files with the `YAML` format. While these do give some structure to how a pipeline should be set up, a lot of configuration freedom is left to developers.

The paper aims to address the question: "How are Continuous Integration pipelines set up in GitHub software projects?".

The rest of this report is structured as follows. Section 2 presents background on relevant topics and the research questions. Section 3 outlines the used method. Section 4 lays out the results. Section 5 reflects on the responsible aspects of this research. Section 6 discusses the findings. Finally, Section 7 concludes and mentions future work.

2 Background

This section of the paper provides an overview of the relevant literature and previous research on Continuous Integration. It will also describe the research questions and their motivation.

2.1 Related work

In 2000, Fowler and Foemmel introduced the Continuous Integration (CI) concept through their blog post [7]. Their work emphasized ten core CI practices to enhance software development speed and improve overall software quality. Of these practices, a key focus was on implementing fully automated and reproducible builds alongside running tests multiple times a day. Their pioneering insights have significantly influenced the adoption and implementation of CI methodologies, shaping the modern software development and delivery landscape.

Since the conception of CI, much research has been done that shows the positive effects of adapting CI in software projects. Valescu et al. [17] study the impact of CI on team productivity and software quality in projects on GitHub. They discovered that teams that use CI are more effective at merging pull requests while at the same time finding more bugs, thus improving the quality of the software.

Hilton et al. [13] researched CI's usage, costs and benefits by analyzing open-source projects on GitHub and surveying developers. They examined metrics like *time for CI adoption* and *number of changes to CI configs*, which will also be interesting for this research. They agreed that CI helps developers integrate pull requests more quickly and catch bugs earlier.

Golzadeh et al. [12] have demonstrated a significant shift in the usage of CI services. Since its publication in 2019, GitHub Actions has quickly become the most prevalent platform, overtaking all established services in 18 months. They also outline the co-usage and migration between multiple CI platforms.

Durieux et al. [5] performed an analysis of more than 35 million jobs of Travis CI. They also investigated the metrics *time for CI adoption* and *number of changes to CI configs*, as well as *the number of jobs per usage category*. It has been discovered that the programming languages in projects that employ Travis CI are not equally distributed to those used on GitHub and that most jobs are used for testing.

Chen et al. [3] performed a similar analysis of repositories that use GitHub Actions. They show that the average *number of configured workflows* is 2.8, most jobs use servers provided by GitHub and Ubuntu is the most used platform.

Gautam et al. [8] performed a cluster analysis of projects that use Travis CI. They were able to identify clusters based on repository metrics like *activity*, *popularity*, *size*, *testing*, and *stability*. These clusters showed distinct characteristics, with which they hope to advise new developers about which software project to join.

2.2 Continuous Integration

Continuous Integration is a development practice where developers regularly integrate their code changes into a shared repository. The integrated code is then automatically built, tested, and validated to identify any issues or conflicts early in the development process.

A CI *pipeline* is a sequence of automated *jobs*. The jobs in a pipeline typically include code compilation, unit testing, integration testing, code analysis, and deployment. It allows for the automation of various tasks required to validate and deliver software changes. They can be triggered by a number of *events*, like a push to a branch or the creation of a pull request.

[1]http://travis-ci.org

[2]http://github.com/features/actions

A concept unique to GitHub Actions is that of *workflows*, which are independently configured pipelines that can perform different sets of tasks. This allows for more control over when certain jobs are triggered. It could also reduce code duplication because they can be reused and referenced from other workflows.

2.3 Research Questions

Much of the prior research that includes empirical analysis on CI, as outlined above, was done on a single CI service. This can be attributed to the following three factors:

Popularity of Travis CI. Travis CI was the most popular continuous integration service for GitHub [2] [12] since its introduction in 2011, being used by significantly more repositories than every other service. It has also initiated the creation of *TravisTorrent*, a "*freely available data set based on Travis CI and GitHub that provides easy access to hundreds of thousands of analysed builds from more than 1,000 projects*" [1]. The popularity of Travis CI, together with this readily available data set, makes it a great candidate for performing research.

Newness of GitHub Actions. In comparison to other CI services, GitHub Actions is relatively young. It was introduced in 2019 but quickly became the most popular platform used on GitHub. Due to this, not much historical data is available on GitHub Actions.

Difference between CI services. While the base for every service is a `YAML` configuration file, there can be some significant differences in how they should be structured. For example, Travis CI relies on structured phases like *install* and *script*, while GitHub Actions is entirely built around *jobs* and *steps*. This can make it difficult to compare the two.

This paper aims to analyse the implementation of CI across multiple platforms. A more comprehensive overview of the current state of CI in software projects can be obtained by extending previous research to encompass a broader range of platforms. In this analysis, the type of CI service used will be considered one of the metrics rather than the primary focus. This approach allows for a broader perspective on CI practices.

When this is achieved, a future goal is to combine the results with those of other members of the GitHub Mining Research Project group. Together we are looking into different aspects of software project repositories and mining metrics about them. Like Gautam [8], the goal is to categorise repositories based on metrics like project activity, maturity, topic, build life cycle and CI state. This will be done to ultimately be able to inform and advise developers on maturing the CI implementation based on the context of their project.

The analysis will be structured around five research questions (RQ) to accomplish this. Based on these results, the state of CI will be discussed. These questions are as follows:

RQ 1: When are CI pipelines introduced into a project? Projects do not have CI configured by default. Looking at the time it takes projects to start using CI could give insight into adoption patterns and may be linked to project maturity.

RQ 2: How are jobs triggered in CI pipelines? Different rules can be set for when to trigger a pipeline, based on different events. Exploring this could reveal common use cases.

RQ 3: How are jobs structured in CI pipelines? CI pipelines are configured in `YAML` files, which offer only a loose structure to how different jobs should be set up. Having a deeper understanding of how CI is structured can result in better advice for developers on how to configure their CI.

RQ 4: What distinct types of jobs are set up in CI pipelines? A CI pipeline comprises different jobs, all having separate tasks. The two most common uses for CI are building and testing[5], but does this differ between platforms?

RQ 5: Which operating systems are used for CI pipelines? CI services offer multiple platforms to execute pipelines on. Examining how this feature is used can provide insight into cross-platform development strategies and may be linked to performance.

3 Methodology

This section presents the methodology employed to mine data from GitHub repositories. The process involved three key phases: repository curation, data mining and data analysis. Splitting the process into these different steps allowed them to be executed independently from each other, storing intermediate data or results. Using the GitHub API[3], relevant pieces of information were collected about the presence and configuration of CI pipelines within repositories. Subsequently, data analysis techniques were applied to extract meaningful patterns, identify trends and draw conclusions regarding CI pipeline configurations and practices.

3.1 Repository Curation

Selecting repositories for analysing their CI usage is critical in conducting a comprehensive and representative study. Besides the base criterion of needing to have either GitHub Actions or Travis CI configured in a repository, the following criteria and considerations were used to identify and include repositories in the analysis:

- **CI usage** The first and most important criterion for selecting repositories is whether they use Continuous Integration. While more than 20 different CI platforms and tools are available [12], I have determined to limit this research to only analysing repositories that employ either Travis CI or GitHub Actions. As of May 2021, these cover 90.9% of all repositories that use CI [12]. By concentrating on these two widely adopted platforms, the analysis significantly reduces the workload associated with examining many different CI configurations.
- **Programming language** While the primary focus of this research is not on the programming language of software projects, it is still essential to consider the language aspect during the repository selection process. Using a tool called *GitHut 2.0*[4], the top ten most-used languages

[3] https://docs.github.com/en/rest

[4] https://madnight.github.io/githut/#/pull_requests/2023/1

in quarter 1 of 2023 were identified, together spanning almost 85% of all pull requests. Repositories with these languages should be present in the final dataset to ensure a representative overview of software projects on GitHub. Appendix A shows the ten considered languages and their share in PRs.

- **Active development** To ensure the currency and relevance of the selected repositories, only repositories were selected that had a commit within the last six months, between December 2022 and June 2023. The analysis incorporates actively maintained projects likely to reflect the most current CI practices by focusing on repositories with at least one commit within this timeframe. This approach helps to avoid outdated or abandoned repositories that might not accurately represent the prevailing trends and advancements in CI adoption. By including repositories with recent activity, the analysis provides a more accurate and timely snapshot of the CI landscape, enabling valuable insights into developers' contemporary practices and configurations.
- **Number of stars** To find potential repositories, the SEART GitHub Search Engine[5] was used. Researchers can utilise this tool to select repositories for empirical studies based on various selection criteria combinations. One design choice of this tool is that it only mines projects with at least ten stars. While the project's creators noted that stars are not a good proxy for the quality or relevance of repositories, this criterion allowed the tool to be more scalable and excluded projects that are unlikely to be relevant for empirical studies [4].
- **Non-forked projects** Projects that are forks are excluded to reduce redundancy and ensure consistency. Forked repositories are copies of the original repository that may have undergone modifications or diverged from the original codebase [10]. The analysis maintains a more accurate representation of CI practices within individual projects by focusing solely on the original repositories.

Using SEART GitHub Search Engine, multiple searches were performed to find repositories that use CI for each of the ten languages identified above. In addition to the language, last commit date, number of stars, and exclusion of forks, a final search query was added. While the search engine does not have the functionality to determine whether a repository uses CI directly, it allows filtering based on user labels added to issues in a repository. By adding the search for a label ci, the number of search results decreased significantly while the chance of them including CI increased. This was useful as it was infeasible to check all resulting repositories for the employment of CI. Adding this final filter decreased the total number of potential repositories from roughly $250,000$ to 1815.

To check whether a repository uses CI, they need to be checked for the existence of a configuration file. The criteria for finding Travis CI and GitHub Actions will be discussed in Section 3.2, Workflow Config Module. After this check, a list of 1640 repositories was curated. This means that just over 90% of the repositories that use a ci label utilised either CI platform. This is the same coverage mentioned earlier [12], although it should be noted the other 10% were not checked for the existence of any other platform.

Some final manual filtering was done on the dataset based on outlying results achieved during the data analysis phase. The number of jobs and configuration files per repository was analysed, as discussed in Section 4.4, GitHub Actions. Four repositories had significantly more configuration files and jobs than all other projects. These repositories either contained a large number of code examples (*azure-samples/java-on-azure-examples*[6], *optuna/optuna-examples*[7]) or contained a large number of sub-projects or tools with individual pipelines set up in the parent repository (*dogtagpki/pki*[8], *litmuschaos/test-tools*[9]). In both cases, they do not represent the type of software project this research focuses on, so they were removed.

3.2 Data Mining

The data mining phase involved retrieving data from GitHub repositories, focusing on specific metadata that provides insights into the presence and usage of CI pipelines.

The software for this phase was written in collaboration with four other teammates. Our collective goal was to make versatile software to accommodate all our different needs. We came up with a structure where various modules could mine different metrics. For instance, the *Commits module* is responsible for extracting all commits-related data, like the contents of commit messages or the total number of commits for a repository. A main central runner was developed where the modules could be toggled on or of. This allowed us to work with the same software, configured per our specialised needs.

The three modules that were relevant to my research are the following:

Repository Module. This module was used to extract information about a repository as a whole. The metric I use is a repositories creation date, which is used for answering RQ 1.

Workflow Config Module. This module was used to identify and store any Continuous Integration configuration $.yaml$ files. Where these files are stored and how they should be called differs between GitHub Actions and Travis CI. GitHub workflow files should be stored in the $.github/workflows$ directory and can have any name [11], while Travis CI files should be stored in the root directory and have to be called $.travis.yml$ [16]. The repository is searched for both options, and any files matching the conditions will be downloaded and saved for future analysis.

Commits Module. This module was initially created to mine data about every commit made to a repository. However, this did not benefit my research as I am only interested in commits that add or change the Continuous Integration

[5]https://seart-ghs.si.usi.ch

[6]https://github.com/azure-samples/java-on-azure-examples

[7]https://github.com/optuna/optuna-examples

[8]https://github.com/dogtagpki/pki

[9]https://github.com/litmuschaos/test-tools

configuration files. The module was adapted to receive an optional file path parameter, resulting in only commits being processed that change that particular file.

3.3 Data Analysis.

Once all the data has been collected, the next step is to analyse it thoroughly. This was done by creating three new analysis classes and a visualiser class.

Repository Analyser. The first and relatively most straightforward analysis class was the repository analyser. It was developed to examine each repositories metadata. The main task of this class was to deduce when CI was introduced into each repository, which results are used in RQ1.

YML Analyser. The second analysis class was developed to inspect the contents of the workflow configuration files. Once initialised, it reads all *.yml* files into to be investigated further.

Travis Analyser. The last class was a hybrid miner and analyser class. Each repository that uses Travis CI has access to an online portal to configure some settings further. This includes setting when pipelines should be triggered, which RQ 2 aims to answer. This class collects that data through the Travis CI API[10] and aggregates it.

Visualiser. Visualising is the final step after all data is gathered, aggregated and stored. This is done by the Visualiser class, which uses the Python library Matplotlib[11] to create all the graphs used for this paper.

4 Results

The following section presents and discusses the results obtained from the analysis for each research question. Appendix B shows an overview of all extracted metrics and for which research question they were used.

4.1 Usage of CI Platform

While it is not included as one of the five research questions, examining how many repositories use Travis CI, GitHub Actions, or a combination of both is useful. Out of the 1637 mined repositories, 1574 use GitHub Actions, and the other 63 have configurations for both. This means none of them exclusively use Travis CI.

4.2 RQ 1: When are CI pipelines introduced into a project?

The analysis of RQ1, focusing on when CI pipelines are introduced into projects, reveals compelling findings regarding the timeline of CI adoption. To visualise the results, two dates had to be extracted: the creation date of the repository and the creation date of the CI configuration file. Based on these two dates, the number of days between repository creation and the first CI configuration creation, the Time To Introduction ($\mathcal{TTI}$), was computed.

When aggregated by platform, it was revealed that the $\mathcal{TTI}$ for Travis CI was considerably lower than that of GitHub Actions. Travis CI was introduced into repositories on average after 612 days (median = 281), while GitHub Actions took 1272 days (median = 1018). However, these platforms do not have the same age. Considering the first potential moment a platform's configuration could have been introduced, the $\mathcal{TTI}$ were normalised. While not as large as before, there is still a difference between the two platforms. These results can be seen in Figure 1.

Figure 2 displays the normalised $\mathcal{TTI}$ against the creation date of their respective repositories. The blue and red coloured dots indicate Travis CI and GitHub Actions configuration files, respectively. It can be observed that since Travis CI's introduction in 2011, many repositories have been quick to add it to their projects. For repositories created before GitHub Actions' introduction, it takes anywhere between 2 and 6 years to add to their project, while newer ones tend to add it shortly after creation. The steep slope that can be seen on the right side indicates the maximal moment CI could have been added to the repositories. A project created 30 days ago can only configure CI within those 30 days, causing this clear line.

Some data points returned a negative $\mathcal{TTI}$, meaning that the commit that introduced a CI configuration file predates the creation of the repository. This could happen when existing projects, including their git history, are ported to a new repository. These points are excluded in Figures 1 or 2, as the $\mathcal{TTI}$ cannot be determined.

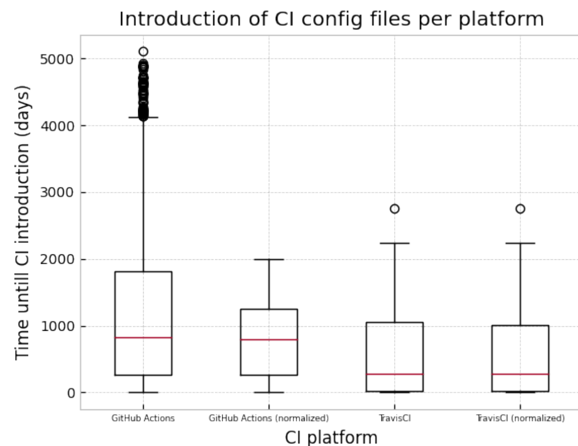

Figure 1: Average number of days between creating repositories and introducing CI configuration files per platform. Normalised values consider the first potential moment a platform's CI configuration could have been introduced into a repository.

RQ 1: When are CI pipelines introduced into a project?
An overall trend was found that newer projects introduced CI pipelines more quickly after their creation. On average, Travis CI is introduced more quickly than GitHub Actions. Moreover, projects created before the existence of GitHub Actions take between 2 and 6 years to still adopt it.

[10]https://developer.travis-ci.com/
[11]https://matplotlib.org/

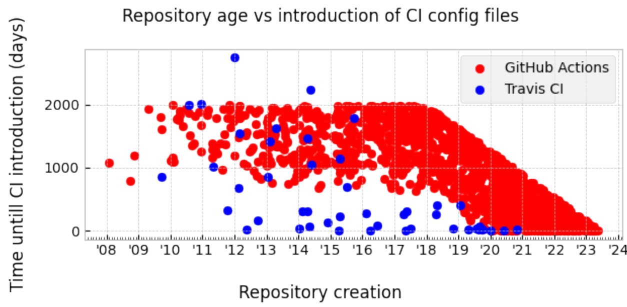

Figure 2: Repository creation vs the number of days until CI is introduced into a project.

4.3 RQ 2: How are jobs triggered in CI pipelines?

For this research question, the trigger setup of CI pipelines was analysed. Some differences exist between the available triggers and how they can be configured between Travis CI and GitHub Actions.

Travis CI

Travis CI will only trigger every push made to a repository by default. This can be extended to also happen on the creation of pull requests. This configuration is not set inside the $.travis.yml$ configuration file but in the online dashboard. These triggers can be refined by adding a **blocklist** or **safelist** to the **branches** attribute inside the configuration file.

As can be seen in Table 1, it is common for Travis CI pipelines to be triggered on both *pushes* and *pull requests*. Some repositories have turned off both push and pull request triggers, which disables Travis CI altogether for that project.

		Push			
		on		*off*	
Pull request	*on*	28	(77.8%)	2	(5.6%)
	off	2	(5.6%)	4	(11.1%)

Table 1: Number of times combination of the *push* and *pull request* triggers are used in Travis CI pipelines.

GitHub Actions

Unlike Travis CI, GitHub offers many configuration options for triggering pipelines. This configuration is done inside the $.github/workflows/*.yml$ files, under the **on** attribute.

Figure 3 displays that *push* and *pull_request* are most commonly used, in 61% and 59% of all workflow, respectively. Besides that, two common triggers are *schedule* and *workflow_dispatch*, both only available in GitHub Actions. Schedule allows the pipeline to be triggered at a scheduled time or interval, based on a POSIX Cron Syntax string[12]. Workflow dispatch makes pipelines available to be manually triggered using the GitHub API, GitHub CLI, or GitHub browser interface [9]. All other available options listed in the GitHub Documentation[13] were found at least once.

Because GitHub offers many more options for setting triggers, a further investigation had to be done to examine how they are used together. Figure 4 shows that the combination of *pushes* and *pull requests* is most common among GitHub Actions pipelines. Moreover, *schedule* and *workflow_dispatch* are more often used together with *push* and *pull_request* than the other way around.

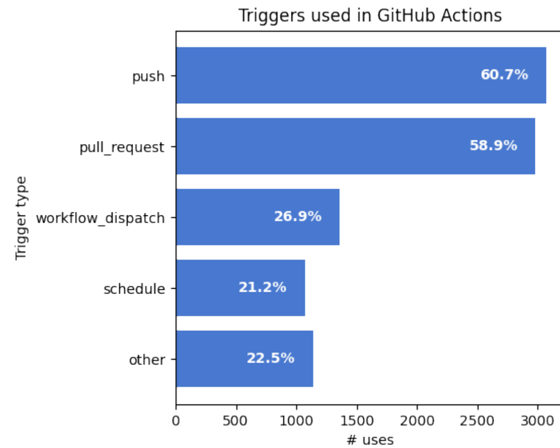

Figure 3: Number of times triggers are used in GitHub Actions pipelines. Percentages are relative to the total amount of pipelines.

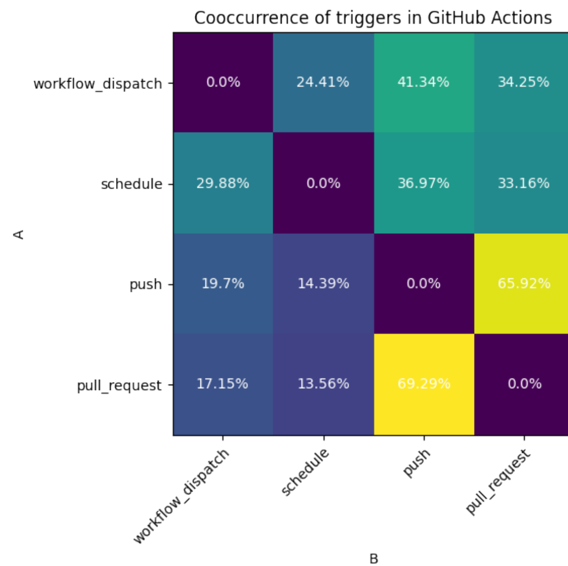

Figure 4: Proportion of workflows using both trigger types A and B relative to the total amount of usages of A.

[12] https://pubs.opengroup.org/onlinepubs/9699919799/utilities/crontab.html#tag_20_25_07

[13] https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows

RQ 2: How are jobs triggered in CI pipelines? While the method of setting up triggers differs between Travis CI and GitHub Actions, the most common configuration is a combination of *push* and *pull request*. Besides, GitHub offers two more commonly used trigger types: *schedule* and *workflow_dispatch*.

4.4 RQ 3: How are jobs structured in CI pipelines?

This research question examines how the jobs are set up in CI pipelines. There is a difference in how they are defined between Travis CI and GitHub Actions so they will be discussed separately.

Travis CI

Travis CI offers multiple methods of configuring a pipeline. Because all configuration needs to be done in a single file per project, it can be hard for developers to split it into different jobs with separate tasks. Upon manual inspection of several $.travis.yml$ files, I have identified three main methods of how jobs get configured:

1. **A single script.** The primary way that Travis CI presents its configuration is through the use of several phases in a job lifecycle. These are defined as root properties inside configuration files. The two main phases are *install* and *script*, which install dependencies and run the build script, respectively. This method can be identified by having a `script` property present only in the root of the configuration file.
2. **A job matrix.** A second method that is also presented in the Travis CI documentation is the use of the `jobs` or `matrix` properties. These allow developers to define multiple jobs that will be run in parallel [15]. They can be identified by having the `script` property inside at least one of the jobs listed in the `include` list.
3. **A combination.** The last method combines both, in which the clauses to identify the first two methods both hold.

Table 2 shows the division between the detected methods based on the abovementioned criteria. There were two configuration files that neither had a script tag nor a job matrix, making it seem like they were not functional. The table lists each configuration method's average and median lines of code (LOC). It is shown that the average loc is significantly higher when a job matrix is implemented.

Method	#	%	Config loc avg	Config loc median
Single script	39	78%	33.54	24
Job matrix	6	12%	75.17	60
Combination	3	6%	169.34	168
Neither	2	4%	7.0	7

Table 2: Distribution of job configuration methods used in Travis CI.

GitHub Actions

GitHub allows developers to separate workflow configuration files containing multiple jobs, making it easy to define workflows for different triggers, environments or releases. This allows for numerous ways of setting up a CI environment, from a single file with many jobs to multiple files with a small number of jobs.

First, the number of configuration files per repository was investigated. On average, projects contained 3.65 configuration files (median = 3). Figure 5 shows several outliers, with one repository even configuring 45 individual workflow files. Upon manual inspection of the three repositories with the most workflow files, *stdlib-js/stdlib*[14], *alibaba/bladedisc*[15] and *realm/realm-dotnet*[16], a common trend can be seen. These are large multi-platform projects with CI pipelines configured for many operating systems or language versions.

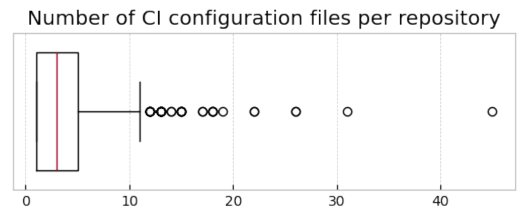

Figure 5: Number of GitHub Actions workflow configuration files per repository.

Next, the analysis focused on the number of jobs per repository, as shown in Figure 6. On average, projects contained 6.13 jobs between all their configuration files (median = 4). There again are several outliers, but they are not entirely the same as those in the number of configuration files discussed before.

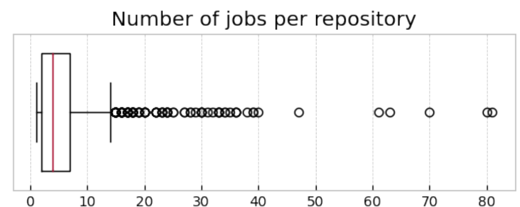

Figure 6: Number of GitHub Actions jobs configured between all workflow files per repository.

Figure 7 plots the total number of jobs per repository against the number of workflow configuration files. It also includes a trendline which shows that, in general, repositories that have more configuration files will also have more total jobs configured.

[14]https://github.com/stdlib-js/stdlib
[15]https://github.com/alibaba/bladedisc
[16]https://github.com/realm/realm-dotnet

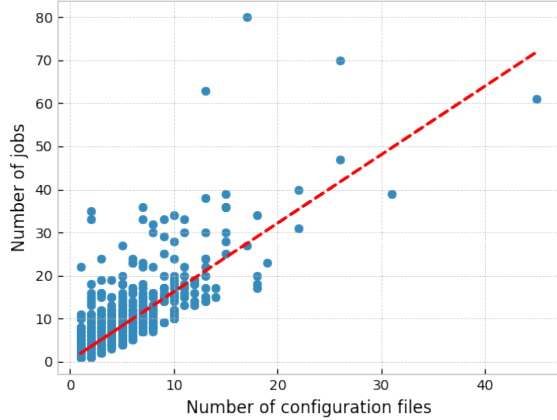

Figure 7: Total number of jobs in a repository vs number of configuration files in a repository.

RQ 3: How are jobs structured in CI pipelines? Travis CI allows for two methods of configuring jobs, through a single script or a job matrix. The former option is most prevalent, but the latter or a combination of both does happen, although resulting in a larger configuration file size on average. Most GitHub Actions projects configure three workflow files and six jobs within their repositories. Projects that set up considerably more workflow configuration files tend to have more jobs configured too.

4.5 RQ 4: What distinct types of jobs are set up in CI pipelines?

This question researches the different kinds of jobs. An in-depth analysis was carried out on Travis CI jobs in a study by Durieux et al. which identified eight distinct categories of these jobs [5]. The categorisation was done through an automated process that relied on specific keywords to determine the class of each job.

This process was repeated for the jobs in our mined GitHub Actions workflows. Due to its tight integration into GitHub, a ninth *organisation* category could be identified. It encompasses any jobs interacting with other aspects of GitHub, like automatically labelling pull requests or closing issues. The keywords used for automatic categorisation in the previous research were not disclosed, so these had to be reidentified.

Table 3 shows this categorisation's results and includes the Travis CI findings from [5]. Compared to Travis CI, there seems to be a bigger focus on jobs besides *testing* and *building* in GitHub Actions.

RQ 4: What distinct types of jobs are set up in CI pipelines? Using previous large-scale research on jobs in Travis CI, 9 different job categories were devised. Comparing the outcomes of that study to the results of categorising GitHub Actions jobs, there seems to be a stronger focus on jobs outside of *building* and *testing* in GitHub Actions.

Category	# Jobs	GHA %	TCI %
Testing	2040	20.56%	58.64%
Building	1883	18.98%	8.30%
Analyzing	1605	16.17%	0.18%
Releasing	817	8.23%	1.43%
Organisation	716	7.22%	-
Documentation	401	4.04%	3.26%
Formatting	69	0.70%	1.82%
Communication	63	0.63%	0.07%
Unknown	2329	23.47%	26.26%

Table 3: Distribution of categorized jobs in GitHub Actions and Travis CI.

4.6 RQ 5: Which operating systems are used for CI pipelines?

The previous research question focussed on the various jobs used in different pipelines. This final research question will analyse what operating systems (OS) are used to run these jobs. Travis CI and GitHub Actions allow developers to specify which operating system to run a pipeline on. This can be especially useful for multi-platform software that must be built, tested, and validated for all platforms.

There are two key differences between the configuration of an OS between Travis CI and GitHub Actions. First, where Travis CI only offers the options *Ubuntu*, *MacOS*, and *Windows*, GitHub also allows their workflows to be run on self-hosted runners. Second, the OS is configured on a *pipeline level* in Travis CI, while it is on a *job level* in GitHub Actions.

The analysis results of used operating systems can be seen in Figure 8 and Table 4. It stands out that no OS is specified for 62% of Travis CI pipelines. It should be noted that for both CI services, Ubuntu will be used when no operating system is specified. This total Ubuntu usage is listed as the last row in the table. It can be concluded that Ubuntu is the prevailing operating system used by both services.

OS	GHA Jobs #	GHA Jobs %	TCI Pipelines #	TCI Pipelines %
Ubuntu	6386	73.7%	16	29.1%
MacOS	216	2.5%	4	7.3%
Windows	242	2.8%	1	1.8%
Not specified	717	8.3%	34	61.8%
Other	1102	12.7%	-	-
Ubuntu total	7003	82.0%	50	90.9%

Table 4: Distribution of operating systems used in GitHub Actions jobs and Travis CI pipelines. The last row shows the total usage of Ubuntu, as that is the default when no system is specified.

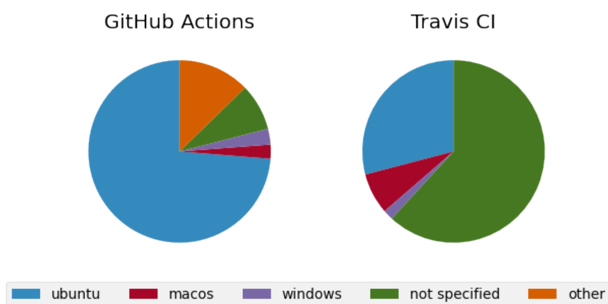

Figure 8: Distribution of operating systems used in GitHub Actions jobs and Travis CI pipelines.

RQ 5: Which operating systems are used for CI pipelines? Although GitHub offers a lot more freedom in customising which OS to run pipelines on, the majority (totalling 80%) will use the same main options that Travis CI offers: *Ubuntu*, *MacOS* and *Windows*. About 62% of Travis CI pipelines have no OS configured, defaulting to *Ubuntu*.

5 Responsible Research

Ensuring that research results are reproducible and independently verifiable is crucial. To make this possible, the code created for this study has been made accessible to the public on TU Delft's servers and GitHub via https://github.com/raduConstantinescu/Descriptive-CI-Metrics. The code includes documentation on each component, a description of how to install and start running it, and lists the API keys required to start mining data.

All the collected data has been made publically available through 4TU Research Data[17]. It includes the complete raw data that the miner collected and processed intermediate results used to create this paper's tables and figures.

All data used for this paper was open-source and publically available, mainly gathered through either the GitHub API or Travis CI API. When actively mining repositories still in development, the data collected reflects their current state at the extraction time. Essentially, the information obtained is a snapshot that captures the repositories' progress up to that point. This can threaten reproducibility, as trying to mine that same repository later does not guarantee the return of the same data. To combat this, either this paper or the dataset will include the data's retrieval date.

6 Discussion

This research focussed on the two most used CI platforms. Combining established results with new insights can paint a picture of the current CI landscape. The extracted metrics cover a wide range of important aspects of CI pipelines. The described methodology and results should create a strong foundation for further research. As described in Section 2.3, this research is part of an overarching goal to inform the configuration of CI based on contextual factors of software projects.

As already indicated by Golzadeh et al., there has been a shift to GitHub Actions[12], which can be primarily attributed to Travis CI's pricing and policy changes in November 2020. This was further confirmed by the dataset curated for this research, which found 1574 repositories that use GitHub Actions, 63 that use Travis CI and GitHub Actions and none that exclusively use Travis CI. While this relatively low number of Travis CI repositories could be a limiting factor for the results of this research, the repository curation process was chosen to be as general and unbiased as possible.

An overall trend was that newer projects introduced CI pipelines more quickly after their creation. On average, Travis CI is introduced earlier than GitHub Actions. Repositories created before the introduction of GitHub Actions in 2018 took multiple years to adapt to the new platform. This may be because they already utilise a different CI service, or developers may find introducing it into established software difficult.

Throughout Travis CI and GitHub Actions, it is most common to trigger pipelines on both pushed and pull requests. These seem to be the two most critical moments where checks could be beneficial. Interestingly, for some Travis CI workflows, both those triggers are disabled, meaning the pipeline will never be run. This is probably a low-effort means of disabling Travis CI because GitHub Actions is also used in all those projects.

Perhaps the most remarkable result is the number of configuration files and jobs some repositories have set up. I hypothesised that repositories would use a couple of configuration files with a few or many jobs each or many configuration files with a few jobs each. While there were a lot of the former, none of the latter were found. Additionally, the number of repositories with many configuration files and many jobs was surprising. Future research could look into how these jobs are used and whether there is room for optimisation regarding duplication.

When comparing the types of jobs that are set up in GitHub Actions to Travis CI pipelines, there seems to be less focus on *building* and *testing* and more on *analysis* and *organisation*. The GitHub Marketplace[18] offers preconfigured jobs which can be reused inside workflows. Even slightly more actions are available for *code quality analysis* than *testing*, which could explain the notable differences within those categories.

There are some limitations with how the GitHub Actions jobs are categorised. First, the exact keywords used in classifying Travis CI[5] were not disclosed, so they had to be recreated, which was done with the help of GitHub Copilot[19] (see Appendix C). Second, the keywords are only matched against the user-configured titles of jobs. A more in-depth approach

[17] https://data.4tu.nl

[18] https://github.com/marketplace?type=actions

[19] https://github.com/features/copilot

could lead to a more accurate categorisation.

There is a significant difference between the percentage of *Travis CI pipelines* and the percentage of *GitHub Actions jobs* with no OS specified. While in Travis CI, the OS is determined on a pipeline level, in GitHub Actions, they get set on a job level. Developers who create a GitHub pipeline will likely be more aware of configuring it properly, resulting in greater consistency.

7 Conclusions and Future Work

This paper describes a method for mining data about software projects hosted on GitHub to analyse the configuration and state of their CI. Together with a team of peers, we worked to develop software that mines various metrics for a given list of repositories. These metrics can be customised to suit specific use cases.

The results show a wide variety of methods to set up CI pipelines. While the `YAML` files and services give some structure, a lot is left up to decide by the developers. The relatively new CI platform GitHub Actions is on the rise, which is more tightly integrated into GitHub. This is shifting the focus from mainly on building and testing to including code analysis and automating organisational tasks.

In the future, these results can be combined with those of my team of peers to find a relation between a repository's contextual factors and the configuration of its CI. To be able to advise developers on how CI could be matured for their projects, more research needs to be done on what effect different configurations have on performance and efficiency. There is also room for improvement regarding code duplication inside CI configuration files. Some repositories use many separate files for similar tasks, like testing on multiple operating systems. More research could determine whether this is a flaw of the CI services or developers and how this could be resolved.

References

[1] Moritz Beller, Georgios Gousios, and Andy Zaidman. Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 447–450, 2017.

[2] Moritz Beller and Joseph Hejderup. Blockchain-based software engineering. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 53–56, 2019.

[3] Tingting Chen, Yang Zhang, Shu Chen, Tao Wang, and Yiwen 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)*, pages 01–10, 2021.

[4] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. Sampling projects in github for MSR studies. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*, pages 560–564. IEEE, 2021.

[5] Thomas Durieux, Rui Abreu, Martin Monperrus, Tegawendé F. Bissyandé, and Luís Cruz. An analysis of 35+ million jobs of travis ci. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 291–295, 2019.

[6] Omar Elazhary, Colin M. Werner, Ze Shi Li, Derek Lowlind, Neil A. Ernst, and Margaret-anne D. Storey. Uncovering the benefits and challenges of continuous integration practices. *CoRR*, abs/2103.04251, 2021.

[7] Martin Fowler. Continuous integration, Sep 2000. https://martinfowler.com/articles/continuousIntegration.html.

[8] Aakash Gautam, Saket Vishwasrao, and Francisco Servant. An empirical study of activity, popularity, size, testing, and stability in continuous integration. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 495–498, 2017.

[9] GitHub. Events that trigger workflows. GitHub Docs, 5 June 2023. https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows.

[10] GitHub. Fork a repo. GitHub Docs, 5 June 2023. https://docs.github.com/en/get-started/quickstart/fork-a-repo.

[11] GitHub. Workflow syntax for github actions. GitHub Docs, 5 June 2023. https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions.

[12] Mehdi Golzadeh, Alexandre Decan, and Tom Mens. On the rise and fall of ci services in github. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 662–672, 2022.

[13] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, page 426–437, New York, NY, USA, 2016. Association for Computing Machinery.

[14] Puneet Kaur Sidhu, Gunter Mussbacher, and Shane McIntosh. Reuse (or lack thereof) in travis ci specifications: An empirical study of ci phases and commands. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 524–533, 2019.

[15] Travis CI. Build matrix. Travis CI Documentation, 5 June 2023. https://docs.travis-ci.com/user/build-matrix.

[16] Travis CI. Customizing the build. Travis CI Documentation, 5 June 2023. https://docs.travis-ci.com/user/customizing-the-build.

[17] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 805–816. Association for Computing Machinery, 2015.

A Distribution of languages

#	Language	%
1	Python	$17,207\%$
2	Java	$11,314\%$
3	Go	$10,423\%$
4	C++	$10,082\%$
5	JavaScript	$9,553\%$
6	TypeScript	$7,899\%$
7	PHP	$5,307\%$
8	Ruby	$5,012\%$
9	C	$4,407\%$
10	C#	$3,141\%$
	Total	$84,345\%$

Table 5: Top 10 most used languages in GitHub pull requests as of Q1 of 2023. Lists the percentage of usage in those PRs. Data collected from https://madnight.github.io/githut/#/pull_requests/2023/1

.

B Extracted metrics

	Metric	Additional information
Repository	Name	
	Creation date	
	CI platform(s)	
	CI config files: $\mathcal{CF}$	all configuration files found in repository
	$\mathcal{TTI}$	# days between repository creation and first CI addition
	# config files	*GitHub Actions only*
	# jobs	*GitHub Actions only*
	Travis CI settings: $\mathcal{TCI}$	*Travis CI only*
$\mathcal{CF}$	CI platform	
	Filename	
	Creation date	Inferred from associated commits
	Commits: $\mathcal{AC}$	All commits that include config file
	Contents: $\mathcal{CFC}$	
$\mathcal{AC}$	Message	
	Date	
	SHA	
	File status	added, modified, renamed, deleted
$\mathcal{CFC}$	Triggers	
	Trigger cooccurrence	
	Job configuration method	*Travis CI only*
	Jobs: $\mathcal{J}$	
	OS	*Travis CI only*
$\mathcal{J}$	Category	*GitHub Actions only*
	OS	*GitHub Actions only*
$\mathcal{TCI}$	Trigger on push	
	Trigger on pull request	

Table 6: A breakdown of extracted data related to the continuous integration pipelines employed in the analysed repositories. The metrics are organised into multiple nested levels, indicated by matching abbreviations.

C Usage of Large Language Models

ChatGPT. The role of ChatGPT in this report was strictly limited to providing guidance and assistance in structuring paragraphs or sentences. However, it is important to note that ChatGPT generated no content-related ideas or specific research insights.

GitHub Copilot. Using GitHub Copilot, some keywords were generated for categorising jobs in GitHub Actions workflows.