



UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale in Ingegneria del Software 2

*Analysis of usage of Continuous
Integration practices in open-source
projects*

Anno Accademico 2018/2019

relatore

Ch.mo prof. Porfirio Tramontana

candidato

Ewelina Jablonska

matr. M63000829

Contents

1	Background	3
1.1	GitHub	3
1.2	Continuous integration	5
1.2.1	Tools	5
1.3	GitHub actions	6
1.3.1	Workflows	6
1.3.2	Limits	8
1.4	YAML	9
1.4.1	Processes	10
1.4.2	Block styles	10
1.4.3	Flow styles	11
1.4.4	Node properties	11
1.4.5	Character streams	12
1.4.6	Workflow GitHub	13
2	Related work	18
2.1	Information sources	18
2.2	Mining Github	19
2.3	Continuous integration	26
2.3.1	General usage	26
2.3.2	Bad Practices	30
3	Technology	34
3.1	Node.js	34

3.1.1	Advantages of Node.js	35
3.1.2	NPM	35
3.1.3	Used libraries	36
3.2	GitHub Api	37
3.2.1	Authentication	38
3.2.2	Pagination	38
3.2.3	Search	39
3.2.4	Rate limiting	39
3.2.5	Example	40
4	Preliminar study	41
4.1	Research questions	41
4.2	Objects	42
4.3	Variables	42
4.4	Experimental procedure	43
4.5	Results	48
4.6	Threats to validity	49
4.6.1	Internal	49
4.6.2	External	50
5	Empirical study	51
5.1	Research questions	51
5.2	Objects	52
5.3	Variables	52
5.4	Experimental procedure	54
5.5	Results	59
5.6	Threats to validity	78
5.6.1	External	78
6	User manual	80
6.1	Installation	80
6.2	Prerequisites	81

6.3	Execution	82
7	Final discussion	85
8	Conclusions and future directions	89
8.1	Conclusions	89
8.2	Future directions	90

Introduction

Open-source software projects are the epitome of collaboration. They represent the amalgamation of the work and effort of hundreds or thousands of developers coming together to achieve a single purpose: to create an application that fulfills user needs. However, there is a point where such a large workforce becomes too difficult to manage.

Social coding sites like GitHub have started offering solutions, such as contribution guidelines and continuous integration (CI) tools, to get core developers and contributors on the same page and help unify expectations. Because of the benefits of using CI tools, GitHub now offers a native, fully integrated CI solution. Due to the role CI plays in evaluating code contributions on GitHub, developers have started considering CI among their contribution evaluation criteria.

This work aims to analyze the usage of Continuous Integration practice, in particular in open-source projects, and to study the diffusion of bad practices related to the use of this practice.

Continuous Integration (CI) is a set of software development practices that allow software development teams to generate software builds more quickly and periodically. Thanks to the advent of the Continuous Integration practices builds can be generated more frequently, which allows the earlier identification of errors.

Recently, researchers have begun to study CI empirically to understand its associated costs and benefits. Indeed, Continuous Integration has been claimed to introduce several benefits in software development, including high software

quality and reliability. However, recent work pointed out challenges, barriers, and bad practices characterizing its adoption.

This work investigates the general usage of CI practice in open-source projects, but also how the bad practices are widespread while applying CI. In summary, previous work discussed the advantages of CI, outlined possible bad practices, and identified barriers and challenges in CI's usage. However, prior work particularly focuses on the usage of Travis CI tool, whilst, to the best of our knowledge, there is no prior investigation about the usage of Github actions CI tool.

Chapter 1

Background

1.1 GitHub

GitHub is the largest collaborative source code hosting site built on top of the Git version control system. It represents the newest generation of software forges, which are web-based collaborative platforms providing tools to ease distributed development, especially useful for Open Source Software (OSS) development. GitHub uses a "fork & pull" collaboration model, where developers create their own copies of a repository and submit requests when they want the project maintainer to incorporate their changes into the project's main branch. Every repository can optionally use GitHub's issue tracking system to report and discuss bugs and other concerns.

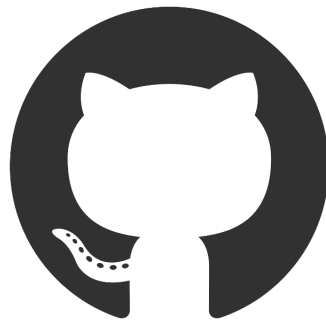


Figure 1.1: Github logo

Moreover, GitHub contains integrated social features, such as the opportunity for an user to subscribe to update by "watching" projects and "following" other users. Therefore, GitHub has become both the target of choice and the source of data for various research efforts, ranging from distributed collaboration to deep learning on software data. The typical pull request development model, as used by GitHub, is a newer method for collaborating in distributed software development. With this model, the project's main repository is not writable by potential contributors. Instead, the contributors fork (clone) the repository and make their changes independent of each other. Due to this popular development model, repositories can be divided into two types: base repositories (ones that are not forks) and forked repositories. Each GitHub pull request contains a branch (local or in another repository) from which a core team member should pull commits. GitHub automatically discovers the commits to be merged and attaches them to the pull request. By default, pull requests are submitted to the destination repository for review.

Also, Git repositories are an important source of empirical software engineering product and process data. Running the Git command-line tool and processing its output with other Unix tools allows the incremental construction of sophisticated data processing pipelines. The use of Unix command tools to analyze Git repositories offers interactivity, readability, performance, scalability, and portability. Git data analytics on the command-line can be systematically presented through a pattern that involves fetching, selection, processing, summarization, and reporting. The git command-line tool has a notoriously complex command-line interface. However, using just a few key commands allows researchers to access source code across revisions without requiring expensive copies to disk. After extracting the data with git commands, we can apply traditional Unix tools to perform relational algebra operations, without needing to store the data in an intermediate SQL database. Then, a big part of empirical work in software engineering is ex-

ploratory, when researchers try to understand the data, perform basic hypothesis testing, or draw simple plots of raw data. The final step of all research efforts is reporting the results.

1.2 Continuous integration

Continuous integration (CI) is a software engineering practice typically implemented as the automation and frequent execution of the build and test steps to make software integration easier. Continuous Integration has been introduced by Fowler in a blog post in 2000, as means for the systematic integration and verification of code changes. CI practices have gained momentum only in the last 10 years, being more widely discussed, employed, and researched. Consequently, CI is nowadays one of the pillars of the software engineering practice, not only in commercial projects but also in open source projects. The adoption of CI has been shown multiple benefits for software engineering practices related to build, test, and dependency management. Thanks to the advent of the CI practices, for example, builds can be generated more frequently, which allows the earlier identification of errors.

1.2.1 Tools

There are several tools offering support for developers that plan to incorporate the CI practices into their software projects, such as TravisCI, CircleCI, and Hudson. More interestingly, however, is the fact that some of these tools are readily available in social coding environments such as GitHub and GitLab, which implies that everyone with a GitHub account can gratuitously benefit from the complex pipeline of version control systems, code review systems, and continuous integration tools, with little to no configuration effort. In particular, Travis CI is an open-source, distributed, CI tool that supports more than 25 programming languages. Every

GitHub repository can be configured to use Travis CI to automatically generate CI builds. Projects that use the TravisCI service inform TravisCI about how jobs are to be executed using a `.travis.yml` specification file. The properties set in this configuration file specify which revisions will initiate builds, how the build environments are to be configured for executing builds, and how team members should be notified about the outcome of the build.

1.3 GitHub actions

GitHub Actions enables you to create custom software development life cycle (SDLC) workflows and to build end-to-end continuous integration (CI) and continuous deployment (CD) capabilities directly in your GitHub repository. [1] GitHub Actions powers GitHub's built-in continuous integration service, and help you automate your software development workflows in the same place you store code and collaborate on pull requests and issues. You can write individual tasks, called actions, and combine them to create a custom workflow.



GitHub Actions

Figure 1.2: Github actions logo

1.3.1 Workflows

Workflows are custom automated processes that you can set up in your repository to build, test, package, release, or deploy any code project on GitHub. They can run on GitHub-hosted virtual machines, or on machines that you host yourself.

You can configure your CI workflow to run when a GitHub event occurs (for example, when new code is pushed to your repository), on a set schedule, or when an external event occurs using the repository dispatch webhook. GitHub provides preconfigured workflow templates to automate your workflow or create a CI workflow for specific languages and frameworks. GitHub analyzes your code and shows you the CI templates that are the best fit for your repository. Moreover, you can browse and search for actions in GitHub Marketplace to use in your workflows. GitHub Marketplace is a central location for you to find actions created by the GitHub community. Actions with a badge indicate GitHub has verified the creator of the action as a partner organization. You can discover new actions from the workflow editor on GitHub and the GitHub Marketplace page.

Workflows need to be stored in the `.github/workflows` directory in the root of your repository, and they must have at least one job, containing a set of steps that perform individual tasks, and able to run commands or use an action. The workflows must be configured using YAML syntax, and be saved as workflow files in your repository. Once a YAML workflow file is successfully created and triggered the workflow, you will see the build logs, test results, artifacts, and statuses for each step of your workflow. At a high level, these are the steps to add a workflow file:

1. At the root of your repository, create a directory named `.github/workflows` to store your workflow files.
2. In `.github/workflows`, add a `.yml` or `.yaml` file for your workflow. For example, `.github/workflows/continuous-integration-workflow.yml`.
3. Use the "Workflow syntax for GitHub Actions" reference documentation to choose events to trigger an action, add actions, and customize your workflow.
4. Commit your changes in the workflow file to the branch where you want

your workflow to run.

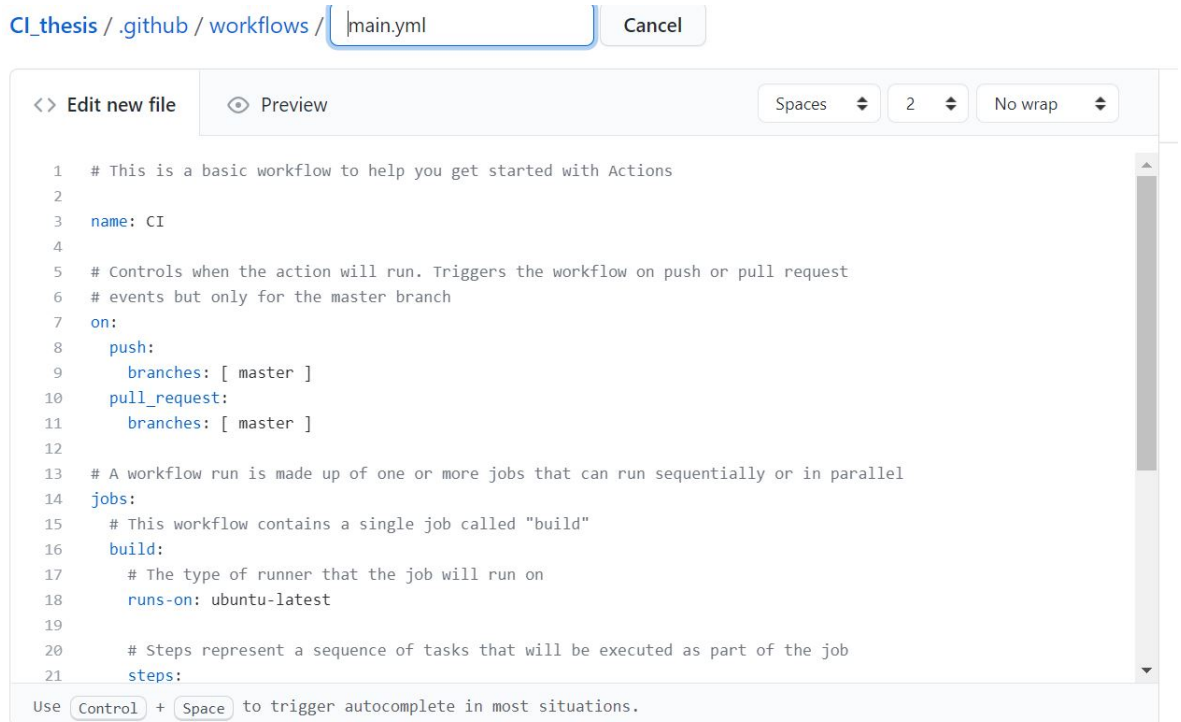


Figure 1.3: An example of github actions workflow

1.3.2 Limits

There are some limits on GitHub Actions usage, depending on whether you use GitHub-hosted or self-hosted runners, and they are subject to change:

Job execution time : Each job in a workflow can run for up to 6 hours of execution time. If a job reaches this limit, the job is terminated and fails to complete. This limit does not apply to self-hosted runners.

Workflow run time : Each workflow run is limited to 72 hours. If a workflow run reaches this limit, the workflow run is canceled. This limit also applies to self-hosted runners.

Job queue time : Each job for self-hosted runners can be queued for a maximum of 24 hours. If a self-hosted runner does not start executing the job within

this limit, the job is terminated and fails to complete. This limit does not apply to GitHub-hosted runners.

API requests : You can execute up to 1000 API requests in an hour across all actions within a repository. If exceeded, additional API calls will fail, which might cause jobs to fail. This limit also applies to self-hosted runners.

Concurrent jobs : The number of concurrent jobs you can run in your account depends on your GitHub plan. If exceeded, any additional jobs are queued. There are no concurrency limits for self-hosted runners.

Job matrix : A job matrix can generate a maximum of 256 jobs per workflow run. This limit also applies to self-hosted runners.

1.4 YAML

YAML is a human-friendly, Unicode-based data serialization language designed around the common native data types of agile programming languages. It is broadly useful for programming needs ranging from configuration files to Internet messaging to object persistence to data auditing.

YAML was first proposed by Clark Evans in 2001, who designed it together with Ingy döt Net and Oren Ben-Kiki. Originally YAML was said to mean *Yet Another Markup Language*, referencing its purpose as a markup language with the yet another construct, but it was then repurposed as *YAML Ain't Markup Language*, a recursive acronym, to distinguish its purpose as data-oriented, rather than document markup. [4] The official recommended filename extension for YAML files has been `.yaml` since 2006. The syntax of YAML was motivated by Internet Mail (RFC0822) and remains partially compatible with that standard. Further, borrowing from MIME (RFC2045), YAML's top-level production is a stream of independent documents, ideal for message-based distributed processing systems.

YAML is both a text format and a method for presenting any native data structure in this format.

1.4.1 Processes

YAML information is used in two ways: for machine processing, and human consumption. The challenge of reconciling these two perspectives is best done in three distinct translation stages: representation, serialization, and presentation. Representation addresses how YAML views native data structures. Serialization concerns itself with turning a YAML representation into a serial form, that is, a form with sequential access constraints. The presentation deals with the formatting of a YAML serialization as a series of characters in a human-friendly manner. There are myriad flavors of data structures, but they can all be represented with three basic primitives:

1. **Mappings** (hashes/dictionaries)
2. **Sequences** (arrays/lists)
3. **Scalars** (strings/numbers)

While most programming languages can use YAML for data serialization, YAML excels in working with those languages that are fundamentally built around the three basic primitives. These include the wave of agile languages such as Perl, Python, PHP, Ruby, and Javascript.

1.4.2 Block styles

In YAML block styles, the structure is determined by *indentation*, in general, defined as zero or more space characters. A block style construct is terminated when encountering a line that is less indented than the construct. Each node must be indented further than its parent node, and all sibling nodes must use the same

indentation level at the start of a line. Block sequences indicate each entry with a dash and space ("- "), while mappings use a colon and space (": ") to mark each key: value pair.

1.4.3 Flow styles

YAML also has *flow styles*, using explicit indicators rather than indentation to denote scope. The flow sequence is written as a comma-separated list within square brackets. Similarly, flow mapping uses curly braces. YAML's flow styles can be thought of as the natural extension of JSON to cover folding long content lines for readability, tagging nodes to control the construction of native data structures, and using anchors and aliases to reuse constructed object instances.

1.4.4 Node properties

Each presentation node includes two major characteristics called anchor and tag. The anchor property represents a node for future reference. The character stream of YAML representation in a node is denoted with the ampersand "&" indicator. The tag property represents the type of native data structure which defines a node completely, and it's represented with the "!" indicator. Subsequent occurrences of a previously serialized node are presented as alias nodes. The first occurrence of the node must be marked by an anchor to allow subsequent occurrences to be presented as alias nodes. An alias node is denoted by the "*" indicator and refers to the most recent preceding node having the same anchor.

Scalar content can be written in block notation, using a literal style (indicated by "|") where all line breaks are significant. Alternatively, they can be written with the folded style (denoted by ">") where each line break is folded to space unless it ends an empty or a more-indented line.

1.4.5 Character streams

To ensure readability, YAML streams use only the printable subset of the Unicode character set. In YAML, you come across various *character streams* as follows:

- Directives
- Document Boundary Markers
- Documents
- Complete Stream

Directives are basic instructions used in YAML processor, and they are the presentation details like comments which are not reflected in serialization tree. Reserved directives are initialized with three hyphen characters (—). Concerning Document Boundary Markers, YAML uses these markers to allow more than one document to be contained in one stream. These markers are specially used to convey the structure of YAML document. Note that a line beginning with “—” is used to start a new document. YAML document is considered as a single native data structure presented as a single root node. The presentation details in YAML document such as directives, comments, indentation, and styles are not considered as contents included in them. There are two types of documents used in YAML:

Explicit Documents : It begins with the document start marker followed by the presentation of the root node. It includes an explicit start and end markers which is “—” and “...”

Implicit Documents : These documents do not begin with a document start marker.

Finally, YAML includes a sequence of bytes called a character stream. The stream begins with a prefix containing a byte order denoting a character encoding.

The complete stream begins with a prefix containing a character encoding, followed by comments.

1.4.6 Workflow GitHub

Workflow Github files use YAML syntax and must have either a `.yaml` or `.yml` file extension. The file is characterized by the following elements:

name : The name of the workflow. GitHub displays the names of the workflows on the repository's actions page. If omitted, GitHub sets it to the workflow file path relative to the root of the repository.

on : The name of the GitHub event that triggers the workflow (Required). You can provide a single event string, array of events, array of event types, or an event configuration map that schedules a workflow or restricts the execution of a workflow to specific files, tags, or branch changes.

- **types**: Selects the types of activity that will trigger a workflow run. Most GitHub events are triggered by more than one type of activity. For example, the event for the release resource is triggered when a release is published, unpublished, created, edited, deleted, or released. The types of keywords enable you to narrow down activity that causes the workflow to run. When only one activity type triggers a webhook event, the types of keyword is unnecessary.
- **branches & tags**: When using the push and pull_request events, you can configure a workflow to run on specific branches or tags. For a pull_request event, only branches and tags on the base are evaluated. If you define only tags or only branches, the workflow won't run for events affecting the undefined Git ref.

- **paths:** When using the push and pull_request events, you can configure a workflow to run when at least one file does not match paths-ignore or at least one modified file matches the configured paths. Path filters are not evaluated for pushes to tags.

env : A map of environment variables that are available to all jobs and steps in the workflow. You can also set environment variables that are only available to a job or step.

defaults : A map of default settings that will apply to all jobs in the workflow. You can also set default settings that are only available to a job.

- **defaults.run:** You can provide default *shell* and *working-directory* options for all run steps in a workflow. You can also set default settings for run that are only available to a job.

jobs : A workflow run is made up of one or more jobs. Jobs run in parallel by default.

- **id:** Each job must have an id to associate with the job. The key job_id is a string and its value is a map of the job's configuration data. You must replace <job_id> with a string that is unique to the jobs object. The <job_id> must start with a letter or _ and contain only alphanumeric characters, , or -. If you need to find the unique identifier of a job running in a workflow run, you can use the GitHub API.
- **name:** The name of the job displayed on GitHub.
- **needs** : To run jobs sequentially, you can define dependencies on other jobs using the jobs.<job_id>.needs keyword.
- **runs-on:** Each job runs in an environment specified by runs-on, that

is required and represents the type of machine to run the job on. The machine can be either a GitHub-hosted runner or a self-hosted runner.

- **outputs:** A map of outputs for a job. Job outputs are available to all downstream jobs that depend on this job. Job outputs are strings, and job outputs containing expressions are evaluated on the runner at the end of each job. Outputs containing secrets are redacted on the runner and not sent to GitHub Actions.
- **env:** A map of environment variables that are available to all steps in the job.
- **defaults:** A map of default settings that will apply to all steps in the job.
- **if:** You can use the if conditional to prevent a job from running unless a condition is met. You can use any supported context and expression to create a conditional.
- **steps:** A job contains a sequence of tasks called steps. Steps can run commands, run setup tasks, or run an action in your repository, a public repository, or an action published in a Docker registry. Not all steps run actions, but all actions run as a step. Each step runs in its own process in the runner environment and has access to the workspace and filesystem. Because steps run in their own process, changes to environment variables are not preserved between steps. GitHub provides built-in steps to set up and complete a job.

id : A unique identifier for the step. You can use the id to reference the step in contexts.

if : You can use the if conditional to prevent a step from running unless a condition is met. You can use any supported context and expression to create a conditional.

name : A name for your step to display on GitHub.

uses : Selects an action to run as part of a step in your job. An action is a reusable unit of code. You can use an action defined in the same repository as the workflow, a public repository, or in a published Docker container image. It's strongly recommended to include the version of the used action by specifying a Git ref, SHA, or Docker tag number. If a version isn't specified, it could break the workflow or cause unexpected behavior when the action owner publishes an update.

with : A map of the input parameters defined by the action. Each input parameter is a key/value pair. Input parameters are set as environment variables. The variable is prefixed with `INPUT_` and converted to upper case. Review the action's README file to determine the inputs required. Actions are either JavaScript files or Docker containers. If the action you're using is a Docker container you must run the job in a Linux environment.

run : Runs command-line programs using the operating system's shell. If you do not provide a name, the step name will default to the text specified in the run command. Commands run using non-login shells by default. You can choose a different shell and customize the shell used to run commands.

shell : You can override the default shell settings in the runner's operating system using the shell keyword. You can use built-in shell keywords, or you can define a custom set of shell options.

env : Sets environment variables for steps to use in the runner environment. You can also set environment variables for the entire workflow or a job.

- **strategy:** A strategy creates a build matrix for your jobs. You can define different variations of an environment to run each job in.
- **container:** A container to run any steps in a job that doesn't already specify a container. If you have steps that use both script and container actions, the container actions will run as sibling containers on the same network with the same volume mounts. If you do not set a container, all steps will run directly on the host specified by runs-on unless a step refers to an action configured to run in a container.
- **services:** Used to host service containers for a job in a workflow. Service containers are useful for creating databases or cache services like Redis. The runner automatically creates a Docker network and manages the life cycle of the service containers.

Chapter 2

Related work

We group our related work into three different areas:

1. Information sources
2. Mining Github
3. Continuous Integration

2.1 Information sources

Due to the abundance and availability of data, code hosting services, such as GitHub, have piqued the interest of many software engineering researchers. The public availability of data from many projects simplifies the data collection and processing issues that are often encountered in research. However, there are still practical difficulties that can potentially alter conclusions drawn from the data.

The availability of a comprehensive API has made GitHub a target for many software engineering and online collaboration research efforts. Some research shows that obtaining data from GitHub is not trivial and that the data may not be suitable for all types of research. First of all, GitHub imposes limits on their API, that can put a significant delay on data acquisition. Moreover, there is no

data schema, since GitHub is only exposing its data as JSON responses through a REST API. Several research projects have provided easier access to the data available through the GitHub API.

The **GHTorrent** project [9] provides a mirror of the GitHub data, which it obtains by monitoring and recording GitHub events as they occur, and applying recursive dependency-based retrieval of the related resources. GHTorrent follows the GitHub event stream and systematically retrieves from it all data, their metadata, and their dependencies. It then processes and stores all retrieved items in a relational database, while also storing the original data in a MongoDB database. GHTorrent offers to interested researchers both downloads of the corresponding database dumps, and online access facilities, including live database access and Google BigQuery. In January 2020, MongoDB stores around 18TB of JSON data (compressed), while MySQL more than 6.5 billion rows of extracted metadata. A large part of the activity of 2012-2019 has been retrieved, while researchers are also going back to retrieve the full recorded history of important projects. GHTorrent has been very successful: indeed, more than 200 researchers have subscribed and used the online access points, so we can say that GHTorrent is becoming the de facto standard dataset for large scale quantitative analysis for GitHub data.

2.2 Mining Github

The large amount of public data on GitHub and its availability via an API make it possible for researchers to easily mine project data. In the last years, a considerable amount of research papers have been published reporting findings based on data mined from GitHub.

A Systematic Mapping Study of Software Development With GitHub [6] conducting a meta-analysis of 342 papers reporting results based on GitHub data mining, aimed at understanding how software development practices have changed

due to the popularization of social coding platforms like GitHub, and how project owners, committers and end-users could optimize their way of collaborating. The main goal is to provide an overview of research efforts focusing on the analysis of all kinds of software development practices. The study addresses the following research questions:

- What topics/areas have been addressed?
- What empirical methods have been used?
- What technologies have been used to extract and build datasets from GitHub?
- What is the research community behind these works like?

The main findings reported by the selected works of the study can be grouped into four main areas of research, focused on:

1. Development
2. Projects
3. Users
4. GitHub ecosystem

Regarding **software development**, and in particular the code contributions, the study found that contributions on the platform are unevenly distributed and that mostly few developers are responsible for a large set of code contributions (more than two-thirds of projects have only one committer). Also, most contributions come in the form of direct code modifications.

Moreover, most pull requests are less than 20 lines long, processed (merged or discarded) in less than 1 day, and the discussion spans on average to 3 comments. The study claims the importance of casual contributions, stating that casual contributions are rather common in GitHub (48.98%).

Furthermore, according to the research, there are several factors to accept or reject pull requests, ranging from technical to social and geographic factors. First, pull requests fully addressing the issue they are trying to solve, self-contained, and well documented, as well as including test cases and efficiently implemented are more likely to be accepted. Conversely, pull requests containing unconventional code are less likely to go through. Second, social factors also influence the acceptance of pull requests. Indeed, contributions from submitters with prior connections to core members, having a stronger social connection or holding a higher status in the project are more likely to be accepted, whereas a pull request from an external collaborator has 13% less chance of being accepted. Concerning the geographical factors, the research shows that when submitters and integrators are from the same geographical location there are 19% more chances that the pull requests will get accepted.

Also, the latency of processing pull requests has been discussed in different works, finding that the increase in the time needed to analyze a pull request reduces the chances of its acceptance.

Furthermore, several findings of the study concern the issue topic. It founds that issues tracker are scarcely used and that issues are unevenly distributed on the projects that actively use the GitHub issue tracker. The number of open issues is on average higher right after the project creation, while it tends to decrease a few months later. Moreover, even though GitHub offers a label mechanism to ease the categorization of issues, only less than 30% of issues are tagged. Furthermore, the older an issue is, the smaller is the chance that it will be addressed.

Another topic the study deals with is forking. According to the findings, the distribution of forks follows a power-law distribution, meaning that there are lots of projects with few forks, and few projects forked a very large number of times. Moreover, the number of forks of a project is positively correlated with the number of open issues, watchers, and stars, as well as with the number of commits and

branches, but it does not correlate with the number of pull requests accepted. The study found that forks are mostly used to fix bugs and add new features, and less frequently to add documentation. Also, forking is considered positive for several reasons such as to preserve abandoned programs and to improve the quality of a project. The chances to get a project forked depends on different factors: a project where developers provide additional public contact information that is clearly active or with popular project's owners are more probable to be forked.

With regard to **projects**, the study shows that most projects are personal and little more than code dumps such as example code, experimentation, backups, or exercises not intended for customer consumption, and most of them ignore Github collaboration capabilities. Also, GitHub is not only for open source software, but commercial projects use GitHub and the functionalities provided by the platform as well. The top three domains for the projects are system software, web, and non-web libraries, followed by software tools, whereas JavaScript, Ruby, Python, Objective-C, and Java are the top used languages in terms of the number of projects in GitHub. Also, projects that are actively maintained are more likely to be alive in the future than projects that only show occasional commits. The study focuses also on the project's popularity factor, claiming that popularity is useful to attract new developers. Nevertheless, only a few projects are popular (in particular the distribution of stars and downloads follows a power-law distribution).

Moreover, projects owned by organizations tend to be more popular than the ones owned by individuals. According to the study, there exists a relationship between the popularity and the documentation effort. In particular, popular projects exhibit higher and more consistent documentation activities and efforts. Regarding teams, most of the projects have small development teams (72% of projects have only a single contributor). Furthermore, the study claims that there exists a positive impact on the collaboration experience and productivity in diverse teams.

With regard to **users** in Github, most of them come from North America, Europe, and Asia, and a large majority of them are males, while females account only for a small percentage of GitHub. Approximately half of GitHub's registered users work in private projects, thus their activities on the platform are not publicly visible. User productivity depends on factors such as the number of projects the user is involved in and her commitment to the project. A rockstar, or popular user, is characterized by having a large number of followers, who are interested in how she codes, what projects she is following, or working on. Users become popular as they write more code and monitor more projects.

Finally, regarding the **ecosystem**, defined as a collection of software projects, most ecosystems in GitHub revolve around one central project, and most of them are interconnected. However, activities around a software development project are not exclusively performed in GitHub, but they leverage on other platforms and channels with superior capabilities in terms of social functions: Twitter is used by developers to stay aware of industry changes, for learning, and for building relationships, while StackOverflow is often used as a communication channel for project dissemination.

The previous work [11] aimed at examining how GitHub is used for collaboration through surveys and interviews. The study used the GHTorrent dataset and identified **thirteen perils** that pose potential threats to validity for studies involving software projects hosted in GitHub. They can be divided into 4 main categories:

1. Projects related
2. Pull request related
3. User related

4. Github related

With regard to **projects**, the research found that a repository is not necessarily a project (a project is typically part of a network of repositories). Moreover, most projects have low activity, for example, they have very few commits, and most of them are inactive. Also, the study shows that a large portion of projects is not used for software development activities and that most of them are personal. Indeed, more than two-thirds of projects (71.6 % of repositories) have only one committer: its owner. Finally, many active projects do not conduct all their software development activities in GitHub, and only a fraction of projects use pull requests.

Concerning **pull requests**, merges only track successful code (if the commits in a pull request are reworked in response to comments, GitHub records only the commits that are the result of the peer review, not the original commits), and many merged pull requests appear as non-merged. The **user** related perils concern the fact that not all activity is due to registered users, and only the user's public activity is visible. Nevertheless, approximately half of GitHub's registered users do not work in public repositories.

In conclusion, the study has shown that the **GitHub** API exposes either a subset of events or all data entities and that GitHub continues to evolve and it has changed some features and provided new ones. Similarly, the projects evolve and are capable of changing their own history. Finally, one project can be subject to more than one perils.

The social coding site GitHub provides developers with many management tools to facilitate project maintenance and developer collaboration. **Milestone tool**, in particular, plays an important role in organizing and tracking progress on groups of issues and pull requests in a project. Milestone tool acts as a container for

issues and pull requests, corresponding to specific features or project phases. Using this tool, developers can easily file bugs that need to be fixed before launching the beta of the project. Also, developers can easily file issues that they would like to work on to focus their efforts or file issues related to redesigning the project to collect ideas.

A recent research [14] has investigated the use of the milestone tool in GitHub open-source projects, to find its benefits and limits. First, some data were collected by using the GHTorrent and GitHub API V3, which yielded 184 464 projects. The study asks the following questions:

- How many projects used the milestone tool in their development history?
- Do certain types of projects tend to use milestone tool more than others?
- What is the relationship between the usage of milestone tool and project outcomes?
- Are the milestone setting details associated with the milestone completion time?

The findings are that nearly 20% of GitHub analyzed projects used milestone tool in their development history, which means that many projects did not use the milestone tool, although it is automatically set up for them. Also, since the project creation, 73.5% of the projects need more than one month to create their first milestone.

Moreover, there is no obvious correlation between the project programming language and the milestone tool's usage, but projects that have been created for a long time tend to use the milestone tool, depending largely on the size of the tasks that need to be processed. Besides, projects that are small, unsuccessful, or developed by small teams, may not use the milestone tool to manage issues and pull requests, because they may not use the GitHub issue tracking system or

not have too many issues or pull requests. Also, projects with milestone tend to release more than projects without milestone, and are associated with more stars.

According to the research, the most common reason for developers using the milestone tool is that it helps developers focus their attention and efforts on important things, and many developers want to use milestone tool to communicate to users, which indicates that the current milestone tool has benefits in terms of work efficiency, visibility, and collaboration. On the other side, the most common reasons for developers not using the milestone tool are that their projects don't have many issue data and they are not familiar with milestone tool, which has limitations in terms of operability, functionality, and maintenance. When using the milestone tool, developers would like the milestone tool to be both powerful and easy to manage and maintain. This can cause some tension, since adding feature/support tends to increase complexity and simplification may reduce functionality.

2.3 Continuous integration

2.3.1 General usage

The impact of CI on the software development process is a topic of active research.

Research published in 2016 studies the usage of CI in open-source projects, analyzing 34,544 open-source projects from GitHub. [10] They answer several research questions grouped into three themes:

1. Usage of CI
2. Costs of CI
3. Benefits of CI

In particular, we are interested in the questions belonging to the first category, that are:

RQ1: What percentage of open-source projects use CI?

RQ2: What is the breakdown of usage of different CI services?

RQ3: Do certain types of projects use CI more than others?

RQ4: When did open-source projects adopt CI?

RQ5: Do developers plan on continuing to use CI?

The study found that 40% of all the projects in breadth corpus use CI. With regard to services, according to the research Travis CI is, by far, the most widely used CI service. Because of this result, the study's further analysis focus on the projects that use Travis CI as a CI service. The research shows that most popular projects (as measured by the number of stars) are also the most likely to use CI. The answer to RQ4 is that the median time for CI adoption is one year. Finally, the study predicts that in the future CI adoption rates will increase even further.

A recent study [5], in particular, addresses the impact of CI on a paradigmatic socio-technical activity within the software engineering domain, namely **code reviews**. Basically, it aimed at seeing how pull request review discussions changed after the introduction of CI, by using a dataset of code reviews (pull request discussions) from a sample of 685 Github open-source projects that adopted Travis-CI as their sole CI service at some point in their history. The notion of code inspections to improve software quality was introduced by Fagan in 1976, who describes a process of line-by-line inspection of source code during team-wide meeting, requiring a significant amount of human effort.

The study addresses the following research questions:

- How does the amount of communication during pull request reviews change with the introduction of CI?
- How does the amount of updates to pull requests after they have been opened

change with the introduction of CI?

Overall, the study shows that on average and while controlling for other variables such as pull request updates after creation, there is a less general discussion of pull requests overtime after adopting CI.

Also, a previously increasing trend in the number of line-level review comments made during pull request reviews is, on average, reversed after CI adoption, just like the trend in the number of change-inducing review comments.

Finally, the outcome of the pull request review process, measured in terms of subsequent code changes made during a pull request review, does not change on average after adopting CI. The apparent hand-off gives rises to the idea of continuous integration as a silent helper, where some of the tasks traditionally executed by human reviewers are now carried out by CI, leading to fewer discussions in code reviews.

Recent research has shown that a considerable amount of development time is invested in optimizing the generation of builds. An empirical study of the long duration of continuous integration builds [8] has investigated which factors are associated with the **long duration of CI builds**, using 67 GitHub projects that are linked with Travis CI. The research questions of the study are the following:

- What is the frequency of long build durations?
- What are the most important factors to model long build durations?
- What is the relationship between long build durations and the most important factors?

The findings are that over 40% of the considered builds took over 30 minutes to run and that only 16% of the builds had durations under 10 min. According to the research, build durations have a strong association with CI build factors, such as

the build triggering time, the number of times to rerun failing commands, caching, and finishing as soon as the required jobs finish. Indeed, findings Configuring CI builds to finish as soon as the required jobs finish is most likely to be associated with short build durations. Moreover, caching content that does not change often has a strong inverse association with long build durations. Finally, maintaining a stable build status has also a strong association with long build durations, but with a negligible reduction in the build failure ratio.

A recent work [12] aimed at **improving the robustness and efficiency** of Continuous Integration and Deployment, detecting some problems related to CI services, such as misconfiguration of CI environments, misinterpretation of CI results, and inefficient use of CI resources. A typical CI service has different nodes for creating build jobs, processing them, and reporting on the outcome. Nevertheless, configuring job processing nodes is complex. The study addresses the following research questions:

- How are features in CI/CD environments being used?
- How are features in CI/CD environments being misused?
- To what extent are noise and heterogeneity present in off-the-shelf CI/CD outcome data?
- Is CI/CD queuing time a problem? If so, how can it be improved?
- Can CI/CD be accelerated without relying on the build dependency graph?

The study found that 48% of the studied TravisCI specification code is instead associated with configuring job processing nodes. This shows that the developers rarely use CI/CD services for CD, despite CI/CD service providers supporting the deployment to many popular cloud services including AWS, AZURE, and

HEROKU. For this reason, the study proposes Gretel, an anti-pattern removal tool for TravisCI specifications, which can remove 70% of the most frequently occurring anti-pattern automatically.

Moreover, CI/CD outcome data, used by software practitioners and researchers when building tools and proposing techniques to solve software engineering problems, can be used “off the shelf” without checking for noise and complexity. The study shows that one in every 7 to 11 builds is incorrectly labeled. This noise may influence analyses based on CI/CD outcome data, suggesting that noise needs to be filtered out before subsequent analyses of CI/CD outcome data.

Furthermore, the results of the CI/CD builds could be delayed due to waiting time in the queues of CI/CD service providers and bottlenecks in the execution of CI/CD jobs, so to improve overall CI/CD performance queuing and job execution time needs to be reduced. Queuing time can be improved by evaluating different queuing algorithms under historical CI/CD service workload conditions while regarding execution time, a large-scale empirical evaluation of the job decomposition solution by comparing the duration of accelerated CI/CD builds of a sample of projects to baselines from popular CI/CD service providers can be performed.

2.3.2 Bad Practices

Continuous Integration (CI) has been claimed to introduce several benefits in software development, including high software quality and reliability. However, recent work pointed out challenges, barriers, and bad practices characterizing its adoption.

A mostly quantitative analysis [7] investigated a set of CI bad practices that are employed in open source projects, related to the use of CI with infrequent commits on the master branch, in a software project with poor test coverage, with builds that remain broken for long periods for time, and with builds with

considerably long duration. These bad practices constitute what is known as the “Continuous Integration Theater”. The study found that, in general, 60% of the projects in the considered dataset, curated by TravisTorrent, suffer from infrequent commits. In particular, half of Java have infrequent commits, which may hinder software development activities. With regard to build test coverage, although the overall coverage was 78%, the coverage of Java and Ruby projects differs greatly. The average code coverage of Ruby projects was 86%, whilst for Java projects, it was 63%. This suggests that although poor test coverage exists, a significant number of studied projects take care of their code coverage. Also, 85% of the analyzed projects have at least one build that took more than four days to be fixed. Interestingly, it was observed that large projects have less long broken builds than smaller projects. Finally, the study discovered that only 16% of the projects do not adhere to the 10 minutes rule of thumb for build duration. However, when considering Java large projects, the landscape changes significantly: 52% of them take more than 10 minutes.

The closest work to ours is by Zampetti et al.[13]. They performed an empirical characterization of bad practices in Continuous Integration, aimed at investigating what bad practices developers incur when using CI in their daily development activities. The investigation has been conducted by leveraging semi-structured interviews of 13 experts and mining more than 2,300 Stack Overflow posts. In particular, the goal of the study is to identify the bad smells developers incur when adopting CI and assess the perceived importance of such bad smells.

The study aims at addressing the following research questions:

- What are the bad practices encountered by practitioners when adopting CI?
- How relevant are the identified CI bad smells for developers working in CI?
- How our pieces of evidence confirm/contradict/complement the existing CI

pattern/antipattern catalog by Duvall (2011)?

79 CI bad smells emerged from the investigation, grouped into 7 categories related to different dimensions of CI pipeline management:

Repository : groups bad smells concerning a poor repository organization, and misuse of version control system (VCS) in the context of CI. Some smells deal with problems related to the repository structure which may affect the modularity of CI solutions; then, there are bad smells about branch misuses; finally, some smells concern the poor choice of configuration items.

Infrastructure Choices : groups bad smells related to a sub-optimal choice of hardware or software components while setting a CI pipeline. Hardware issues are mainly related to a poor allocation of the CI process across hardware nodes that could overload development machines or lose scalability, while software-related bad smells concern poor tool choices and configuration.

Build Process Organization : This category, the one with the largest number of bad smells (29), features CI bad smells related to a poor configuration of the whole CI pipeline. Some of such bad smells are related to the CI environment's initialization.

Build Maintainability : Since build configuration files often change over time and their changes induce more relative churn than source code changes, their maintainability is also an important concern. Problems can arise when a build configuration is coupled with a specific workspace, or when the build script is poorly commented, uses meaningless variable names, and modularity is not used when it should be.

Quality Assurance : This category relates to CI bad smells that are linkable to testing and static analysis phases. Bad smells related to testing are due

to the lack of optimization for testing tasks within a CI pipeline. Moreover, this category features smells related to how the test coverage thresholds are set.

Delivery Process : This category of bad smells concerns the storage of artifacts related to a project release. Furthermore, this category includes bad smells related to software release in the production environment.

Culture : This category includes bad smells whose symptoms might not be inferred by observing the CI pipeline, but are more human-related. They deal with the lack of a shared culture on how developers should behave when adopting CI. These include bad push/pull practices, e.g., pushing changes before a previous build failure is being fixed; poor prioritization of CI-related activities including the fixing of build failures, and Dev/Ops separation.

In the table below we list some of the bad smells identified, that we took into consideration in our analysis.

ID	Category	CI Bad Smell
BP3	Build Process Organization	Wide and inchoesive build jobs are used
BP4	Build Process Organization	Monolithic builds are used in the pipeline
BP14	Build Process Organization	Use of nightly builds
BM1	Build Maintainability	Absolute/machine-dependent paths are used
BM3	Build Maintainability	Environment variables are not used at all
BM8	Build Maintainability	Missing/Poor strict naming convention for build jobs

Chapter 3

Technology

3.1 Node.js

The entire project was realized using Node.js, an open-source, cross-platform, JavaScript runtime environment that executes JavaScript code outside a web browser.

Though .js is the standard filename extension for JavaScript code, the name "Node.js" doesn't refer to a particular file in this context and is merely the name of the product.

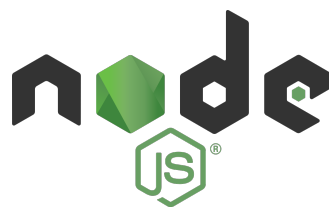


Figure 3.1: Node.js logo

Node.js has an event-driven architecture capable of asynchronous I/O. These design choices aim to optimize throughput and scalability in web applications with many input/output operations, as well as for real-time Web applications.

Node.js was written initially by Ryan Dahl in 2009, about thirteen years after the introduction of the first server-side JavaScript environment, Netscape's

LiveWire Pro Web. The initial release supported only Linux and Mac OS X. Its development and maintenance were led by Dahl and later sponsored by Joyent. In June 2011, Microsoft and Joyent implemented a native Windows version of Node.js. The first Node.js build supporting Windows was released in July 2011.

3.1.1 Advantages of Node.js

1. Node.js is an open-source framework under MIT license. (MIT license is a free software license originating at the Massachusetts Institute of Technology (MIT).)
2. Uses JavaScript to build the entire server-side application.
3. Lightweight framework that includes bare minimum modules. Other modules can be included as per the need of an application.
4. Asynchronous by default. So it performs faster than other frameworks.
5. Cross-platform framework that runs on Windows, MAC or Linux

3.1.2 NPM

In January 2010, a package manager was introduced for the Node.js environment called npm. The package manager makes it easier for programmers to publish and share the source code of Node.js packages and is designed to simplify the installation, updating, and uninstallation of packages.



Figure 3.2: Npm logo

Recapping, NPM is a package manager for Node.js packages or modules, and it is installed on your computer when you install Node.js. A package in Node.js contains all the files you need for a module, a JavaScript library you can include in your project. Downloading a package is very easy:

```
npm install package_name
```

3.1.3 Used libraries

Below we list the libraries that we used for the project, installed using the NPM program.

- **fs**: a module that provides a lot of very useful functionality to access and interact with the file system. There is no need to install it.
- **readline**: a module providing a way of reading a data stream, one line at a time.

```
var readline = require('linebyline'),
    rl = readline('./somefile.txt');
rl.on('line', function(line, lineCount, byteCount) {
    // do something with the line of text
})
.on('error', function(e) {
    // something went wrong
});
```

- **node-fetch**: a light-weight module that brings window.fetch to Node.js.
- **axios**: promise-based HTTP client for the browser and node.js
- **path**: provides utilities for working with file and directory paths.

- **yaml:** yaml is a JavaScript parser and stringifier for YAML, a human friendly data serialization standard. [3] It supports both parsing and stringifying data using all versions of YAML, along with all common data schemas. As a particularly distinguishing feature, yaml fully supports reading and writing comments and blank lines in YAML documents.

The library is released under the ISC open source license, and the code is available on GitHub. It has no external dependencies and runs on Node.js 6 and later, and in browsers from IE 11 upwards.

The API provided by YAML has three layers, depending on how deep you need to go: Parse & Stringify, Documents, and the CST Parser. The first has the simplest API and "just works", the second gets you all the bells and whistles supported by the library along with a decent AST, and the third is the closest to YAML source, making it fast, raw, and crude.

Finally, to include the installed modules, use the `require()` method as follows:

```
const fs = require('fs');  
const readline = require('readline');  
const fetch = require('node-fetch');  
const axios = require('axios');  
const path = require('path');  
const yaml = require('yaml');
```

3.2 GitHub Api

The accomplishment of our goal was possible mainly thanks to the use of API Rest GitHub v3. [2]

By default, all requests to <https://api.github.com> receive the v3 version of the REST API. All API access is over HTTPS and accessed from <https://api.github.com>.

All data is sent and received as JSON.

When you fetch a list of resources, the response includes a subset of the attributes for that resource. This is the "summary" representation of the resource. When you fetch an individual resource, the response typically includes all attributes for that resource. This is the "detailed" representation of the resource.

3.2.1 Authentication

There are two ways to authenticate through GitHub API v3, to prevent the accidental leakage of private repositories to unauthorized users:

1. **Basic authentication:** using GitHub username
2. **OAuth2 token:** a token sent in a header.

The one used in this project is the OAuth2 token : you can generate a personal access token for quick access to the GitHub API by going in your GitHub account/settings/developer settings/personal access tokens. The token was used to make the calls to Api GitHub in the following way:

```
const headers = {  
  "Authorization": 'Token putyourtokenhere '  
}
```

3.2.2 Pagination

Requests that return multiple items will be paginated to 30 items by default. You can specify further pages with the ?page parameter. For some resources, you can also set a custom page size up to 100 with the ?per_page parameter.

3.2.3 Search

The Search API helps you search for the specific item you want to find. For example, you can find a user or a specific file in a repository. Think of it the way you think of performing a search on Google. It's designed to help you find the one result you're looking for (or maybe the few results you're looking for). Just like searching on Google, you sometimes want to see a few pages of search results so that you can find the item that best meets your needs. To satisfy that need, the GitHub Search API provides up to 1,000 results for each search. However, it is possible to narrow your search using queries, as we will see in the next chapter.

Each endpoint in the Search API uses query parameters to perform searches on GitHub. See the individual endpoint in the Search API for an example that includes the endpoint and query parameters.

A query can contain any combination of search qualifiers supported on GitHub.com. The format of the search query is:

`q=SEARCH_KEYWORD_1+SEARCH_KEYWORD_N+QUALIFIER_1+QUALIFIER_N`

The Search API does not support queries that:

- are longer than 256 characters (not including operators or qualifiers).
- have more than five AND, OR, or NOT operators.

These search queries will return a "Validation failed" error message.

3.2.4 Rate limiting

For API requests using Basic Authentication or OAuth, you can make up to 5000 requests per hour. Authenticated requests are associated with the authenticated user, regardless of whether Basic Authentication or an OAuth token was used. This means that all OAuth applications authorized by a user share the same quota

of 5000 requests per hour when they authenticate with different tokens owned by the same user.

However, the Search API has a custom rate limit: for requests using Basic Authentication, OAuth, or a client ID and secret, you can make up to 30 requests per minute; for unauthenticated requests, the rate limit allows you to make up to 10 requests per minute.

3.2.5 Example

In conclusion of this chapter, we show an example of GitHub API Search query and corresponding JSON result:

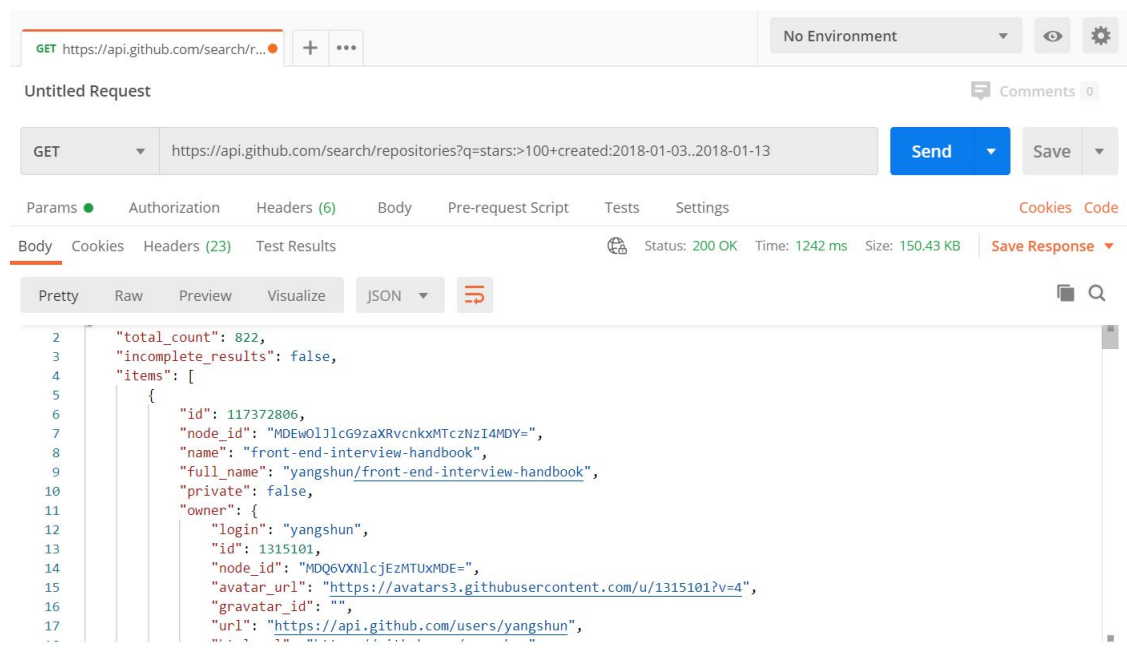


Figure 3.3: Query GitHub Api and JSON result using Postman

Chapter 4

Preliminar study

Continuous Integration is becoming one of the biggest success stories in automated software engineering. CI systems automate the compilation, building, testing, and deployment of software. Despite the growth of CI, to the best of our knowledge, the last published research paper related to CI usage dates back to 2017 [10]. For this reason, we decided to conduct an empirical study to study how the usage of CI changed in the past years.

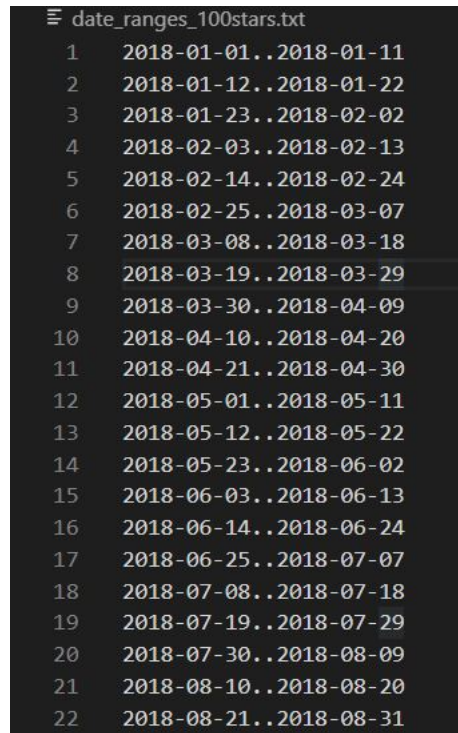
4.1 Research questions

This work aims to study the *usage of CI in open-source projects in the last years*. We analyzed open-source projects from GitHub to answer the following research questions:

- **RQ1:** How frequently is CI practice used in open-source projects?
- **RQ2:** What is the breakdown of usage of different CI services?

4.2 Objects

To answer the above questions, we analyzed *GitHub repositories created from 2018 on*, and applying a filter that considers only the ones with *more than 100 stars*, we studied *43193 repositories* overall.



	date_ranges_100stars.txt
1	2018-01-01..2018-01-11
2	2018-01-12..2018-01-22
3	2018-01-23..2018-02-02
4	2018-02-03..2018-02-13
5	2018-02-14..2018-02-24
6	2018-02-25..2018-03-07
7	2018-03-08..2018-03-18
8	2018-03-19..2018-03-29
9	2018-03-30..2018-04-09
10	2018-04-10..2018-04-20
11	2018-04-21..2018-04-30
12	2018-05-01..2018-05-11
13	2018-05-12..2018-05-22
14	2018-05-23..2018-06-02
15	2018-06-03..2018-06-13
16	2018-06-14..2018-06-24
17	2018-06-25..2018-07-07
18	2018-07-08..2018-07-18
19	2018-07-19..2018-07-29
20	2018-07-30..2018-08-09
21	2018-08-10..2018-08-20
22	2018-08-21..2018-08-31

Figure 4.1: Date ranges from 2018 to nowadays

4.3 Variables

For the first research question, we don't have variables: we just set some parameters, such as *year of creation* and *number of stars* for the project.

The second research question has *type of CI service* as independent variable, and *frequency of occurrence* of that service in GitHub repositories (still created after 2018 and having more than 100 stars) as dependent variable.

4.4 Experimental procedure

Due to GitHub API limits, we first performed some queries using Postman, to see how many repositories with more than 100 stars are created on average daily, to establish what ranges (of how many days) we need to consider to not exceed the limit.

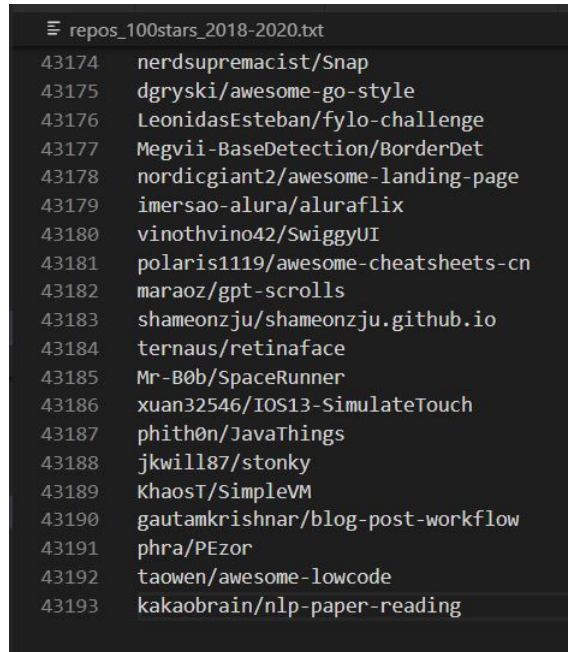
Then, we wrote a script, named *"getRepos.js"*, that uses GitHub Api to obtain repositories created in the considered ranges. To do this, we performed the following query:

```
const api_url = "https://api.github.com";  
const query = (api_url + "/search/repositories?q=stars:>=100");
```

We saved the names of the 43193 repositories, as showed in the figure 4.2, in "repos_100stars_2018-2020.txt" file.

Then, we wrote another script, *"getYamlPaths.js"*, that reads the names of repositories from the file above, and checks if they contain at least one YAML file, thanks to the following query:

```
const yaml = (api_url + "/search/code?q=language:yaml+repo:");
```



repos_100stars_2018-2020.txt	
43174	nerdsupremacist/Snap
43175	dgryski/awesome-go-style
43176	LeonidasEsteban/fylo-challenge
43177	Megvii-BaseDetection/BorderDet
43178	nordicgiant2/awesome-landing-page
43179	imersao-alura/aluraflix
43180	vinothvino42/SwiggyUI
43181	polaris1119/awesome-cheatsheets-cn
43182	maraoz/gpt-scrolls
43183	shameonzju/shameonzju.github.io
43184	ternaus/retinaface
43185	Mr-B0b/SpaceRunner
43186	xuan32546/IOS13-SimulateTouch
43187	phith0n/JavaThings
43188	jkwill87/stonky
43189	KhaosT/SimpleVM
43190	gautamkrishnar/blog-post-workflow
43191	phra/PEzor
43192	taowen/awesome-lowcode
43193	kakaobrain/nlp-paper-reading

Figure 4.2: Names of repositories with more than 100 stars created from 2018 on

However, since projects use YAML language not only for CI but also for other purposes, we applied another filter, by code, that considers only the YAML files containing specific words in their paths. About that, according to the study [10], the four CI services that we were able to readily identify manually and later by our script are *Travis CI*, *CircleCI*, *AppVeyor*, and *Wercker*. Moreover, we added another CI tool, introduced after this study, that is *GitHub actions*. Indeed, the tools listed above can be easily identified, since they presuppose that YAML files contain specific words in the name (Travis, appveyor, wercker), or are collocated in a specific path (.circleci/, .github/workflows).

After identifying the files' paths, we saved them in an appropriate file, as we can see in figure 4.3.


```

1 https://raw.githubusercontent.com/yuzu-emu/yuzu/c63e68c48015bf9dff5ef53b0f14d18e7e214107/.travis.yml
2 https://raw.githubusercontent.com/digitalocean/nginxconfig.io/7444fa6f7793ddb73d4a9cc76eccf8357368dc43/.github/workflows/gh-pages-w
3 https://raw.githubusercontent.com/digitalocean/nginxconfig.io/7444fa6f7793ddb73d4a9cc76eccf8357368dc43/.github/workflows/do-spaces-
4 https://raw.githubusercontent.com/Anankke/SSPanel-Uim/33064f12d49acc7a023b42c73f7f65f18d95593d/.travis.yml
5 https://raw.githubusercontent.com/Anankke/SSPanel-Uim/5a8b73aa239e8fa62988fe4b37206ca74510d7ce/.github/workflows/continuous-integra
6 https://raw.githubusercontent.com/google/data-transfer-project/214ec5866dae841e1d0f4420c901124b9c72292c/.travis.yml
7 https://raw.githubusercontent.com/florent37/ExpansionPanel/22e707bcae511bf26151b0e04adc628a12615d48/.circleci/config.yml
8 https://raw.githubusercontent.com/GitHawkApp/MessageViewController/3916d73bd91e2461f1a6351d8690555a39b5f721/.travis.yml
9 https://raw.githubusercontent.com/boltgolt/howdy/e495bdac5fd14438710a248e8e0c6b5e5d05c1a4/.travis.yml
10 https://raw.githubusercontent.com/uber-common/jvm-profiler/ac78f5733c48455bb960194eac3b8fd95298ab6c/.travis.yml
11 https://raw.githubusercontent.com/hugoam/two/b814e80c79bbc2bf0aa6d0917a51a32e8dfbc308/.travis.yml
12 https://raw.githubusercontent.com/hugoam/two/33cf845a75cf9453fde81310f26f40522724e5b4/.appveyor.yml
13 https://raw.githubusercontent.com/GitHawkApp/StyledTextKit/d47830c8159afe5f3f5fd57d4fe704872285465/.travis.yml
14 https://raw.githubusercontent.com/ipazc/mtcnn/718372bbd7eaac77125505451e37598ae930b88d/.travis.yml
15 https://raw.githubusercontent.com/GoogleCloudPlatform/spark-on-k8s-operator/3e528cbb943ca13a0dae5b2c609daabc844d14f/.travis.yml
16 https://raw.githubusercontent.com/shshaw/Splitting/c91bc37b4b38574dfa1e5a16ab0542152bb8f344/.travis.yml
17 https://raw.githubusercontent.com/michalochman/react-pixi-fiber/6bae161e8e85d8bde38f24db47fe8a7333ca9011/.circleci/config.yml
18 https://raw.githubusercontent.com/pa-bru/graphql-cost-analysis/c42bf117c6ac16a7bd4e8c74c1e4d68cdcb56a2/.travis.yml
19 https://raw.githubusercontent.com/jhipster/prettier-java/646ac86423ab54aa38896e7edb9eeade5c339831/.github/workflows/github-ci.yml
20 https://raw.githubusercontent.com/jhipster/prettier-java/646ac86423ab54aa38896e7edb9eeade5c339831/.github/workflows/binaries.yml
21 https://raw.githubusercontent.com/gfxfundamentals/threejsfundamentals/956b7c9d417c44ac92153178d1e49608b27e76d5/.travis.yml

```

Figure 4.3: YAML files' paths

Then, we classified the different YAML files with *"downloadYamls.js"* script, downloading and putting them in the appropriate folders. To do this, we used some simple classification rules, mainly based of the presence of a particular word in the file's path, as shows figure 4.4:

```

if(url.includes(".circleci/config.yml") == true){
  const file = fs.createWriteStream(directory_circle + (i+'_'+filename));
  const response = await axios({
    url,
    method: 'GET',
    responseType: 'stream'
  })
  try{
    response.data.pipe(file);
  }
  catch(err){
    console.log(err)
  }
}

else if (url.includes(".github/workflows") == true){

  const file = fs.createWriteStream(directory_github + (i+'_'+filename));
  const response = await axios({
    url,
    method: 'GET',
    responseType: 'stream'
  })
  try{
    response.data.pipe(file);
  }
  catch(err){
    console.log(err)
  }
}

```

Figure 4.4: Some adopted classification rules

Doing so, the files will be downloaded and saved in the proper folder, which represents the adopted CI service, and we can use them for further analysis.

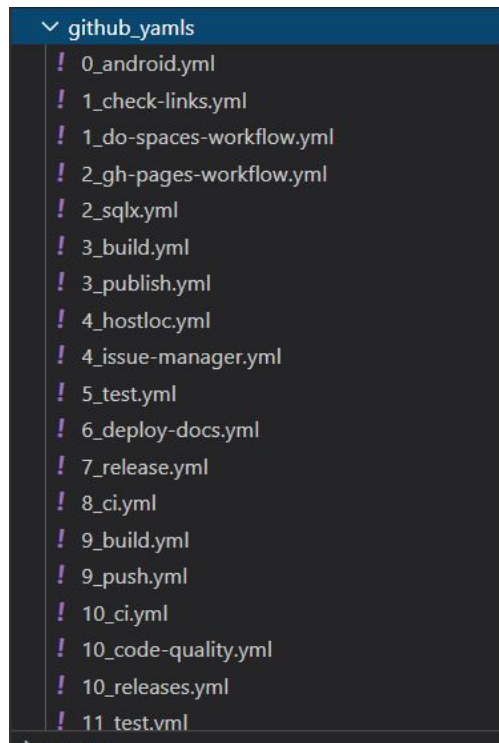


Figure 4.5: GitHub actions yamls

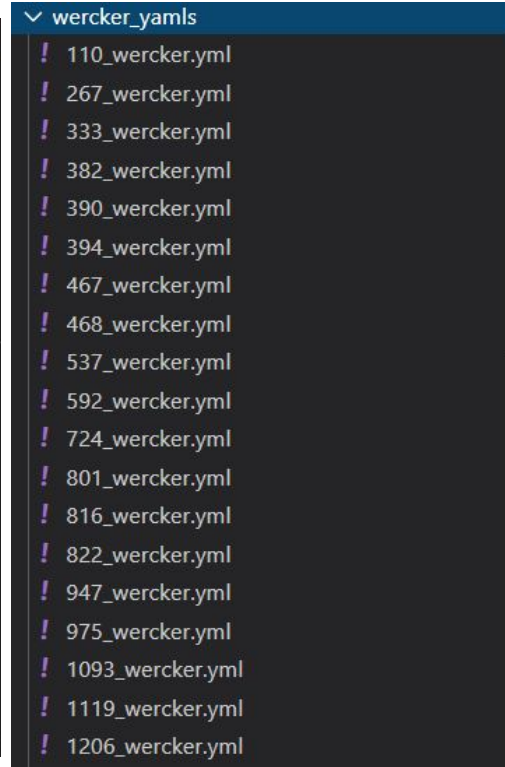


Figure 4.6: Wercker yamls

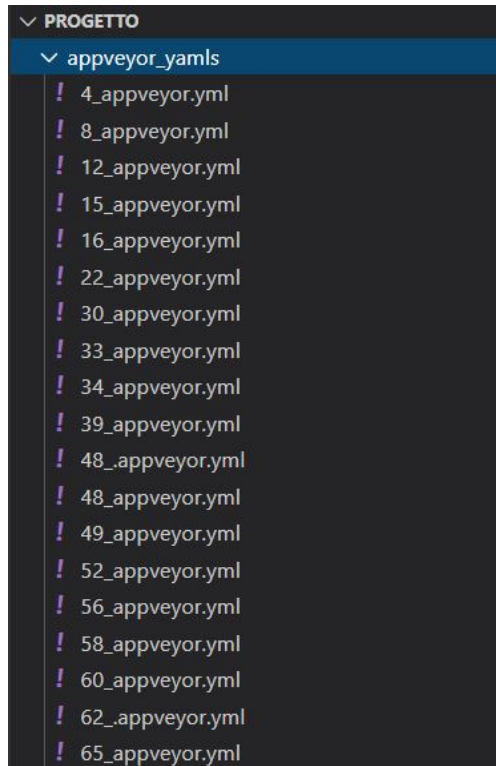


Figure 4.7: Appveyor yamls

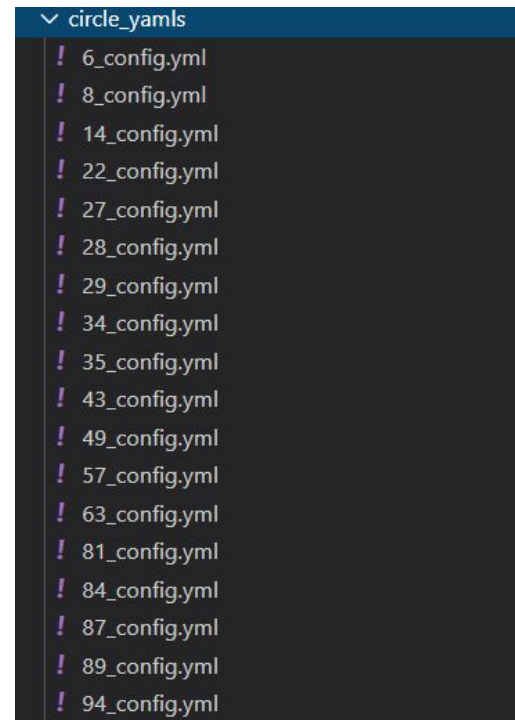


Figure 4.8: Circle yamls

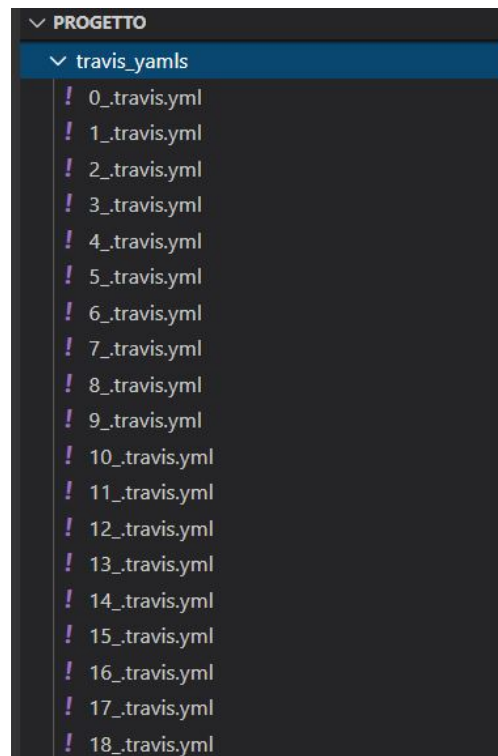


Figure 4.9: Travis yamls

4.5 Results

In this section, we present the results of our research questions.

RQ1: How frequently CI practice is used in open-source projects?

Project uses CI?	Percentage of projects	Number of projects
Yes	9,09%	3930
No	90,91%	39263

Table 4.1: Percentage of analyzed projects using CI

First of all, regarding the first research question, we found that only 9,09% of the considered projects, so 3930 of them, use CI. That means that this practice is still not very used in open-source projects.

RQ2: What is the breakdown of usage of different CI services?

CI services used in Github projects		
Service	Number of projects	Percentage of projects
Travis	2053	52,2%
Github actions	1073	27,3%
Circle	408	10,4%
Appveyor	373	9,5%
Wercker	23	0,6%

Table 4.2: Percentage of usage of different CI services

Next, we investigated which CI services are the most widely used in *3930 projects* that resulted to use CI practice. Of those 3930 projects, we identified 2053 projects that use Travis CI, 1073 use the GitHub actions tool, 408 use Circle, 373 use AppVeyor, and 23 use Wercker.

Therefore, as figure 4.10 shows, Travis CI is by far the most widely used CI service, with 52,2% of projects using this tool, as a confirmation of the previous studies.

However, the percentage of usage of this tool decreased after the introduction of GitHub actions, which is the second most used CI tool, with 27,3%.

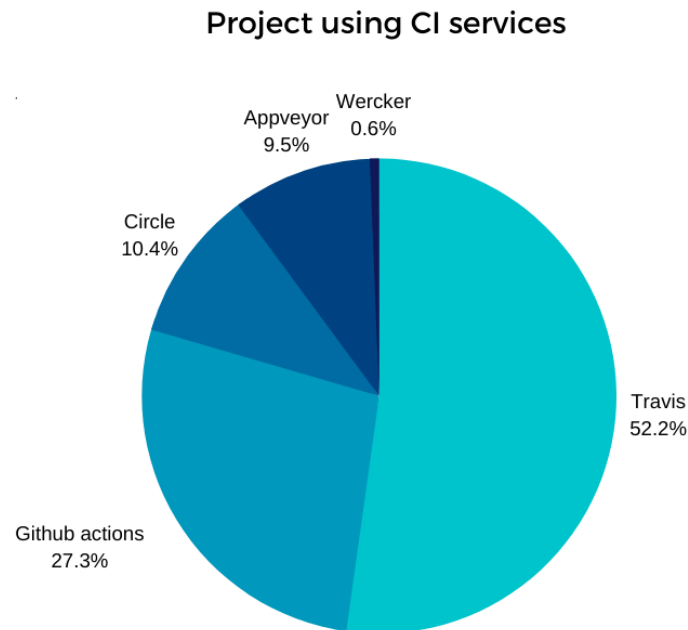


Figure 4.10: Usage of CI services in GitHub projects

On the other hand, only a minor percentage of projects uses tools like CircleCI, AppVeyor, whereas tool like Wercker is almost not used at all, with only 0,6%.

4.6 Threats to validity

4.6.1 Internal

Is there something inherent to how we collect and analyze CI usage data that could skew the accuracy of our results?

It must be said that, due to GitHub API limits, the execution of our scripts required a longer amount of time, since we couldn't perform many HTTP requests

in a certain period.

Therefore, since the execution of several scripts requires an internet connection, and the execution was not monitored for the entire time, it could be that in some moments, due to bad internet connection, was impossible to perform a proper analysis.

4.6.2 External

Are our results generalizable for general CI usage?

We are aware that our scripts do not find all CI usage (e.g., projects that run privately hosted CI systems, or use different services from the ones considered in this work). We can reliably detect the use of (public) CI services only if their API makes it possible to query the CI service based on knowing the GitHub organization and project name.

Moreover, we analyzed only a small part of projects, the ones with more than 100 stars, and created after 2018. Doing so, we are not taking in consideration projects that were created before, but started to use CI practice later.

Therefore, the results we present are a lower bound on the total number of projects that use CI.

Additionally, we only selected projects from GitHub. Perhaps open-source projects that have custom hosting also would be more likely to have custom CI solutions.

About RQ2, it must be said that a threat to the validity of this result could be the fact that we considered the projects created starting from the beginning of 2018, whereas GitHub introduced the actions tool a little bit later, so the difference between the percentage that uses Travis and the one that uses GitHub could be different.

Chapter 5

Empirical study

From the results of our preliminary study, we decided to address further questions. First, since the second most used CI tool resulted to be GitHub actions, and they are already many existing studies about the first one, Travis, we decided to further analyze the use of the tool introduced by GitHub.

5.1 Research questions

We want to answer the following research questions, that we grouped into 2 themes:

Theme 1: General usage

- **RQ3:** How much complex are CI GitHub actions files?
- **RQ4:** When do CI workflows of GitHub actions run?
- **RQ5:** What kind of tasks are performed by Github workflows?
- **RQ6:** How much complex are GitHub actions jobs?

Theme 2: Bad practices

- **RQ7:** How frequently does the job's name not express what task has a job to automate?

- **RQ8:** How frequently are GitHub actions used in GitHub workflows?
- **RQ9:** What is the usage of comments in Github actions?
- **RQ10:** What is the usage of job sequentiality in GitHub actions?
- **RQ11:** What is the usage of environment variables in GitHub actions?
- **RQ12:** What is the usage of absolute paths in GitHub actions?
- **RQ13:** How many bad practices can be found in GitHub actions files?

5.2 Objects

To answer the questions above we used the results from the preliminary study, which are *1773 YAML files* using GitHub actions as a CI tool, belonging to 1073 different repositories. However, since a later analysis found that 5 of them do not contain any jobs, so are not used for CI purposes, we analyzed *1768 GitHub YAML files*.

5.3 Variables

RQ	Independent variable	Dependent variable
RQ3	Number of jobs in a file	Number of files with a certain number of jobs
RQ4	Trigger event	Number of files containing that trigger event
RQ5	Names of jobs	Number of jobs with that name
RQ6	Number of steps in a job	Number of jobs having that number of steps
RQ6	Number of lines per job	Average number of lines per job
RQ7	Job's name significance	Number of jobs having not significant name
RQ8	Usage of Github actions	Number of files using github actions
RQ9	Absence of comments	Number of files not having comments
RQ10	Usage of sequentiality	Number of files using sequential jobs
RQ11	Absence of environment variables	Number of files not using environment variables
RQ12	Presence of absolute paths	Number of files containing absolute paths
RQ13	Number of bad practices	Number of files containing that number of bad practices

For the RQ3, the independent variable is represented by the *number of jobs* in a file, whereas the dependent variable is the *number/percentage of files* having that number of jobs.

To answer the RQ4, we use *type of trigger event* as independent variable, while the dependent variable is the *number/percentage of files* in which that event is contained.

To answer the RQ5 we use the *names of jobs* as independent variable to classify the jobs, and the *percentage of jobs belonging to a certain class* as dependent one.

For the RQ6, the independent variable is represented by the *number of steps* in a job, whereas the dependent variable is the *number/percentage of jobs* having that number of steps.

The independent variable for the RQ7 is represented by *job's name significance*, whilst the dependent variable is the *amount of jobs having not significative names*. We address this research question as one of the bad smells in [13], belonging to Build Maintainability category, is "Missing/Poor strict naming convention for build jobs".

To answer the RQ8 we use the *usage of GitHub actions* as independent variable, and the *number/percentage of files that do/do not use Github actions* as dependent one.

One of the bad smells listed in [13] is that "the build script is poorly commented". For this reason, we decided to address RQ9 and see if this bad smell can be also encountered in GitHub actions files. For the RQ9, the independent variable is represented by the *absence of comments* in a project, whereas the dependent variable is the *number/percentage of projects* not using comments.

For the RQ10, the independent variable is the *usage of sequential jobs*, while the dependent variable is represented by *number of files using sequential jobs*.

As another bad smell in CI usage identified by [13] is that "Environment variables are not used at all", we decided to address RQ11, so to analyze what is the

usage of environment variables in GitHub yamls. The independent variable for this question is the *absence of environment variables*, and the dependent variable is the *number of files that do not use environment variables*, which is considered a bad practice.

Finally, as "Absolute/machine-dependent paths are used" bad smell has a high perceived relevance, considering that their presence will unavoidably limit the portability of the build resulting in statements such as "but it works on my machine", according to [13], we address the RQ12, for which the independent variable is the presence of absolute paths in GitHub files, while the dependent one is represented by the number of files containing absolute paths.

In conclusion, we decided to analyze how many of the above bad practices are present in each file, so we address the RQ13. For this last research question, we considered the *number of bad practices* as independent variable, and *number of files containing that number of bad practices* as dependent one.

5.4 Experimental procedure

As we mentioned above, to answer the research questions above we used the 1768 files contained in the folder *"github_yamls"*.

In particular, the script *"getJobs.js"* writes in *"jobs_github_numbers.txt"* file the number of jobs for every file contained in the considered folder, in order to answer the RQ3. The count of jobs was possible thanks to the *yaml* library, which allows parsing the files to YAML format.

A similar procedure was used also for the RQ4, in *"getTriggerEvents.js"* file, where we wrote all the trigger events present in github files in *"trigger_events.txt"* file.

To answer the RQ5, we classified the 2609 jobs found in github considered yaml files. The classification is performed by *"classifyJobs.js"* file, and is based

```
const directoryPath = path.join(__dirname, 'github_yamls');
const directory = './github_yamls/';

const jobs_numbers = fs.createWriteStream('jobs_github_numbers.txt');
const jobs_ids = fs.createWriteStream('jobs_github_ids.txt');
const job_names = fs.createWriteStream('job_names.txt');

const defaultOptions = {
  maxAliasCount: -1,
}

fs.readdir(directoryPath, function (err, files) {
  if (err) {
    return console.log('Unable to scan directory: ' + err);
  }
  files.forEach(async function (file_name) {
    await getJobs(file_name);
  });
});
```

Figure 5.1: A fragment from "getJobs.js" script

```
function getJobs(file_name){
  try{
    var file = fs.readFileSync(directory + file_name, 'utf8')
    var yaml_file = yaml.parse(file, defaultOptions)
    const n_jobs = yaml2array(yaml_file.jobs).length
    jobs_numbers.write(`${n_jobs}, ${file_name} \n`)

    if(n_jobs==1){
      job_names.write(`${Object.keys(yaml_file.jobs)} \n`)
    }
    for (let i=0; i< yaml2array(yaml_file.jobs).length; i++)
    {
      jobs_ids.write(`${Object.keys(yaml_file.jobs)[i]}, ${file_name} \n`)
    }
  }

  catch(err){
    console.log(err + " " + file_name);
  }
}
```

Figure 5.2: A fragment from "getJobs.js" script

on the jobs' names or/and ids. In case these information are not significant, we took into consideration also the file's name. We sum up the classification rules in the tables below:

Job's classes				
Generic CI	Documentation	Build	Install	Quality assurance
CI, sync, generate, stale, issues, push, invite, rebase, label, run, nightly, setup, notification, notify, purge-cache, license, clean, execute	documentation, docgen, docs, comment	build	install	test, check, quality, bench, qA, audit, fuzz

Job's classes				
Integration test	Unit test	White-box test	Static analysis	Acceptance test
integration-test	unit-test	codecov, coverage	lint, clang, sonar, findbugs, validate, verify, scanner, flake, static-analysis	acceptance

Job's classes				
Release	Deploy	Update	Refactoring	Hello-world
release, publish	deploy, deliver, docker, prepare, macOS, windows, ubuntu, linux	update	format, prettier, style, whitespace	hello-world, greetings

In the tables above we can see the different job classes taken into consideration, and the words that need to be contained in job's name/id to affirm the job belongs to that class.

The considered rules allow classifying 92,99% of jobs, whereas 7,01% of them remain unclassified.

To answer the RQ6, the *"getSteps.js"* file calculates the number of steps for every job, , and writes the results in *"jobs_steps_number.txt"* file. Plus, we also

wrote *getRowsPerJob.js* file. It considers each YAML file as an array of strings, represented by lines. Then, it considers the index of every job's id in this array. Again, it performs the subtraction between each index to see how many lines every job contains. Note that to calculate the number of rows of the last job the index of the last row of the file is considered. The numbers of rows per job of a file are written in *rows_per_job.txt* file.

To answer the RQ7 we used another script, "analyzeJobsNames.js", that is quite similar to the "classifyJobs.js" file. It reads from the "jobs_id_name_file.txt" the job's names and ids, and it writes to the "jobs_names_significance.txt" all the jobs, specifying if they are significant or not. The significance is given by the presence of the different words that we used for the job classification, that we can see in the tables.

To see how frequently are GitHub actions used, and answer the RQ8, the "getActions.js" file was written. It analyzes every file contained in *github_yamls* folder, and writes in "use_of_actions.txt" file if they use or not Github actions.

The *getRowsComments.js* file counts first the number of rows per file, and next to the number of rows starting with "#", that is comment's indicator in GitHub Yaml files, to answer the RQ9. It writes the results in *number_of_rows_and_comments_per_file.txt* file.

To answer RQ10, *getNeeds.js* file writes in *jobs_needs.txt* file the number of "needs" occurrences (that is the indicator of sequentiality between jobs) per file, and the number of jobs in that file, to see the kind of sequentiality that is used.

Regarding RQ11, first of all, we saw that environment variables in GitHub actions can be used in 3 different ways:

1. A map of environment variables that are available to *all jobs and steps in the workflow*.
2. A map of environment variables that are available to *all steps in a single*

job.

3. A map of environment variables that are available *only to a single step*.

For this reason, *getEnvVariables.js* file has 3 functions: "analyzeGenEnv", "analyzeJobsEnv", and "analyzeStepsEnv". It writes per each file if it uses one of the three kinds of environment variable in *use_of_env_variables.txt* file.

Finally, to get an answer to RQ12, first of all is important to understand the difference between an absolute path and a relative one:

Absolute path: An absolute path is defined as specifying the location of a file or directory from the root directory(/). In other words, we can say the absolute path is a complete path from the start of the actual filesystem from / directory. We can use an absolute path from any location.

Relative path: A relative path is defined as path related to the present working directory. If you want to use a relative path we should be present in a directory where we are going to specify relative to that present working directory.

The file *getPaths.js* checks, for each YAML file, if they contain an absolute path, and writes in the output file, *paths.txt*, the number of the absolute paths detected and the paths themselves. It allowed a further visive analysis, that found some false positives, which led to some changes in js file. Indeed, we avoided considering some particular combinations of characters.

In the conclusion of our analysis, we decided to address RQ13, and see how many of the above discussed bad practices can be found at the same time in GitHub actions files. To reach this goal, *getBadPractices.js* file was written.

Mainly, the file reports the scripts already written as sub-functions, and calculates for each yaml file how many of the bad practices described above are contained in it. So, it has the following functions, called for each file:

- `getComments()`
- `getActions()`
- `getEnvironmentVariables()`
- `getAbsolutePath()`
- `getJobsNamesSignificance()`

We don't report the procedure of these functions, since they are quite similar to the scripts described above.

The goal is to find how many of these bad practices can be found simultaneously in a file.

5.5 Results

RQ3: How much complex are CI GitHub actions files?

Number of jobs in GitHub YAML files		
Number of jobs	Percentage of files	Number of files
1	82,5%	1412
2	10,4%	178
3	1,2%	77
4	2,6%	45
5 or more	3,3%	56

Table 5.1: Number of jobs in GitHub actions YAML files

We measured the complexity of Github actions files counting the number of jobs that they contain. The result was that: 82,5% of files contain 1 job, 10,4% contain 2 jobs, 1,2% contain 3 jobs, 2,6% contain 4 jobs, and 3,3% contain 5 or more jobs.

Since the vast majority of files resulted to contain only one job, as the table 5.1 and the figure 5.3 show, we decided to study what are the names of these jobs.

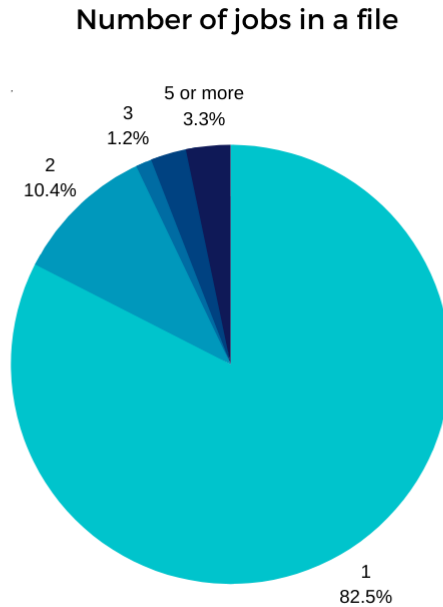


Figure 5.3: Number of jobs in GitHub actions files

The result, that we wrote in *"job_names.txt"* file, was that the majority of them contain **"build"** in their names (build*), which means that their only purpose is to automatize the building process. We can see the most common words present in single jobs' names in the bar graph below:

RQ4: When do CI workflows of GitHub actions run?

Trigger events in github actions files		
Trigger event	Number of files	Percentage of files
push	1344	76,01%
pull_request	888	50,22%
schedule	167	9,44%
issues	121	6,84%
repository_dispatch	35	1,97%
release	33	1,86%
others	24	1,35%

Then we investigated what are the trigger events more widely used in Github

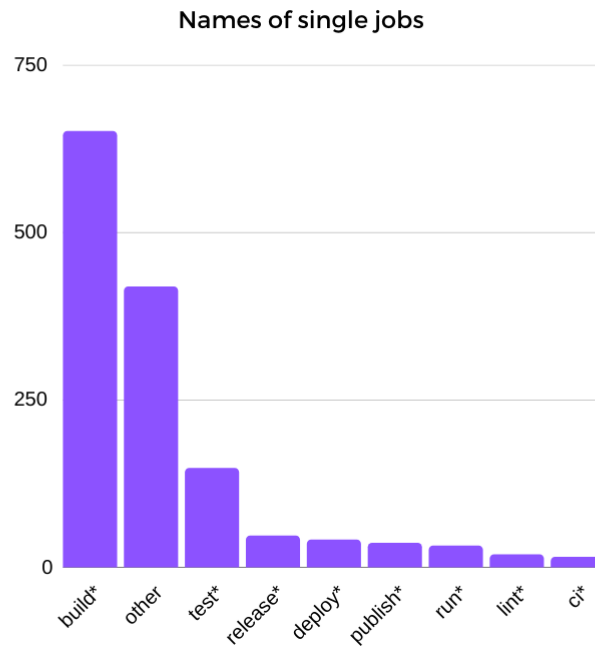


Figure 5.4: Names of single jobs

actions files. The result that we can see in figure 5.5 is that the two more frequently used events are **push** and **pull_request**. It means that the vast majority of workflows are configured to run after local changes are pushed to online repository, or once a pull request is sent.

Moreover, a minor percentage of files use events such as **schedule** and **issues**. A scheduled build either (unnecessarily) builds a change a second time or is a sign that a change is not automatically built, which breaks the idea of always ensuring a working system. We propose to warn about build configurations that schedule builds. Finally, less than 2% use events such as **repository_dispatch**, **release**, or others.

RQ5: What kind of tasks are performed by Github workflows?

To answer this question we used the classification rules describes above, and

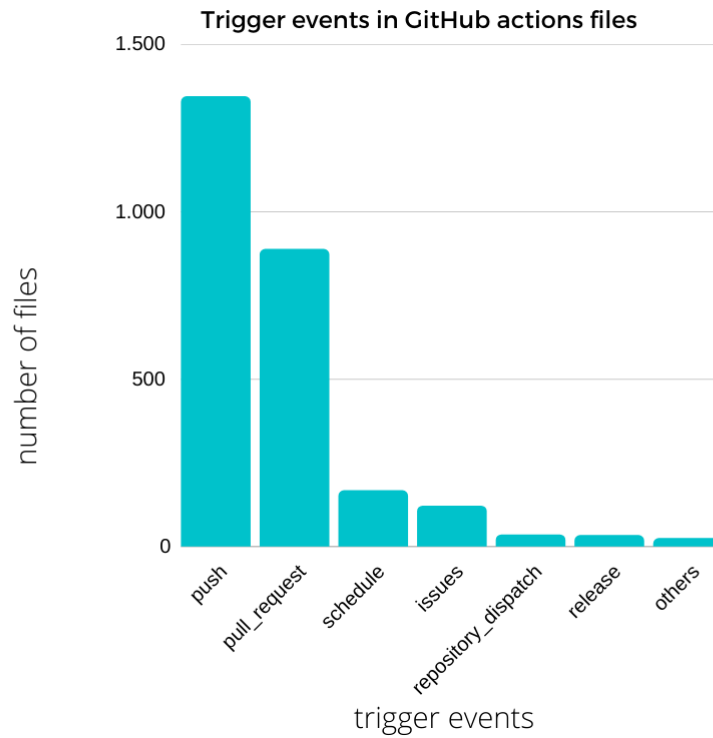


Figure 5.5: Trigger events in GitHub actions files

we analyzed how frequently jobs belong to the proposed classes. The result is shown below:

Job's Classification	
Class	Number of jobs
Build	970
Quality assurance	599
Generic CI	309
Deploy	308
Release	276
Static analysis	152
Update	37
Refactoring	37
Integration test	31
Unit test	30
Documentation	30
White-box test	22
Install	11
Hello-world	11
Acceptance	2

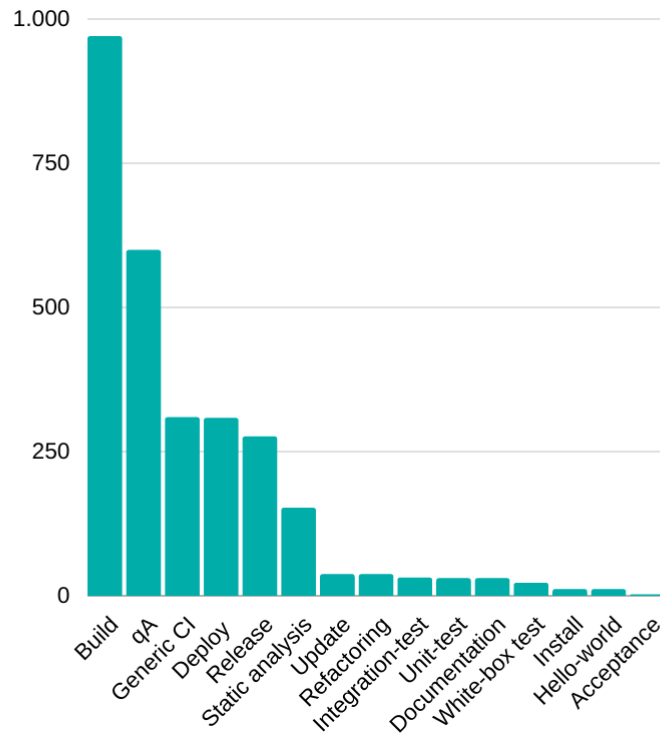


Figure 5.6: Jobs' classes

From figure 5.6 we can see that the most automated task by GitHub workflows is the building activity. Also, many jobs are used to perform quality assurance. The other tasks, least frequently used, are shown in the figure above.

RQ6: How much complex are GitHub actions jobs?

Steps in Github actions jobs		
Number of steps	Percentage of jobs	Number of jobs
1	5,4 %	142
2	13,6 %	354
3	16,4 %	428
4	18,2 %	475
5	15,8 %	413
6	10,4 %	271
7	5,7 %	150
8	5,1 %	134
9	2 %	53
10 or more	7,2 %	189

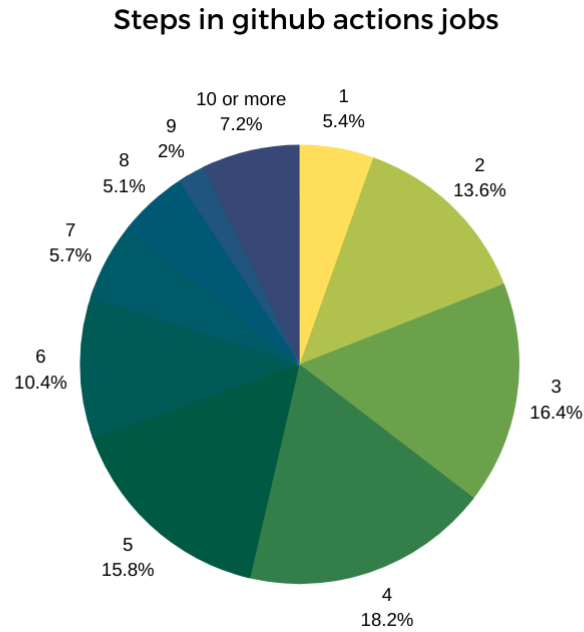


Figure 5.7: Steps in github actions jobs

We measured the complexity of GitHub actions jobs counting the number of steps that they contain. The result is shown in 5.7.

As we can see, only 5,4% of jobs have 1 step, so jobs with more than 1 step are more frequent. This is reasonable if we think that a previous result, about the number of jobs, reports that the majority of GitHub YAML files contain only one job.

As another metric to analyze the job complexity, we asked "What is the average number of lines per job in GitHub actions?"

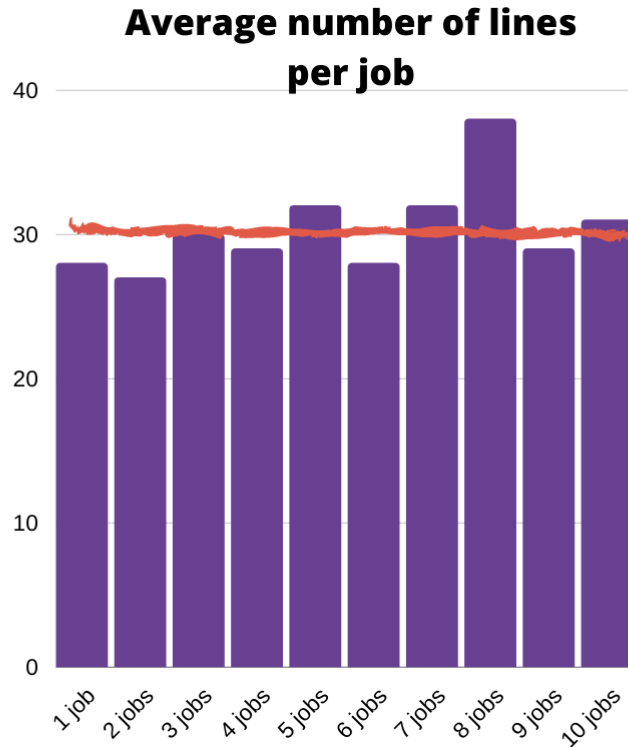


Figure 5.8: Average number of lines per job as the number of jobs changes

Number of jobs	Average number of lines per job
1	28
2	27
3	30
4	29
5	32
6	28
7	32
8	38
9	29
10	31

To answer this question we first calculate the number of rows per each job and then carry out the calculation of the average. In particular, we first performed the average calculation as the number of jobs changes, whose result is shown in the table. Finally, an overall average was calculated, and the result was of **30 lines per job**, as we can see from the red line in figure 5.8.

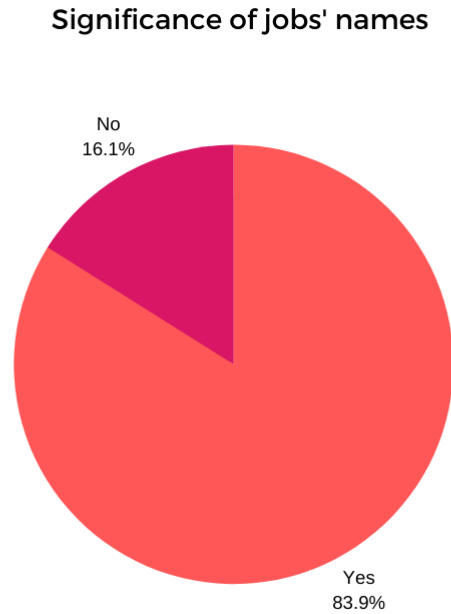


Figure 5.9: Jobs'names significance

RQ7: How frequently does the job's name not express what task has a job to automate?

After the RQ5, we decided to analyze how frequently job names do not express the task performed by the job, or more generically how frequently they are not significant, considering it a bad practice.

The result of this research question is shown in the pie graph in 5.9.

From figure 5.9 we can see that 83,9% (2190 out of 2609) of jobs have significant names, whereas the names of 16,1% (419 out of 2609) of jobs are not meaningful. Note that the percentage of not significant jobs' names is major than the percentage of unclassified jobs for the question RQ5, since for that question we took into consideration also the file name, and since some classes are not considered as significant now.

For the jobs whose names are considered not meaningful, we decided to do further analysis, and see why they are not significant, and what they express. This analysis is performed by the "analyzeNotSignificantNames.js" file.

We identified some common elements expressed by the jobs' names instead of the performed task, that are:

- Operating system
- Technology
- Language

Nevertheless, the vast majority of jobs do not even express one of the above-listed parameters, so they are not really significant.

In conclusion, "The poor strict naming convention for build jobs" bad practice is not really widespread in GitHub actions files.

RQ8: How frequently are GitHub actions used in GitHub workflows?

Usage of github actions	Percentage of files	Number of files
Yes	98,6%	1744
No	1,4%	24

As we can see from figure 5.10, 98.6% of files use GitHub actions, whilst only a little percentage, 1.4% do not use this service.

On the one hand, the usage of this service increases reuse. On the other, it leads to a major dependence on a service that evolves over time and greater difficulty in portability.

For this reason, the usage of GitHub actions could be considered as a bad practice. However, the result of the research shows that this service is still widely used.

Usage of github actions

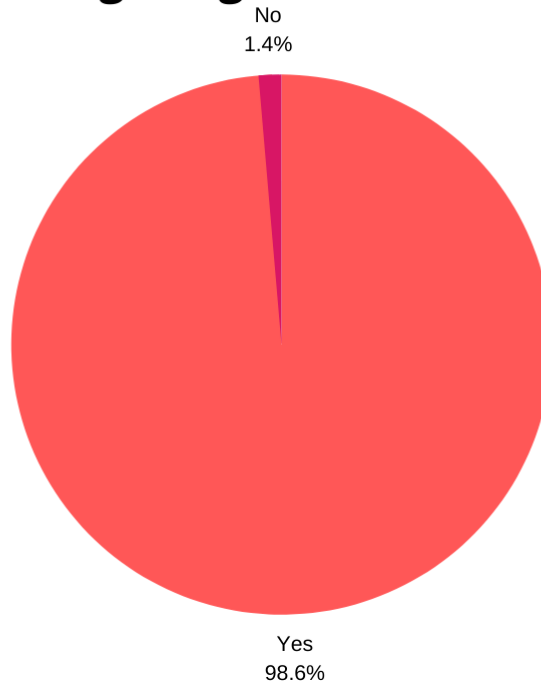


Figure 5.10: Usage of github actions in github workflows

RQ9: What is the usage of comments in GitHub actions?

Usage of comments	Percentage of files	Number of files
Yes	31%	548
No	69%	1220

To answer this question, we analyzed the "number_of_rows_and_comments_per_file.txt" file, and we counted how many projects have 0 rows of comments, so they do not use comments at all. The result is shown in figure 5.11. We can see that 37,8%, so 669 out of 1768 projects use comments, whereas 62,2% of them, so 1099 out of 1768, do not make use of comments. It means that a lot of GitHub workflows do not contain comments inside, which is a bad practice, since the projects are harder to understand.

Usage of comments in github actions files

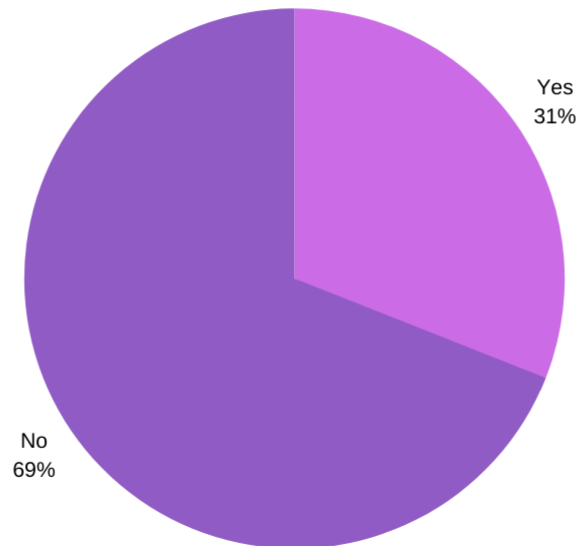


Figure 5.11: Usage of comments in github actions files

Also, we analyzed the density of comments in GitHub actions files, so the ratio between the number of comment lines and the total number of lines per file. The result is shown in figure 5.12.

Density of comments	Number of files
0	1086
0.1	437
0.2	109
0.3	71
0.4	32
0.5	25
0.6	7
0.7	1

In the conclusion of this RQ's argumentation, we can see that the absence of comments is clearly to be considered a bad practice. However, we cannot say that their presence is a good practice since often they are used only to comment portions of code, or for others not meaningful purposes.

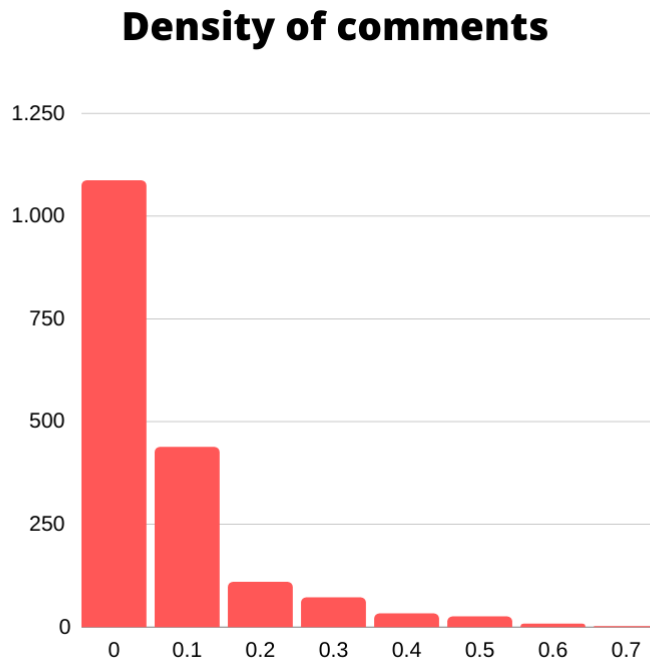


Figure 5.12: Density of comments in gihub actions

RQ10: What is the usage of jobs sequentiality in GitHub actions?

To answer this research question, we considered only 356 files, so the ones with at least 2 jobs. The study found that 219 out of 356 files do not use sequential jobs, so they use the default jobs parallelism. On the other hand, 137 out of 356 files use jobs sequentiality, as we can see in figure 5.13.

We further analyzed these 136 files to find what kind of sequentiality is used. We found that 96 out of 137 files use sequentiality between all of the jobs, whereas others use jobs sequentiality, but not for all of them. The overall degree of parallelism is shown in figure 5.14. As we can see, the mast majority, 1631 out of 1768 of files, have a null degree of sequentialism, which means they only use parallel jobs.

Usage of sequential jobs in github actions files

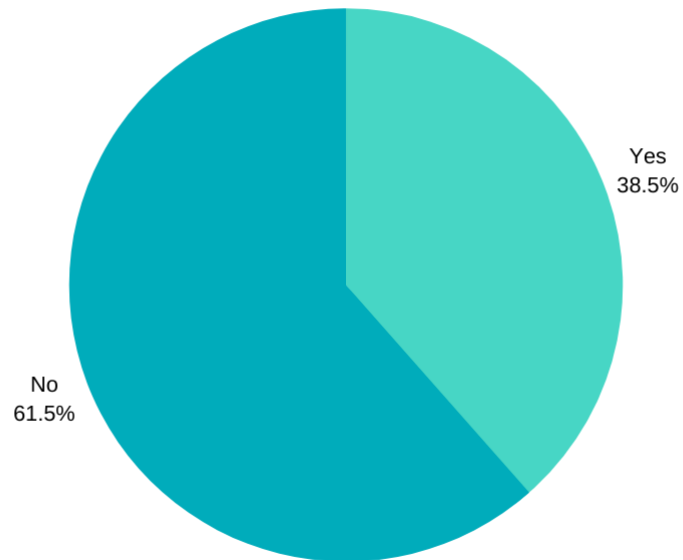


Figure 5.13: Usage of sequential jobs in github actions files

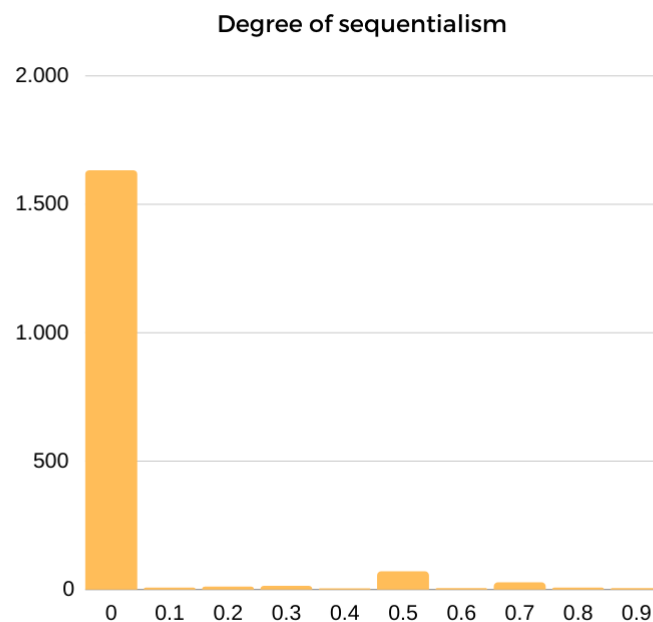


Figure 5.14: Degree of sequentialism in github actions files

Sequential jobs/Total jobs	Number of files
0	1631
0.1	2
0.2	10
0.3	13
0.4	3
0.5	69
0.6	4
0.7	26
0.8	6
0.9	4

For this RQ, we are not able to establish when it has to be considered bad practice since it would require further analysis of the eventually wrong order of job execution.

Indeed, according to [13], the problem dealing with having a sub-optimal ordering of tasks (BP7) in a build process is considered relevant by the survey's respondents. In other words, some build steps should be always performed before others, e.g., integration testing should be scheduled before deployment in the production environment to discover faults earlier. On the other hand, "Independent build jobs are not executed in parallel" (BP5) is another possible bad smell related to this RQ, that we can still properly analyze due to the absence of information.

RQ11: What is the usage of environment variables?

	General usage	Jobs usage	Steps usage
No	1667	1677	1286
Yes	101	91	482

The result of this RQ shows that non-use of environment variables is really a bad practice encountered by practitioners when adopting CI (in our case, across YAML GitHub actions files).

In the table above we can see the number of files not using environment vari-

ables, the mast majority, and the ones that use environment variables, for each of the ways environment variables can be used, listed above. As we can see in the table, for the general usage of environment variables, so the ones that are available to all jobs and steps in the workflow, 1667 out of 1769 files do not use environment variables, whereas only 101 out of 1768 do. Again, 91 out of 1768 use environment variables visible only in all the steps of a single job. Finally, the environment variables in a single step are the more widely used: indeed, 482 out of 1768 files use this kind of environment variable, whereas 1286 do not, that is a lower percentage if compared to the other ones.

In conclusion, in figure 5.15 we show the overall usage of environment variables. As we can see, **1142 out of 1768 files**, so **64,6%**, **do not use environment variables** at all. It confirms that this bad practice is spread when adopting CI practice.

Moreover, we decided to see the average number of global and local variables in GitHub actions YAML files. The result was an average of **0,14 global variables per file**, and **0,38 local variables per file**. This confirms that environment variables are not widely used when adopting CI.

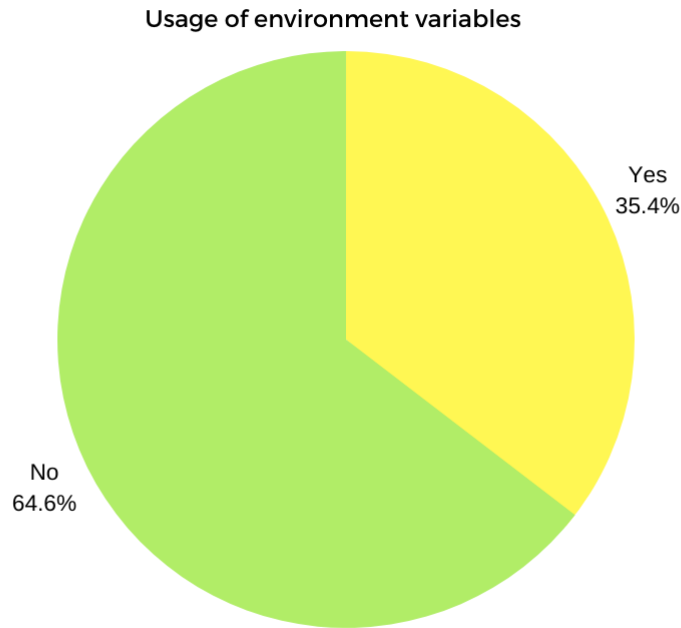


Figure 5.15: Usage of environment variables in github actions files

RQ12: What is the usage of absolute paths in GitHub actions?

Project uses absolute paths?	Percentage of files	Number of files
Yes	9,9%	175
No	90,1%	1593

The result for this RQ was that **175 out of 1768, so 9,9% of files**, have at least one absolute path.

Also, we calculated the average number of absolute paths, and the result was of **0,43 absolute paths per file**.

It must be said that some of the found absolute paths are less "severe" than others, such as `"/dev/null"`, a special file that's present in every single Linux system: however, unlike most other virtual files, instead of reading, it's used to write. Whatever you write to `/dev/null` will be discarded, forgotten into the void. It's known as the null device in a UNIX system.

Usage of absolute paths in github actions files

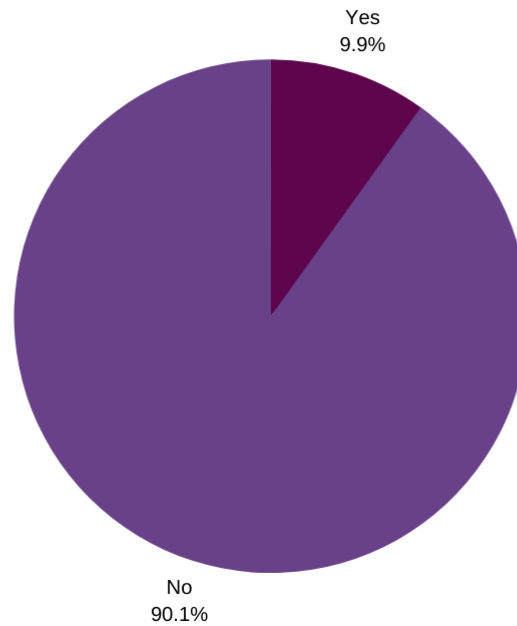


Figure 5.16: Usage of absolute paths in github actions files

Since build configuration files often change over time and their changes induce more relative churn than source code changes, their maintainability is also an important concern.

The high perceived relevance of such a bad smell is justified considering that its presence will unavoidably limit the portability of the build resulting in statements such as "but it works on my machine".

RQ13: How many bad practices can be found in GitHub actions files?

Number of bad practices	Percentage of files	Number of files
0	0,2%	3
1	9,5%	167
2	37,6%	665
3	42,1%	744
4	10,1%	179
5	0,5%	9

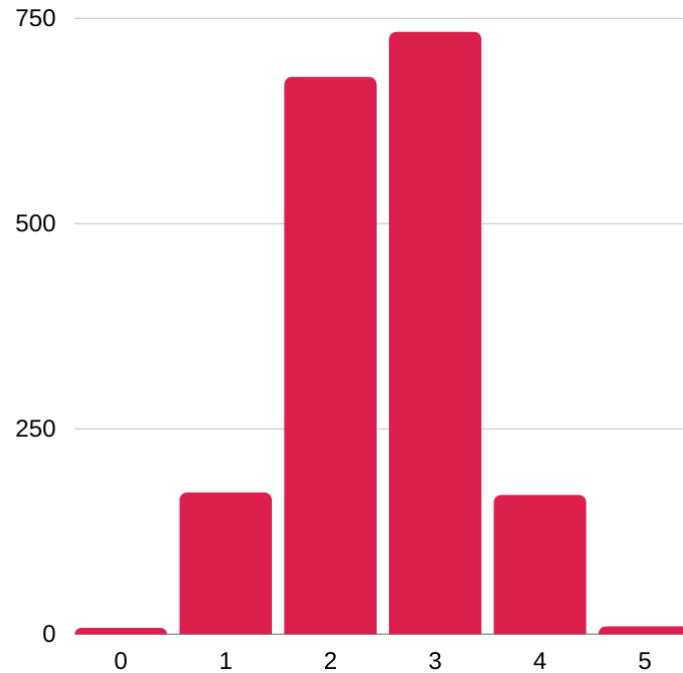


Figure 5.17: Bad practices per file

This section shows the result of the last RQ we addressed, whose purpose is to see how many of the bad practices discussed in this work can be found in the analyzed GitHub actions YAML files.

Note that we do not take into consideration the bad practice related to jobs' sequentiality/parallelism, since, as we said before, we are not able to define when this bad practice takes place. Therefore, the bad practices considered for this RQ are:

- The absence of comments
- The presence of GitHub actions
- The absence of environment variables
- The poor jobs' naming convention
- The presence of absolute paths

The result is shown in figure 5.17. As we can see, few files are containing all of the above bad practices simultaneously (**9 files containing 5 bad practices**). However, they're also a few files not containing bad practices at all (only 0,2%, which means 3 out of 1768 files). Indeed, the mast majority of files containing an average of **3 bad practices (42,1%)**, followed by 37,6% of files containing 2 out of 5 bad practices. The overall result is shown in the table above.

Downstream of the result, we decided to perform further analysis of the best files (0 bad practices) and the worst ones (5 bad practices).

Best files

As is shown in the table, we found 3 "best files", so the ones with no bad practices.

However, they are not really interesting, since they all have only one job, and the average number of rows per file is 31. It is clear that with a low number of rows it is easier to have a few bad practices.

Worst files

A more interesting analysis is the one conducted on the worst files. Here, the average number of lines per files is 86. In the table below we listed their names:

Yaml files with 5 bad practices
ci.yml
tests.yml
test.yml
ci.yml
example-2.yml
main.yml
buildx.yml
deploy.yml
release-sync-rpm.yml

As we can see, a first "bad practice" can be found by analyzing their names, that are not particularly meaningful, especially the ones containing words such as "ci", "main", or "example".

Second, all of them contain at least one job with a name that is not significant.

Below we can find the list of these names:

Job's names in the "worst files"
signing-artifacts
archive
teslamate
grafana
example-2-linux
example-2-mac
example-2-win
MacOS
linux
others
docker
package-source
package-wheel
package
commentTestSuiteHelp

It proves our earlier finding so that the not significant names mainly report the operating system.

5.6 Threats to validity

5.6.1 External

Are our results generalizable for general Github actions usage?

Since we did not analyze a large number of Github actions YAML files contained in open-source repositories, we cannot guarantee that these results will be the same for a larger number of analyzed files.

Moreover, many projects use more than one GitHub actions file, so studying individual files, and not grouping them by the project they belong, could cause several threats to validity.

Are our results generalizable for general Continuous Integration usage?

Some results cannot be extended to the general usage of Continuous Integration practice.

For instance, the [13] states that "The two most positively- assessed bad smells were related to the usage of absolute paths in the build (BM1), and the coupling between the build and the IDE (BM2)". However, the RQ13 result reports that the presence of absolute paths is not a widespread bad practice in GitHub actions files. This result could be because who uses the GitHub action tool to perform CI does not need to use absolute paths in a virtual environment. Nevertheless, it does not guarantee that this bad practice is not widespread in other CI tools.

Chapter 6

User manual

In this chapter we describe all the implementation process, and the procedure needed to execute code that allowed to obtain the described results.

6.1 Installation

As we said before, the project was realized using node.js, JavaScript runtime environment, so the first step is to download the framework.

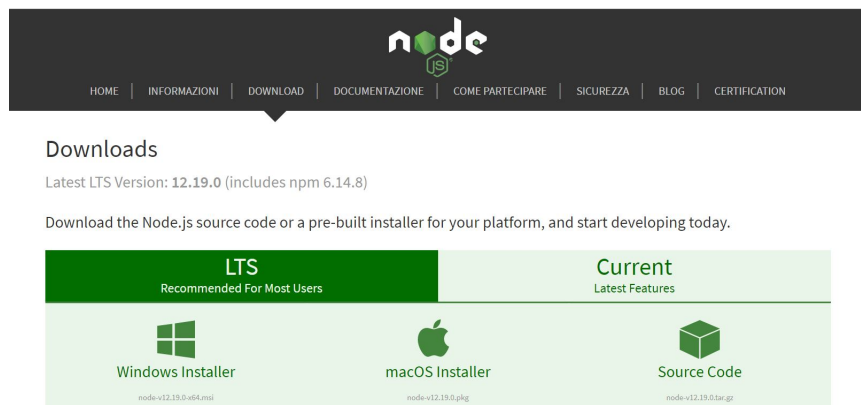


Figure 6.1: Node.js installation

Moreover, npm, a package manager for the JavaScript programming language, was used to install all the necessary libraries, using "npm install" by terminal. In figure 6.2 we can see an example of usage of this tool, in this case to install one of

the most important library: YAML. Similarly, we installed the others necessary libraries, such as:

- fs
- readline
- node-fetch
- axios
- path



```
PROBLEMS  OUTPUT  TERMINAL  DEBUG CONSOLE

Windows PowerShell
Copyright (C) Microsoft Corporation. Tutti i diritti riservati.

Prova la nuova PowerShell multiplatforma https://aka.ms/pscore6

PS C:\Users\eweli\Desktop\unina\tesi\progetto> npm install yaml
```

Figure 6.2: An example of usage of npm install command

6.2 Prerequisites

To be able to execute our first files, that use API Rest Github v3, it is necessary to be in possession of a token. A personal access token for quick access to the GitHub API can be generated by going in your github account developer settings.

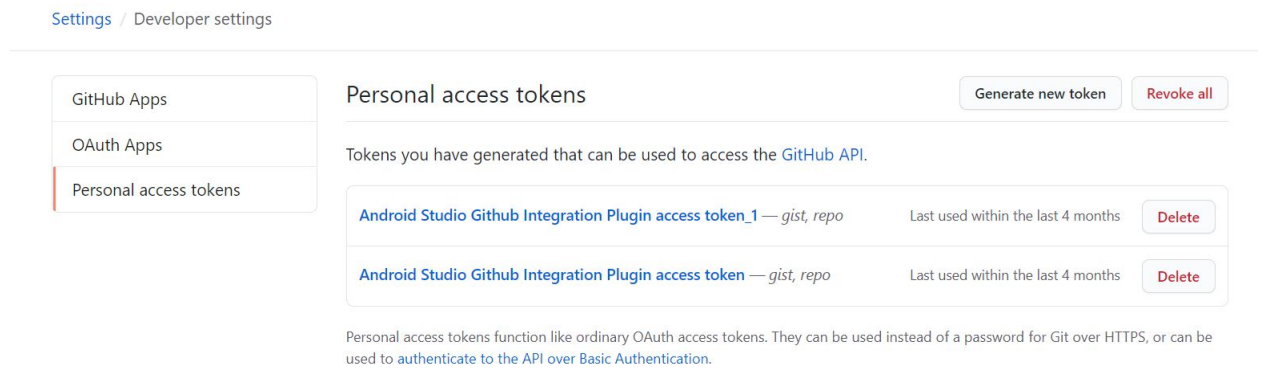


Figure 6.3: Github token generation

6.3 Execution

After Node.js and the used libraries are installed, we can proceed with the code execution, following the order as follows:

- **getRepos.js:** script that queries for the repositories having more than 100 stars and created between 2018 and 2020, reading from `date_ranges_100stars.txt` file and writing repositories names in `repos_100stars_2018-2020.txt` file
- **getYamlPaths.js:** script that reads all the repos found previously from the `repos_100stars_2018-2020.txt` file, and checks if they contain yaml files (if yes, saves their paths in `yaml_paths_2018-2020_100stars.txt` file)
- **downloadYamls.js:** script that reads the yaml paths from `yaml_paths_2018-2020_100stars_5.txt` file, and downloads them, putting them in the proper directory, based on what kind of CI tool they adopt.
- **getJobs.js:** script that writes in `jobs_github_numbers.txt` file the number of jobs for every file contained in `github_yamls` folder.

- **getTriggerEvents.js:** script that writes all the trigger events present in github files in `trigger_events.txt` file.
- **classifyJobs.js:** classify all the jobs and writes the result in `jobs_per_file_classified.txt` file.
- **getSteps.js:** script that calculates the number of steps for every job, and writes the results in `jobs_steps_number.txt` file.
- **getRowsPerJob.js:** script that calculates the number of rows per job of a file and writes it in `rows_per_job.txt` file.
- **analyzeJobsNames.js:** script that reads from the `jobs_id_name_file.txt` the job's names and ids, and writes to the `jobs_names_significance.txt` file all the jobs, specifying if they are significant or not.
- **getActions.js:** script that analyzes every file contained in `github_yamls` folder, and writes in `use_of_actions.txt` file if they use or not Github actions.
- **getRowsComments.js:** script that counts the number of comment rows per file, and writes the results in `number_of_rows_and_comments_per_file.txt` file.
- **getNeeds.js:** script that writes in `jobs_needs.txt` file the number of "needs" occurrences per file, and the number of jobs in that file.
- **getEnvVariables.js:** script that writes per each file if it uses environment variable in `use_of_env_variables.txt` file.
- **getPaths.js:** script that checks, for each YAML file, if they contain an absolute path, and writes in the output file, `paths.txt`, the number of the absolute paths detected and the paths themselves.

- **getBadPractices:** script that counts how many of the previously analyzed bad practices are present in each Github YAML file, and writes the result in bad_practices.txt file.

RQ	Script .js	Input file	Output file
1	getRepos getYamlPaths	date_ranges_100stars.txt	repos_100stars_2018-2020.txt
2	downloadYamls	yaml_paths_2018-2020 _100stars.txt	files to github_yamls folder
3	getJobs	each file from github_yamls folder	jobs_github_numbers.txt jobs_id_name_file.txt
4	getTriggerEvents	each file from github_yamls folder	trigger_events.txt
5	classifyJobs	jobs_id_name_file.txt	jobs_per_file_classified.txt
6	getSteps getRowsPerJob	each file from github_yamls folder	jobs_steps_number.txt rows_per_job.txt
7	analyzeJobsNames	jobs_id_name_file.txt	jobs_names_significance.txt
8	getActions	each file from github_yamls folder	use_of_actions.txt
9	getRowsComments	each file from github_yamls folder	number_of_rows_and_ comments_per_file.txt
10	getNeeds	each file from github_yamls folder	jobs_needs.txt
11	getEnvVariables	each file from github_yamls folder	use_of_env_variables.txt
12	getPaths	each file from github_yamls folder	paths.txt
13	getBadPractices	each file from github_yamls folder	bad_practices.txt

Chapter 7

Final discussion

Our study aimed to perform an analysis of bad practices in Github CI tool, according to the bad practices stated by a previous study.

First of all, we studied the breakdown of CI tools in open-source projects. We found out, as a confirm of previous studies, that amongst all the CI/CD tools, Travis CI is undoubtedly one of the most popular choices. Initially, it was created for open-source projects but with time, the tool has also migrated to close source projects. Travis CI is also one of the early players in the CI/CD tools market. The tool is written in Ruby and is developed & maintained by the Travis CI community. Travis CI was earlier available only for GitHub hosted projects but now it also supports Bitbucket hosted projects. It is available for Linux, macOS, and Windows (early stage) operating systems, and it is free of charge for every open-source project. For using Travis CI, you should have an account on GitHub or Bitbucket. There is no installation required and you can get started by simply signing up and adding a project. Below we report the salient features of Travis CI, that make of it the most popular choice as CI tool:

- Free for testing open-source applications.
- Available for Continuous Delivery (CD) and Continuous Integration (CI).

- Available for macOS, Linux, Windows (early stages).
- Supports around 30 different programming languages like Ruby, Perl, Python, Scala, etc.
- Can be configured after adding `.travis.yml` file (i.e. YAML format text file).
- Supports integration with external tools.
- Supports build matrix feature to accelerate the project execution.

Also, Travis CI supports parallel testing. It can also be integrated with tools like Slack, HipChat, Email, etc. and get notifications if the build is unsuccessful. Developers can speed up their test suites by executing multiple builds in parallel, across different virtual machines. The ‘build matrix’ feature offered by Travis CI allows developers to break down a build into assorted parts and thus speed up the suites.

However, we did not perform a further analysis of this tool, since there are already many studies about Travis. Indeed, we analyzed the second most used tool, the one provided by Github. GitHub Actions is a tool within GitHub that enables continuous integration and a wide range of automation. It supports the three major operating systems, Windows, MacOS and Linux, and you can run any programming language supported by those. Apart from being triggered by pull requests and commits, actions allows you to respond to any GitHub event. It allows you to trigger certain GitHub Actions workflows (including open source actions) based on:

- the creation of issues
- comments
- the joining of a new member to the repository

- changes to the GitHub project board.

Actions comes with a strong level of integration with GitHub, removing the requirement of an additional vendor for CI.

Further more, our study shows that tools like CircleCi, AppVeyor or Werker are not used that much indeed. This result is similar to the one provided by a study from 2016 [10]. However, the percentage of usage of different tools changes, due to a subsequent introduction of Github actions tool. As follows we show the two results in comparison:

Usage by CI service				
Travis	CircleCI	AppVeyor	CloudBees	Werker
90,1%	19,1%	3,5%	1,6%	0,4%
12528	2657	484	223	59

Table 7.1: Usage by CI service by Hilton

Usage by CI service				
Travis	GitHub actions	CircleCI	Appveyor	Werker
52,2%	27,3%	10,4%	9,5%	0,6%
2053	1073	408	373	23

Table 7.2: Usage by CI service in our study

Next, since there are not previous studies, to the best of our knowledge, about Github actions tool, we have focused our analysis on this tool, with an analysis of the general usage of this tool first, and then continue with an analysis of bad practices that can be encountered when adopting this CI tool. Since popular projects are more likely to use CI, according to [10], we analyzed only the GitHub repositories with more than 100 stars. About the general usage of this tool, we found out that:

- the vast majority of YAML files have only one job
- the workflows are executed mainly after a push event

- they are mostly used to automate the building process
- they have between 3 and 5 steps in average, and an average of 30 lines per job

Nevertheless, we are mostly interested in analyzing the bad practices related to the use of this tool, or more generally of any CI tool. To do this, we took into consideration *An Empirical Characterization of Bad Practices in Continuous Integration* by Zampetti et al. [13]. In particular, we focused on the bad practices that can be detected by only analyzing the YAML code.

According to our study, 16% of build jobs have not a significative name. The usage of suitable naming conventions inside the build scripts was considered relatively important by the respondents to the survey of the study [13].

Also, we found out that the most widespread bad practices among the Github actions YAML files are:

- absence of comments
- lack of usage of environment variables

However, the lack of usage of environment variables received more negative than positive assessments, so it is not considered that relevant.

Though the two most positively-assessed bad smells concerning Build Maintainability were related to the usage of absolute paths in the build and the coupling between the build and the IDE, according to our results, this bad smell is not that widespread when adopting Github actions tool. Indeed, only 9,9% of files make use of absolute paths.

In conclusion, we can state that the bad practices considered relevant are not widespread in Github actions workflows, whilst the one widespread are not considered that relevant by the previous studies in literature.

Chapter 8

Conclusions and future directions

8.1 Conclusions

CI has been rising as a big success story in automated software engineering. However, when adopting Continuous Integration (CI) practitioners often face bad practices.

To study the usage of CI and related bad practices, we first analyzed 43193 open-source projects from GitHub, to find how much this practice is widespread along with open-source projects and see which tools are mainly used. Our results show that even though Travis is so far the most used CI tool, there are good reasons for the rise of usage of Github actions CI tool.

Secondly, we conducted a further analysis of the usage of bad practices when adopting the Github CI tool. To achieve this goal, we analyzed 1768 YAML files using Github actions as a CI tool. We based our questions about bad practices on some previous studies, such as [13]. In particular, we faced bad practices that can be detected by static analysis.

Overall, we investigated 13 research questions, grouped into 2 themes: one about the general usage of CI in GitHub actions files, and another one about bad practices.

The results of our study show that some bad practices are really frequent when adopting the Github actions CI tool, such as absence of comments, poor naming convention for build jobs, or absence of environment variables, and some others that are not much widespread (presence of absolute paths).

8.2 Future directions

The research presented in the previous chapters leaves many promising avenues open for future research. This section looks at some of them:

- **Deepen the questions already asked:** as a future development, we might investigate further about the questions already asked, such as usage of sequential jobs, and when this results in bad practice, or investigate the nature of the comments present in CI workflows.
- **Enlarge list of questions:** we might address other questions, especially about other bad practices that developers can face when adopting CI tools.
- **Dissemination of bad practices in other tools:** We focused our attention on the GitHub native CI tool. However, we could extend the study to the other CI tools widespread in open-source projects.
- **Validate the answers by interviewing the developers:** Finally, one might consider validating the results obtained from the experiment by interviewing the developers based on their experience.

Bibliography

- [1] GitHub actions. <https://docs.github.com/en/actions>.
- [2] GitHub API v3. <https://developer.github.com/v3/>.
- [3] Yaml npm library. <https://eemeli.org/yaml/#yaml>.
- [4] Oren Ben-Kiki, Clark Evans, and Ingy döt Net. Yaml ain't markup language (yamlTM) version 1.2. 2001.
- [5] Nathan Cassee, Bogdan Vasilescu, and Alexander Serebrenik. The silent helper: the impact of continuous integration on code reviews. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 423–434. IEEE, 2020.
- [6] Valerio Cosentino, Javier L Cánovas Izquierdo, and Jordi Cabot. A systematic mapping study of software development with github. *IEEE Access*, 5:7173–7192, 2017.
- [7] Wagner Felidré, Leonardo Furtado, Daniel A da Costa, Bruno Cartaxo, and Gustavo Pinto. Continuous integration theater. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10. IEEE, 2019.
- [8] Taher Ahmed Ghaleb, Daniel Alencar Da Costa, and Ying Zou. An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering*, 24(4):2102–2139, 2019.

- [9] Georgios Gousios and Diomidis Spinellis. Mining software engineering data from github. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 501–502. IEEE, 2017.
- [10] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 426–437. IEEE, 2016.
- [11] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. An in-depth study of the promises and perils of mining github. *Empirical Software Engineering*, 21(5):2035–2071, 2016.
- [12] K.Gallaba. Improving the robustness and efficiency of continuous integration and deployment. 2019.
- [13] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald Gall, and Massimiliano Di Penta. An empirical characterization of bad practices in continuous integration. *Empirical Software Engineering*, 25(2):1095–1135, 2020.
- [14] Yang Zhang, Huaimin Wang, Yiwen Wu, Dongyang Hu, and Tao Wang. Github’s milestone tool: A mixed-methods analysis on its use. *Journal of Software: Evolution and Process*, 32(4):e2229, 2020.