Tackling DevOps and CI within ML projects

by

Dhia Elhaq Rzig

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer and Information Science)
in the University of Michigan
2024

Doctoral Committee:

Associate Professor Foyzul Hassan, Chair
Professor Bruce Maxim
Associate Professor Khouloud Gaaloul
Professor Hafiz Malik


DEARBORN

Dhia Elhaq Rzig

dhiarzig@umich.edu

ORCID iD: 0000-0002-2757-9257

DEDICATION

I dedicate this work to my family, my friends, and my pets, whose love and support have helped me navigate the tumultuous ocean of the Ph.D life

ACKNOWLEDGEMENTS

First, I would like to acknowledge the help, support and knowledge given to me by the professors I met during my academic path, the mentors I was blessed with, and my advisor, Dr. Foyzul Hassan. I stand on the shoulders of giants because of your help and kindness.

I would also like to acknowledge the University of Michigan-Dearborn, specifically the College of Engineering and Computer Science and the department of Computer and Information science for their support of my Ph.D., especially through the grants that have allowed me to pursue this degree and present my work within the International Symposium on Software Testing and Analysis 2023. Furthermore, I would like to thank the NSF for their generous support through the grant number 2152819 awarded to my advisor which helped support my research, and the travel grant that allowed me to attend the Empirical Software Engineering and Measurement 2023 conference, and last but not least, SIGSOFT for their grant that allowed me to attend the International Symposium on Software Testing and Analysis 2023.

Finally, I would like to acknowledge my co-authors without whom the different works within this thesis would not be accomplished.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

LIST OF TABLES

TABLE

LIST OF ACRONYMS

GHA GitHub Actions

ML Machine Learning

CI Continuous Integration

OSS Open Source Software

API Application Programming Interface

LLM Large Language Models

CI/CD Continuous Integration/Continuous Deployment

SE Software Engineering

ABSTRACT

Machine Learning (ML), including Deep Learning (DL), based systems are emerging technologies applied to solve complex problems like autonomous driving and recommendation systems. To enhance the quality and deliverability of ML-based applications, the software development community is adopting DevOps practices. However there is a lack of insight about how DevOps in the context of ML projects. This lack of insight has shaped the overarching goal of my thesis: **To perform a use-driven and empirical-data validated discovery and resolution of problems related to DevOps in ML projects**. This thesis is split between two phases: the first phase is the exploration phase, where we rely on empirical studies to understand the current state of DevOps in ML projects, and the second phase is the resolution phase, where we propose solutions to the problems identified in the exploration phase.

The first obstacle to achieve this goal was a lack of knowledge about DevOps adoption trends, maintenance efforts, and benefits in ML projects. Hence my first research project was a large-scale empirical analysis on 4031 ML projects to quantify DevOps adoption, maintenance effort, and benefits. These ML projects were categorized into ML-Tool and ML-Applied projects, where Tool projects are libraries, frameworks, and tools for ML development, and Applied projects are projects that apply ML to solve real-world problems. Additionally, we performed the same analysis on 4076 Non-ML projects to contextualize the results. We found that ML projects, especially ML-Applied projects, have slower, lower, and less efficient DevOps adoption compared to traditional software projects. Despite this, adopting DevOps in ML projects correlates with increased development productivity, improved code quality, and reduced bug resolution time, especially in ML-Applied projects.

After identifying the DevOps adoption trends in ML projects, we further investigated Continuous Integration (CI), a subset and the central tenant of DevOps, in ML projects. CI tools automate repetitive tasks such as building, testing, and deployment, which are essential for ML projects. However, unlike traditional software, the adoption and issues of CI in ML projects have not been empirically studied. This study compares CI adoption between ML and Non-ML projects using `TraVanalyzer`, the first Travis CI configuration analyzer, and a CI log analyzer. Our findings show Travis CI is the most popular tool for

ML projects, though their CI adoption lags behind Non-ML projects. ML projects using CI focus more on building, testing, code analysis, and deployment. CI in ML projects faces varied build-breakage reasons, with testing-related problems being the most frequent.

This project helped us gain a better picture of CI in ML from a static point of view, but were interested in gaining a more dynamic understanding of how CI evolves in ML projects. While several works discussed how CI/CD configuration and services change during their usage in traditional software systems, there is very limited knowledge of how CI/CD configuration and services change in ML project. To fill this knowledge gap, we manually analyzed 701 commits from 578 open-source ML projects, and devised a taxonomy of 14 co-changes in CI/CD and ML components. We also expanded `TraVanalyzer` to support GitHub Actions in order to identify frequent CI/CD configuration change patterns in 38,982 commits encompassing Travis CI and GitHub Actions changes. We found that most changes in Travis CI and GitHub Actions were related to build policy, with fewer changes related to performance, maintainability, and infrastructure. We also identified some CI bad practices, such as the direct inclusion of dependencies in CI files, and we found that experienced developers were more likely to modify and maintain CI/CD configurations.

After having performed this exploration phase, we focused on 2 common problems in ML DevOps: the difficulty of CI Migration between platforms, and issues with ML-testing and ML-issue resolution. Concerning CI Migration, based on existing research and findings concerning CI, we believe that the efficiency of CI systems is a crucial factor for development velocity. And as a result, developers often migrate from their existing CI systems to new CI systems with more features like matrix building, better logging support, etc. We also noticed trends of this migrations between Travis CI and GitHub Actions, two popular CI systems, which were also confirmed by other studies. However, this process is challenging and error-prone due to limited knowledge and complex configurations. To address this, we propose CIMig, which uses Apriori Rule Mining and Frequent Tree mining to automate CI system migrations. Our automatic evaluation using a set of 251 project shows CIMig achieves a 70.82% success rate for GitHub Actions and 51.86% for Travis CI, with comparable performance to manual-mapping-based tools. Our user-study evaluation also revealed ratings competitive with the manual-mapping-based tools. Unlike other tools, CIMig supports bidirectional migrations and relies on technology-agnostic techniques, making it versatile and beneficial for developers.

Finally, we shift our focus to the emerging sector of Large Language Models (LLMs). The advancements in LLMs has led to their swift integration into application-level logic using prompts, referred to as Developer Prompts. Through our previous works, we noted a severe lack of application of ML-specific testing practices, hence putting into doubt whether these

Dev Prompts are being properly tested for vulnerabilities, bias, and performance. Further complicating matters, unlike traditional software artifacts, Dev Prompts blend natural language instructions with artificial languages such as programming and markup languages, thus requiring specialized tools for analysis. In our study of 2,173 Dev Prompts, we found that Dev Prompts 3.46% contained one or more forms of bias, and that 10.75% were vulnerable to prompt injection attacks. We introduce *PromptDoctor* to addressed these issues, using which we de-biased 68.29%, hardened 41.81%, and improved the performance of 37.1% of the flawed prompts. We developed a PromptDoctor VSCode extension and we plan to extend PromptDoctor for easy integration with other IDEs and CI/CD pipelines.

CHAPTER 1

Introduction

Machine learning, a subset of Artificial Intelligence, has enabled computers to learn from data and make predictions or decisions without the need for explicit instructions, thus making previously inscrutable problems solvable. Indeed, ML has become central to various applications, such as natural language processing, computer vision, speech recognition, and many more. Machine learning also has the potential to transform various domains, such as healthcare, education, business, and science, by providing new insights, solutions, and innovations. And in certain ways, it already has. ML has been used in application such as Alzheimer's disease diagnosis [186], Blood glucose prediction in diabetics [211], Autonomous-driving cars [51], Loan approval prediction [237], etc. Furthermore, The Worldwide Developer Population and Demographic Study 2019 [69] estimates that approximately 7 million developers have used ML in their development activity, and expects another 9.5 million developers to use ML in the next twelve months. Although ML-based approaches are becoming widely adopted by the industry as well as the research community, one major challenge remains: the integration of ML components in complex production systems and processes while maintaining their reliability and efficiency in the context of continuously evolving ML projects.

To improve the software delivery process, a closer collaboration between the development and operations teams, known as DevOps [181] has become popular within the software engineering community due to the many advantages it brings to software engineering processes. DevOps is a modern software engineering paradigm that brings changes to production processes with the approach of automating the building, testing, code analysis and deployment of software.

While DevOps practices are slowly becoming more common and standardized for traditional software products [45], the state of DevOps within ML-based projects remains largely unknown. This has shaped the over-arching goal of this thesis as such:

Goal

To perform a use-driven and empirical-data validated discovery and resolution of problems related to DevOps in ML projects

This thesis is divided into two main phases. The first phase is Exploration, where we gain an understanding of the current state of affairs of DevOps in ML projects, composed of projects detailed in Chapter 2, Chapter 3, and Chapter 4. The second phase is Resolution, where we attempt to resolve some of the problems we identified via the Exploration phase, composed of projects detailed in Chapter 5, and Chapter 6.

When starting with the exploration phases, we realized there were no empirical studies into the state of DevOps, we tackles this gap in knowledge via Chapter 2, where we perform an empirical study on the adoption of DevOps practices within ML projects, the efforts required to adopt them, and the benefits they bring.

Continuous Integration is a subset, and the central tenant of DevOps, and it is a software development process for shared repositories that automatically integrates the changes committed by their developers. CI is considered a central pillar of DevOps that helps to automate the building, testing, and deployment of software. CI allows its adopters to catch bugs earlier, increase the frequency of their releases, and integrate pull requests faster [148]. Its adoption has grown from 40.27% in 2016 [148] to 68% within larger teams in 2018 [73].

Similar to traditional software, ML projects rely on iteration within their development, and the automation prowess of a CI system may be a great fit for these projects' iteration needs. However, it's notable that most CI tools were conceived before ML project development became mainstream, and that both CI tools and ML projects have their specific problems. For example, debugging CI build failures and errors can be non-trivial due to complex logs [334], and ML projects require new development processes and practices such as data engineering and model management [196], or require a different approach to existing processes in comparison to traditional software, such as the example of traditional testing being ineffective on ML projects [170]. Yet, there is a gap in research concerning the adoption of CI within ML projects, the tasks performed by CI within them, as well as the problems CI tools face when they are used in these projects. Hence, and we wanted to reach a better understanding of CI and the CI-related issues ML projects encounter. Within Chapter 3, we build on the results of the previous chapter and delves deeper into some of them by Characterizing the usage of CI tools in ML projects. Furthermore, how these systems evolve over

time, and who is responsible for maintaining them, is also unknown. We perform a more dynamic analysis in Chapter 4 where we discuss the project: Empirical Analysis on CI/CD Pipeline Evolution in Machine Learning Projects. Overall, the results of these chapters will provide a better understanding of the CI-related issues ML projects face, and will help us to identify the challenges and opportunities for improving the CI process in ML projects.

Moving on the Resolution phases, two main issues we noted in the Exploration phases are the lack of tools to help developers migrate their CI systems to new platforms, and issues regarding how developers test and improve their ML components.

Concerning CI Migration, while there are a variety of CI tools available; Travis CI and GitHub Actions are currently the most popular CI tools for Open Source Software (OSS) projects in general [125, 148], and for ML-based projects [276]. Travis CI has been the early market leader, but GitHub Actions has recently become a top contenderwith growth fueled in part by migrations away from Travis CI, as 87.1% out of repositories that discontinued Travis CI usage migrated to GitHub Actions (GHA) [125]. However, this migration is far from trivial, as detailed in the research of Mazrae et al. [272]. The empirical analysis pointed out that the CI migration process is slow and error-prone due to the steep learning curve, fundamental differences between the source and target CI systems, and the trial-and-error-based migration of CI systems. It's also notable that, GitHub Actions Importer [117], relies on manual mapping and lacks support for some features, such as the migration of secrets, which contain private information such as authorization tokens, and the migration of certain job properties [112]. Focusing on research works, the majority of existing migration works focus on analyzing and migrating source code from one programming language to another [88, 83, 219, 15, 231], with few works concerning the analysis and migration of configuration code [144, 124, 321, 260], and none tackling the automatic migration of CI configuration code. We resolve this issue in Chapter 5 by proposing CIMig, an automated technology-agnostic approach to migrating CI systems that would simplify the migrations of CI infrastructures of ML projects to new platforms.

Concerning the second issue, ML components testing, we focus on LLM-powered projects as a new and emerging type of ML projects. These projects require new development and validation processes. This is especially important as these projects rely on a new type of software artifact: Prompts. Prompts are a mix of natural language and code that are used to guide the model in generating the desired output. The quality of the prompt is crucial to the quality of the model's output. However, writing prompts is a non-trivial task, as it requires a deep understanding of the model's architecture, the dataset, and the task at hand. Furthermore, the quality of the prompt can be subjective, and it can be hard to evaluate its quality. This is especially important as the quality of the prompt can have a significant

impact on the model's output. However, there is a lack of tools that can help developers validate and improve their prompts. We believe that building tools that allow developers to evaluate and improve their prompts within the context of software engineering and CI processes can help improve the quality of the prompts and the models that are generated from them. We address this issue in Chapter 6 by proposing PromptDoctor, a tool that would help developers detect and fix various issues with their prompts.

Finally, Chapter 8 concludes this report and discusses planned future work .

CHAPTER 2

An Empirical Study on ML DevOps Adoption Trends, Efforts, and Benefits Analysis

This work was published in ELSEVIER Information and Software Technology (IST) in August 2022, and accepted in the journal-first track of ESEM 2023, the 17th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement.

2.1 Introduction

Recently, Machine Learning (ML), including Deep Learning (DL), has become prevalent with many applications: Alzheimer's disease diagnosis [186], Blood glucose prediction in diabetics [211], Autonomous-driving cars [51], Loan approval prediction [237], etc. The Worldwide Developer Population and Demographic Study 2019 [69] estimates that approximately 7 million developers have used ML in their development activity, and expects another 9.5 million developers to use ML in the next twelve months. Although ML-based approaches are becoming widely adopted by the industry as well as the research community, one major challenge remains: the integration of ML components in complex production systems and processes while maintaining their reliability and efficiency in the context of continuously evolving ML projects.

To improve the software delivery process, a closer collaboration between the development and operations teams, known as DevOps [181] has become popular within the software engineering community. DevOps is a modern software engineering paradigm that brings changes to production processes with the approach of automating the building, testing, code analysis and deployment of software. A recent GitHub study [109] discovered that highly-performing DevOps teams recover from downtime 96 times faster, have a 5 times lower failure rate, and a 46 times more frequent deployment rate. While DevOps practices are slowly becoming more common and standardized for traditional software products [45], the state of DevOps within ML-based projects, the advantages, and the challenges it brings, still require more

study within the research community.

Recently, there have been many works focused on ML DevOps support. MLFlow [50] and Amazon SageMaker [292] were designed to improve the workflow of ML project development, which involves the data collection, data preparation, model definition and training, and results-testing [196]. Package managers such as Spack [102] and EasyBuild [154] were conceived to allow the automatic rebuilding of ML models. Container-based technology such as Docker [74] and Kubernetes [130] has proven apt for shareable models. Aguilar et al. [264] proposed Ease.ml/CI for continuous integration (CI) and data management within ML projects. Fursin et al. [98] proposed CodeReef to perform benchmarking for ML projects and enable their reusable automation. However, the majority of these tools are still premature, require an important development effort, and can only be used in conjunction with specific ML technologies or frameworks [347, 194, 98]. Prior research [98] also identifies that workflows using these solutions are not easy to put into practice. Moreover, very little is known about ML projects' DevOps adoption and the difficulty of maintaining correctly functioning DevOps tools within them. This motivates our large-scale study on DevOps tools' adoption within ML projects, their maintenance effort and goals, and the benefits they bring. In order to obtain more information about these aspects, we defined the following research questions:

1. What are the current and historical adoption rates of DevOps Tools for ML and Non-ML projects?

2. What are the maintenance efforts and goals associated with DevOps tools across the different project categories ?

3. What are the advantages of adopting DevOps tools across the different project categories?

In this empirical study, we conducted a large scale analysis on 4031 ML projects that we manually curated from the dataset by Gonzalez et al. [128]. We also performed the same analysis on the 4076 Non-ML projects from the same dataset [128] for comparative purposes. Our main contributions through this paper can be summarized as follows:

- Characterization of the current and historical adoption of DevOps tools within a subset of popular Open-source ML projects. Indeed, we found that ML Tool projects, which are general purpose projects meant for use by other developers, had similar current and historical DevOps tools' adoption to Non-ML projects, while ML Applied projects, which are specific-purpose projects meant for use by other developers and end-users, had a lower and slower DevOps tools adoption in comparison

- An empirical analysis of the development effort in regards to employing DevOps tools for different types of ML projects. We believe that more DevOps-related development effort is invested within ML Tool projects than ML Applied projects, and that the adoption of certain DevOps tools within these project categories is linked to a larger effort invested by their development teams.

- Characterization of the common goals behind the changes in DevOps configuration files and their other accompanying changes ML projects. We found that ML Tool and Non-ML projects achieve more Bug fixes than ML Applied projects. Both in ML Tool and Non-ML project, this increase in bug fixes is correlated with their adoption of DevOps tools such as Test and Code analysis tools, while this correlation was not found within ML Applied projects. A small percentage of DevOps-altering commits were found to have Build fixes as a goal, and the majority of them were concerned with other miscellaneous changes.

- An empirical analysis of the improvements in the development process resulting from the usage of DevOps tools within ML projects. Across all categories of projects, we found that the adoption of one or more DevOps tools was positively correlated with an increase in commit frequency, merge frequency, code quality, and a reduction of the average issue resolution duration.

The rest of this paper is organized as follows: We start by discussing related works in Section 2.2. After that, in Section 2.3, we discuss the methodology of our analysis, which includes data set selection, DevOps tools classification, and the methods of analysis we used to answer our Research Questions. Section 2.4 presents the results of the empirical analysis within our study and Section 2.5 discusses the possible implications of our study. Finally, we discuss the threats to validity and our conclusion in Section 2.6 and Section 2.7, respectively.

2.2 Related Work

As DevOps became a modern software engineering paradigm, it received growing attention from the research community [181, 308, 158, 84]. Luz et al. [193] compared different approaches of adopting DevOps and identified the main concerns of DevOps. They believe that collaboration is an important DevOps concern in addition to the more common and equally important tool usage. However, this work mainly focused on interview outcomes rather than an empirical analysis of DevOps as adopted by the software projects. Moving on to guidance on adopting DevOps, Leite et al. [181] analyzed DevOps within general-purpose

software projects from a multitude of facets. They developed conceptual maps that described DevOps and linked them to engineering and management perspectives.

McIntosh et al. [207] analyzed Build files, a type of DevOps configuration files, in order to estimate the effort invested by developers to maintain functioning Build systems in 9 open-source and 1 closed source projects. They found that the level of correlation between source files and build files is linked to a project's programming languages. But, their work only covered a limited set of C and Java projects and a handful of build tools, such as Make and ANT. This means that their findings may not apply to projects with other programming languages and other Build and DevOps tools.

However, none of these aforementioned works focus specifically on ML projects or considered them as a specific project-category. We consider this an oversight due to the fundamental differences between ML and Non-ML software projects. Unlike Non-ML projects, ML projects are not given an explicit solution. Instead, they attempt to solve a problem by analyzing data, testing their findings, evaluating their results, and iterating on these phases. Furthermore, they require new development processes and practices such as data engineering and model management [196, 297, 167], follow different collaboration strategies between their collaborators [128, 223], and may require different approaches to existing software development processes in comparison to traditional software, such as the example of Non-ML software testing being ineffective on ML projects [169].

Lwakatare et al. [196] outlined some of the problems teams face while attempting to integrate ML workflows within DevOps processes, such as the inadequacy of existing code versioning tools for ML artifacts management, and proposed alternative processes to employ DevOps in ML projects. Yet, their work relied on existing literature and expert knowledge when discussing DevOps adoption problems within ML projects, and did not perform empirical analysis to validate the actual factors behind ML projects' success or failure at adopting DevOps.

To analyze ML project development aspects, the work of Gonzalez et al. [128] conducted a large-scale empirical study of Open-source ML Tools (700) and Applications (4,524) hosted on GitHub. For comparative purposes, they also analyzed 4,101 Non-ML projects. Their work provided insight into collaboration and autonomy rates in development teams and identified ML Applied projects as the most autonomous, Non-ML projects as less autonomous, and ML Tool projects as the least autonomous. However, we uncovered problems with their data-set in regards to the selection and classification of ML projects. Furthermore, their work is more interested in analyzing development practices and collaboration aspects of the projects rather than analyzing their DevOps adoption and DevOps practices.

Focusing more on the intersection of DevOps and ML projects, Karlaš et al. [169] dis-

cussed the shortcomings and the lack of support of existing CI tools of ML projects in practice. Their work proposed implementation details that attempted to solidify and build on existing theoretical concepts concerning CI systems for ML projects. However, their work did not consider other aspects of DevOps processes such as Code Analyzers, Build systems, Deployment Automation, etc.

In contrast to existing works, our goal within this paper is to analyze the adoption rates and trends of all DevOps components such as Code Analyzers, Build systems, Continuous Integration systems, etc., within ML projects, to characterize their associated maintenance efforts, goals, as well as the advantages they bring to the projects that adopt them.

2.3 Methodology

2.3.1 Data Set Collection

For this work, our goal was to analyze DevOps tools' adoption within a set of active and currently developed Machine Learning (ML) software projects, referred to as *ML* projects, and a comparison set of Software Projects that do not use ML, referred to as *Non-ML* projects. However, preparing a data-set of ML projects and Non-ML projects is effort-intensive, and not the goal of this work. Initially, we opted for a recent dataset proposed by Gonzalez et al. [128] for our analysis. This dataset was supposed to contain 5224 ML projects and 4101 non-ML projects for comparative purposes. However, we found several problems with it such as the inclusion of toy projects, learning guides and other types of projects that were supposedly manually removed from it, as well as the misclassification between the two subsets of ML projects. To resolve this problem, two authors re-curated the ML projects by reading their descriptions on their main GitHub page, and any websites linked to by that page. The resulting new dataset we used within this work contained:

1. **1116 ML Tool** projects: frameworks and libraries such as Tensorflow, which can be used by developers to solve a variety of problems. These projects are generally only usable via an API.

2. **2915 ML Applied** projects: Applications and libraries that use ML components or libraries from the ML Tool projects, to solve a specific problem. FaceSwap is an example of an application and Document-Classifier-LSTM is an example of a library. These projects may offer a combination of a UI and an API.

3. **4076 Non-ML** projects: A comparison set of classic software projects that don't use ML. These projects may offer a combination of a UI and an API.

In addition, we used the GitHub API [251] in order to collect the following information about each project in our set: Age In days, Number of Stars, Number of Forks, Team size, Number of Pull Requests open, Number of Pull Requests merged, Number of Pull Requests rejected, Number of Core Pull Requests Open , Number of Core Pull Requests Merged, Number of Core Pull Requests Rejected, and Number of Issues open. The project properties with *Core* in their name refer to those managed by core developers and other project insiders, for example, *Number of Core Pull Requests Open* refers to the Number of PRs opened by project insiders. Vasilescu et al. [318] chose these data-points as representative characteristics of each project and its activity, and their works' validation by the research community indicate the validity of their variable selection.We especially note that the Age In days, Number of Stars, Number of Forks, Team size, are used as numerical estimators of the size of the projects in our work, similar to other works [318, 355, 36]. We collected these project properties to enrich the data-set and facilitate the statistical analyses within this work such as ANCOVA [172, 275].

2.3.2 DevOps Tools Classification

DevOps has many competing definitions, consequentially, there is no consensus on how to determine whether or not a project is employing DevOps. Prior research [193, 258, 97] on DevOps and DevOps tools also identified the same challenge. To circumvent this problem, we used the adoption of DevOps tools as an indicator of the adoption of DevOps, and we focused on analyzing these tools and their usage within our chosen project-set. DevOps tools are defined by Leite et al. [181] as the tools pursuing human collaboration across different departments, enabling continuous delivery, and maintaining software reliability. We opted for this definition as it is similar to those found within other research works concerning DevOps and DevOps tools [104, 46, 180]. Initially, we considered the list of DevOps tools determined by Leite et al. [181]. However, since this list was formed by analyzing traditional software, we wanted to expand the number of tools within our analysis to avoid missing any popular tools that are more popular with ML projects. To expand our list of DevOps tools to consider within this work, we followed the method outlined in Section 2.3.3.1, to discover new DevOps tools in-use within our projects but not described within previous works. We classified the different tools we found into 6 categories:

1. **Build Tools:** Responsible for generating packages meant for deployment, also referred to as builds. They are also generally responsible for generating other artifacts and providing feedback to developers using only the source code as input.

2. **Continuous Integration (CI) Tools:** Responsible for the orchestration of several

steps that ensure the development pipeline and automation of development tasks such as package generation, automated test execution, and deployment to both development and production environments.

3. **Deployment Automation Tools:** Make use of certain outputs of the continuous delivery process. They are employed in the deployment stages in order to allow frequent and reliable deployment processes.

4. **Monitoring and Logging Tools:** Responsible for tracking non-functional properties, such as performance, availability, scalability, resilience, and reliability.

5. **Test Tools:** Validate the functionality of software, and identify possible errors, or missing requirements.

6. **Code Analysis Tools:** Static code analyzers that perform several operations, such as code coverage, static error detection, etc.

The Code Analysis category was proposed by Yin & Filkov et al. [344], and Leite et al. [181] coined the first 4 categories and while they considered Test tools as a part of the Build category, we opted to consider them as a separate category due to the difference in their respective goals, as detailed within the definitions above. We didn't consider Source code management tools in our analysis because the projects in our dataset were all collected from GitHub. Furthermore, our analysis in Section 2.3.3.1 did not uncover any ML-specific tools. To further verify the absence of usage of these tools, we performed an automatic search for the configuration files of some ML-specific tools such as MLFlow [50], Amazon SageMaker [292] and Spack [102], and we found no evidence of their usage within the two categories of ML projects we considered

2.3.3 Methods of Analysis

The overview of our analysis is illustrated in Figure 2.1.

2.3.3.1 Phase 1: File, Name and Import pattern collection

DevOps configuration files are written in a variety of domain specific languages (DSL). For example, the Maven build specification is written in an XML format, while the Gradle build specification is written in a Groovy-based DSL language. On the other hand, Docker uses a DSL that can only be parsed and recognized by the Docker tool. As a result, static program analysis techniques developed for certain programming languages or DSLs might not be

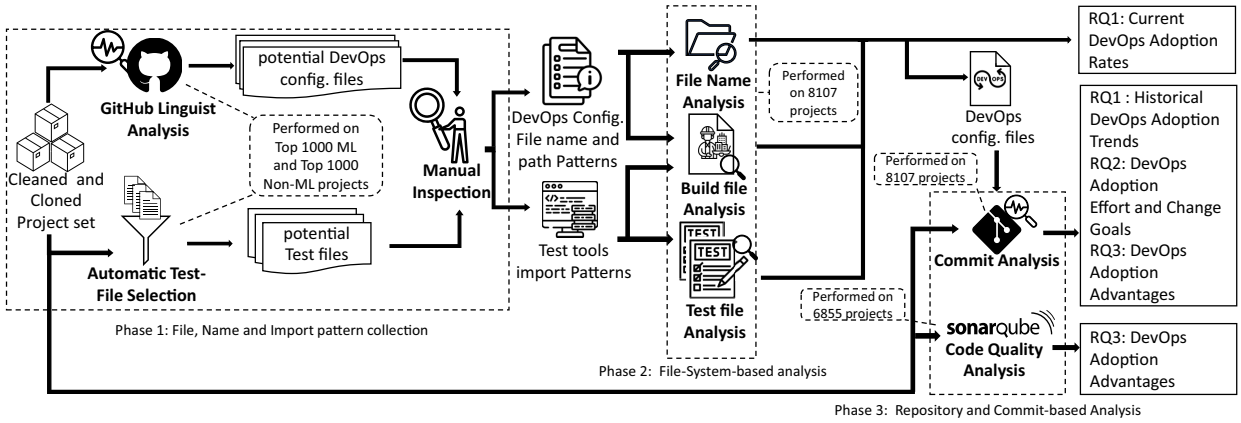


Figure 2.1: Overview of our Approach

sufficient to detect a large pool of DevOps tools. This lead us to use the configuration file name and path patterns to detect DevOps configuration files. We adapted this method from prior works which employed this approach for IaC and Build artifact files [207, 206, 159]. But first, in order to establish the set of DevOps tools to consider in this work, we considered the list of tools proposed by Leite et al. [181] as a starting point. However, upon realizing its limitations, as discussed within Section 2.3.2, we performed a semi-automatic classification of DevOps configuration files on the top 1000 ML projects and 1000 Non-ML projects based on their GitHub project popularity[1]. First, performed an automatic classification of the files within the repositories of the aforementioned projects using the GitHub Linguist tool [107]. Then, a co-author manually verified the resulting classification, and extracted from it the possible DevOps configuration files by ignoring files with known extensions or names, such as source code and readme files. Libraries.io [171] was then consulted to find the tools corresponding to these configuration files and verify if they corresponded to DevOps tools. Finally, these tools' documentation were examined to extract configuration file name and path patterns that correspond to them. These patterns are then used within the phase described in Section 2.3.3.2. However, no such patterns were found for testing tools as they do not rely on specific configuration files. To detect these tools, we identified the testing

[1]The project popularity criteria used was a combination of the number of stars and number of watchers

files within the aforementioned repositories, using the name and path pattern-based method proposed by Zhu et al. [359] Then, the import or import-equivalent (e.g., include, using, etc.) statements within these files were manually checked by 2 co-authors and cross-referenced with the Libraries.io [171] dataset to determine if the modules being imported were testing tools and frameworks. These patterns are used within the phase described in Section 2.3.3.2. Overall, we identified 93 DevOps tools via this phase. Figure 2.2 presents a subset of the tools we identified and processes we used to identify them during our analysis, with a full list available at the replication package.

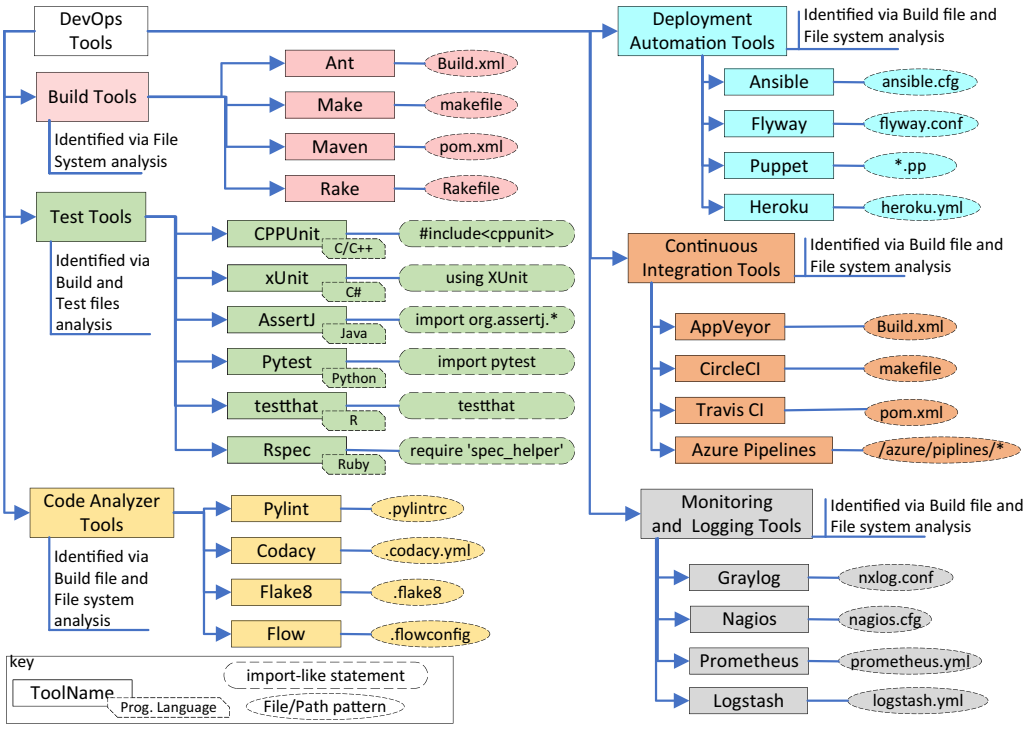


Figure 2.2: Subset of DevOps tools, their categories, and their corresponding configuration file name patterns or import statements used to detect their usage.

2.3.3.2 Phase 2: File System Analysis

Having extracted the file name and path patterns for Build, Continuous Integration, Deployment Automation, Code Analysis and Monitoring and Logging Tools, import-equivalent statements of the Test tools, we used these patterns to verify their adoption within a certain repository. We considered the existence of a configuration file matching the file name and path patterns of a specific DevOps tool as indicative of that tool's usage within the project.

For example, a *pom.xml* file in the project repository indicates that Maven is being used as a Build tool within that project and a *.travis.yml* indicated that the project adopted Travis CI for Continuous Integration. Using the GitPython [122], and PyGitHub [251] libraries, we created a tool that allowed us to access and clone the remote source codes of these projects into a local file system. Then, we analyzed the files of each project and attempted to match them with the aforementioned patterns to detect if the tools corresponding to these patterns were adopted within each project. For the specific case of testing tools, we analyzed the test code files, detected per the method specified by Zhu et al. [359], for the import statements specific to the test file's possible testing tools, which are language specific. For example, if a test file has the `.py` extension, it is identified as a Python file. It is then scanned for the import statements of Python testing tools identified within Section 2.3.3.1. For example, if the statement `import pytest` is found, the project that contains the test file is assumed to be using the `PyTest` tool. In a software system, a build script is responsible for collecting the necessary dependencies, thus analyzing build scripts can provide important information regarding their usage within a project. For example, Fan et al. [87] relied on build-script analysis to find dependency related errors related to building projects. In addition to the two previously described methods, we relied on the analysis of build scripts and considered a project's dependency on a tool to be indicative of its use within it. For example, if a project specified a dependency on Codecov within its Maven *pom.xml* file, we considered the project to be using the Codecov tool. We used this method to detect the usage of DevOps tools of all categories. The categories of DevOps tools and the methods we used to identify the tools of those categories, as well as a subset of the DevOps tools we considered, and their corresponding file name and path patterns or import patterns are illustrated in Figure 2.2. To determine the different variables that contribute to DevOps adoption within different project categories, We performed an ANCOVA [275] analysis, a type of GLM regression for models with categorical and continuous variables, using DevOps adoption as a dependent variable and the additional data we collected, detailed in Section 2.3.1, as covariates. This phase allowed us to answer part of **RQ1** regarding the current adoption of DevOps of the different project categories, the project's properties linked to its DevOps adoption, and the most popular DevOps tools of each type in the different project categories we specified. We also used this phase to extract the different DevOps configuration files used within the different phases described in Section 2.3.3.3.

2.3.3.3 Phase 3: Repository and Commit-based Analysis

Repositories and their commits contain valuable information about a project's development and maintenance efforts [268]. DevOps tools are meant to be configured and updated via

their configuration files, hence, commits affecting these files contain insight into the usage trends and practices of DevOps tools. We extracted the DevOps configuration files via the steps discussed in Section 2.3.3.2. We performed our analysis on the Main branch of the different repositories, using the PyDriller [300] tool and Github GraphQL API [113] to obtain additional data not stored in the Git repository, such as the CI status following a commit. While Test files were analyzed within Section 2.3.3.2 to extract information about a project's testing framework, which we considered a type of DevOps tool, test files are not considered DevOps configurations files within the scope of this analysis. This is because Test code is very similar to source code and test file changes are highly coupled with source-file changes [191, 348]. In contrast, DevOps configuration files are used to configure the different DevOps tools used within a project, such as Continuous Integration tools.We used this commit-based analysis to answer **RQ1** regarding DevOps historical adoption trends via analyzing our projects' commits, where we assumed the date of the first commit within a project to be the date of its creation, and the date of the addition of the first DevOps configuration file within a project to be an indicator of when it adopted DevOps. We also answered **RQ2** using commit-based analysis via the sub-phases detailed in Section 2.3.3.3, Section 2.3.3.3, and **RQ3** using commit-based analysis and repository-based analysis via the sub-phase Section 2.3.3.3.

Phase 3-a: DevOps Adoption Effort To obtain a better idea about the configuration and maintenance efforts of DevOps tools, we analyzed the commits that modified one or more DevOps configuration files. We calculated the *Commit Ratio* metric, which is similar to the amount of commits metric, used by a number of works to estimate activity within a project [337], but adapted to the context of a specific type of files, to estimate the portion of commits that affect DevOps configuration files. This metric is defined as follows:

Commit Ratio:

$$CommitRatio = \frac{NofC_{DevOps}}{NofC}$$

$NofC_{DevOps}$ is the total number of commits that involved DevOps configuration file(s) and $NofC$ is the total number of commits.

To estimate the size of an update per-file-type within a project, We calculated the *Average Normalized Code Churn* of Source and DevOps configuration files, a commonly used metric [337] that was also previously used in the context of build artifacts [207], and that is superior to other metrics such as Lines of Code (LOC) [201]. This metric is defined as follows:

Average Normalized Code Churn:

$$AvgNormal.CodeCh.(Type, Project) = \frac{\sum_{i=1}^{n} \frac{NBFilesChanged(Type, Project)}{NBFilesExist(Type, Project)}}{NbOfDevMonths}$$

$NbOfDevMonths$ is the number of development months[2]. $NBFilesChanged(Type, Project)$ is the number of either Source code or DevOps configuration files of that changed during a development period, $NBFilesExisted(Type, Project)$ is the number of files of a certain type, source code file or DevOps configuration file, that existed during a development period.

For each project category, we performed 2 ANCOVA [275] analyses, using Commit Ratio as a dependent variable for the first analysis and Normalized Code Churn for the second analysis, and using the covariates presented within Section 2.3.1. In total, this was 6 ANCOVA analyses. We also performed 2 ANOVA analyses to detect any statistical differences concerning these metrics between the different project categories. We used this sub-phase and its associated analyses and metrics to partially answer **RQ2** regarding DevOps adoption efforts.

Phase 3-b: DevOps Change Goals While the Normalized Code Churn and Commit Ratio metrics inform us on the properties of DevOps configuration files changes, they do not reveal the underlying causes of the changes occurring to these DevOps configuration files. To approximate the change goals of DevOps configuration files, we selected the projects that adopted at least a Build and a CI tool, then analyzed their commits that affected their DevOps configuration file(s). We analyzed commits from 851 ML Applied projects, 586 ML Tool projects, and 1942 Non-ML projects. We classified the commits' main change goal between 4 different alternates:

- **Bug Fix:** A bug fix is done to remedy a programming bug or error. However, identifying bug-fixing commits in a Git commit-history is a challenging task [29]. To identify this type of commit, we adopted the approach proposed by Ray et al. by scanning commit messages for the keywords ("error","bug","fix","issue","mistake","incorrect","fault","defect","flaw","type") [262].
- **CI Build Fix:** CI Build fix refers to code changes that aim to fix integration failures such as compilation failures, dependency issues, unit test failures, etc. that are reported by CI systems, and also referred to as Build Breakages. To detect these commits, we

[2]We considered a development month to be 30 days within this work

adopted an approach proposed by Hassan & Wang et al. [142], and that's similar to the approach used by Hyunmin et al. [290] to detect a build-failure resolution. Based on this approach, if a commit changes the CI build status from *Build failure* or *Build error* to *Build success*, we consider the commit a CI-fixing commit. We used the GraphQL Github API [113] to detect the CI build status.

- **Bug and CI Fix:** A commit that meets the criteria of a Bug Fix commits and CI fix commit is considered to be attempting to fix both types of problems.
- **Other changes:** We considered commits that contain neither a bug fix nor a CI fix as commits with the main goal of other miscellaneous changes. These commits may add new functionality, refactor existing code, etc.

Finally, in order to make these measures project-specific, we calculated the percentage of each of the aforementioned commit types out of all the commits of a project.

For each project category, we performed an ANCOVA [275] analysis, using the four goals, Bug and CI fix, Bug fix, CI fix, and Other changes, as dependent variables for the analysis, and using the same covariates as the ANCOVA analysis done within Section 2.3.3.1 in addition to the adoption of different DevOps Tool types, such as Build Tool Adoption, CI Tool adoption, etc. In total, this was 3 ANCOVA analyses. We used this sub-phase and its associated analyses to partially answer **RQ2** regarding DevOps change goals

Phase 3-c: DevOps Adoption Advantages Having gained an idea about the properties and goals of the changes performed on DevOps configuration files, we wanted to develop an understanding of the advantages associated with adopting DevOps Tools. To achieve this, we used the metrics of Commit Frequency, Merge Frequency, and Average Issue duration, which also rely on commit-based analysis , and Code Quality, through the widely-used tool SonarQube [299] which relies on repository-based analysis.

DevOps encourages more code sharing via frequent commits and merges, hence Average Commit Frequency and Average Merging Commits Frequency are correlated diretctly to is principles of DevOps. These two metrics are calculated as follows: **Average Commit Frequency**:

$$AverageCommitFrequency = \frac{NBofCommits}{NBofDevMonths}$$

$NBofCommits$ is the total number of commits within a project and $NBofDevMonths$ is the total number of development months within a project.

Average Merging Commits Frequency:

$$AverageMergingCommitFrequency = \frac{NBofMergingCommits}{NBofDevMonths}$$

$NBofMergingCommits$ is the total number of merging commits within a project and $NBofDevMonths$ is the total number of development months within a project.

A reduced issue duration is also an expected result of adopting DevOps, since it is claimed to increase the speed and productivity of teams in relation to resolving software issues, making Average Issue Duration a good metric to evaluate this claim. This metric is calculated as :

Average Issue Duration :

$$AvgIssueDuration(Project_A) = \frac{\sum_{i=1}^{n} Duration(Issue_i, Project_A)}{TotalNBIssues(Project_A)}$$

$Duration(Issue_i, Project_A)$ is the duration of an issue i for a project A,
n $TotalNBIssues(Project_A)$ indicates the number of issues for that project.

Finally, DevOps is associated with an improvement in the quality of the development process, and possibly that of the code-base as well. We used the Maintainability and Reliability code quality metrics as generated by SonarQube to evaluate the quality of the projects within our set.

Prior works [259, 149, 320] used similar metrics and tools to analyze the effectiveness of adopting CI within a number of projects, giving confidence to their effectiveness.

For each project category, we performed 4 ANCOVA [275] analyses, using Average Commit Frequency as a dependent variable for the first analysis, Average Merging Commits Frequency for the second analysis, Average Issue Duration for the third analysis, and Code Quality for the fourth analysis. We used the same covariates as the ANCOVA analysis done within Section 2.3.3.1. In total, this was 12 ANCOVA analyses. We also performed 4 ANOVA analyses to detect any statistical differences concerning these metrics between the different project categories. We used this sub-phase and its associated analyses to partially answer **RQ3** regarding DevOps adoption advantages.

2.4 Results

2.4.1 Adoption rates of DevOps Tools

RQ1

What are the current and historical adoption rates of DevOps Tools for ML and Non-ML projects?

2.4.1.1 DevOps' current Adoption Rates

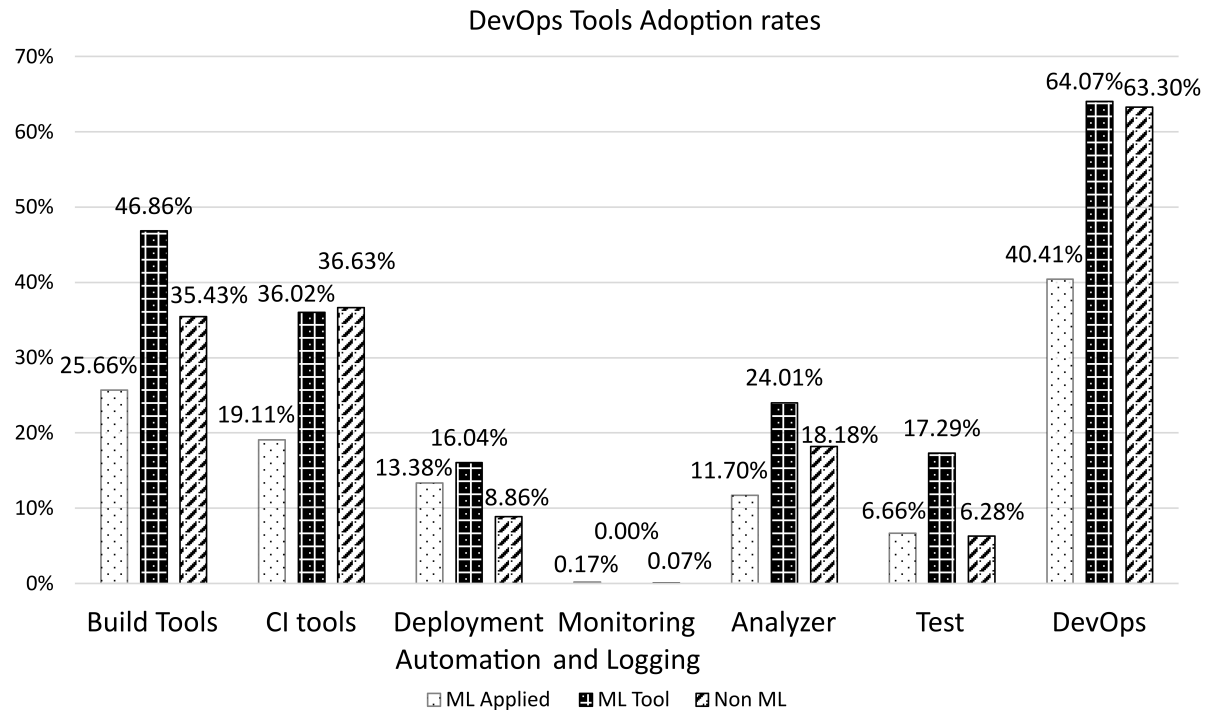


Figure 2.3: DevOps Tools Current Adoption Rates

Adopting DevOps tools and practices within software projects has numerous advantages to the productivity of a development team and the quality of their processes. Since their growth in popularity, DevOps tools are progressively being embraced by independent developers and companies alike. Following our analysis, we were able to confirm this with the high adoption rates of 63.30% for Non-ML projects, and 64.07% for the ML Tool projects. However, ML Applied projects have shown a lower adoption rate of only 40.41%. Focusing on the different DevOps tools categories, ML Tool projects generally had the highest adoption rates across the majority of tool types, with Non-ML projects following as a close second, and Applied projects trailing as the third.

To identify the factors behind adoption of DevOps, we performed an ANCOVA [275] analysis, a type of GLM regression for models with categorical and continuous variables, for each project category. We used DevOps adoption as a dependent variable and the additional data we collected concerning each project, detailed in Section 2.3.1, as covariates. The project-specific data points were: Age In days, Number of Stars, Number of Forks, Team size, Number of Pull Requests open, Number of Pull Requests merged, Number of Pull Requests rejected, Number of Core Pull Requests Open, Number of Core Pull Requests Merged, Number of Core, Pull Requests Rejected, and Number of Issues open.

Source	Sig.	Partial Eta Squared	Details
Intercept	<.001	.007	Intercept of the model
Age In Days	<.001	.027	A project's Age
Team Size	<.001	.008	A project's team-size
N_Pr_Merged	<.001	.008	Number of Pull requests merged
N_Pr_Core_Merged	<.001	.006	Number of Pull requests by core developers merged

R Squared = .119 (Adjusted R Squared = .116)

Table 2.1: ANCOVA analysis of DevOps adoption within Applied projects (Only statistically significant† variables are shown)

† Statistically significant variables have a Sig.(P-value) less than 0.05

Using the results of the ANCOVA analysis illustrated within Table 2.1, we found that for ML Applied projects, the most important statistically-significant factors that contribute to DevOps adoption within them were the Age of a project and its Team size. This is indicated via the Partial Eta Square statistic which informs us which variables have the largest effect on the dependent variable, which is a project's adoption of DevOps in our case. Hence, older and larger ML Applied projects are more likely to adopt DevOps. Similar results were found when performing ANCOVA on ML Applied projects while considering as dependent variables each DevOps tool category, except Analyzer and Test tools where only Team size was a determining variable of their adoption of DevOps. No statistically significant contributor was determined behind the adoption of monitoring and logging tools by ML Applied projects, most likely due to its low adoption by this project category.

For ML Tool projects, as illustrated within Table 2.2, Age was statistically significant correlated to their DevOps adoption. Furthermore, the Number of stars and Number of forks also had a significant correlation to their DevOps adoption. It's important to note that

Source	Sig.	Partial Eta Squared	Details
Intercept	<.001	.131	Intercept of the model
Age In Days	<.001	.023	Age of the project
N_Stars	.036	.004	Number of Stars of project
N_Forks	.016	.006	Number of Forks of project
N_Pr_Open	.020	.005	Number of pull requests open of project

R Squared = .096 (Adjusted R Squared = .087)

Table 2.2: ANCOVA analysis of DevOps adoption within Tool projects (Only statistically significant variables are shown)

while Team size was significantly correlated to DevOps adoption of ML Applied projects, this correlation was not found within Tool projects. Around 50% of ML Tool projects before our re-categorization process were backed by major organization such as Microsoft and IBM [128], and after this process, and we estimate that 30% of these projects are backed by such organizations. This makes all the more surprising the lack of correlation between team-size and DevOps adoption for ML Tool projects, especially considering these organization are more likely to have larger resource and to adopt best practices such as DevOps in comparison to independent developers. It's also important to note that ML Tool projects show more variance within their team sizes than their ML Applied counterparts, as illustrated within Figure 2.4, signaling that a lack of correlation between Team size and DevOps adoption is not due to limitations related to sample size, but rather the properties of ML Tool projects. Focusing on the different categories of DevOps tools, Age was also a key variable in determining whether an ML Tool project adopts Build, CI or Deployment tools, while surprisingly, Team size was the key predictor of Code Analysis tools adoption. Finally, no predictors of Monitoring tools' adoption by ML Tools projects was found.

Considering Non-ML projects, it's clear through Table 2.3 that they show similar results regarding the factors contributing to DevOps adoption to those of ML Tool projects. A Non-ML project's age, pull-request based development activity, popularity as measured by its number of forks, and its team size are significant contributing factors to its adoption of DevOps. Focusing on the different categories of DevOps tools, two or more of the aforementioned projects' characteristics were among the main predictors of the adoption of a specific DevOps tools category, indicating no major difference between the predictors of DevOps adoption in-general and the adoption of a specific category of DevOps tools by Non-ML projects.

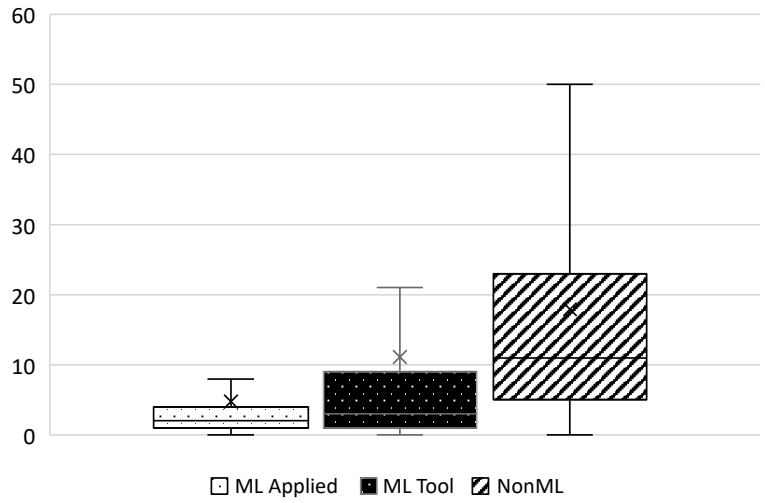


Figure 2.4: Variance of Team size (Outliers removed)

Source	Sig.	Partial Eta Squared	Details
Intercept	<.001	.106	Intercept of the model
Team Size	.003	.002	Size of the project's team
Age In Days	.006	.002	Age of the project
N_Forks	.010	.002	Number of Forks of projects
N_Pr_Open	.951	.000	Number of Pull Requests open of project

R Squared = .060 (Adjusted R Squared = .057)

Table 2.3: ANCOVA analysis of DevOps adoption within Non-ML projects (Only statistically significant variables are shown)

Category	Most important variables affecting DevOps adoption	Interpretation
ML Applied	Age In Days, Team Size, N_Pr _Merged, N_Pr_Core _Merged	An ML Applied projects' DevOps adoption is linked to its age, team size and reliance on PR-based development as measured through its number of pull requests merged
ML Tool	Age In Days, N_Stars, N_Forks, N_Pr_Open	An ML Tool projects' DevOps adoption is linked to its age, popularity as measured with its number of stars and forks, and its Number of PRs open
Non-ML	Team Size, Age In Days, N_Forks, N_Pr _Open	A Non-ML projects' DevOps adoption is linked to its team size, age, popularity as measured with its number of forks and reliance on PR-based development as measured through its number of pull requests open

Table 2.4: Summary of ANCOVA analyses results for DevOps Adoption

A summary of our findings is illustrated in Table 2.4. We found that an ML Applied project's age and team size are more likely to affect its' DevOps adoption more so than that of a Tool or Non-ML project. Similar characteristics related to a project's size, popularity, as measured by its number of stars and forks, and its reliance on PR-based development, are the important factors that affect whether or not it adopts DevOps, regardless of whether or not it's an ML project. One important outlier is that an ML Tool projects' team size does not affect its DevOps adoption outcome. Based on observations by Karlaš et al. [169], Renggli et al. [264], Lwakatare et al. [197, 196], Amershi et al. [13], and Arpteg et al. [23], we attribute the lower adoption of DevOps by ML Applied projects to the differences between traditional software projects and ML projects, and a lack of DevOps tools that were specifically designed for small scale ML projects.

2.4.1.2 Most popular DevOps tools

In addition to exploring the adoption rates of DevOps tools and the factors affecting their adoption, we were interested in exploring which tools were currently popular across the different types of projects. Tables 2.5 to 2.9 illustrate the adoption rates for the DevOps tools that have at least 1% adoption rate by one or more categories of projects. We believe

knowing which tools are popular for each project category can help guide future research regarding DevOps practices within them. For example, research on Code analysis within ML projects should focus on the Coverage and Pylint tools since they are the most popular Code analysis tools within them.

Tool Name	Project Category	Adoption Percentage
setuptools	ML Applied	9.88%
	ML Tool	18.91%
	Non-ML	0.74%
Rake	ML Applied	0.41%
	ML Tool	1.34%
	Non-ML	1.72%
QMake	ML Applied	1.30%
	ML Tool	1.25%
	Non-ML	2.21%
Maven	ML Applied	2.78%
	ML Tool	5.38%
	Non-ML	1.67%
MakeFile	ML Applied	16.78%
	ML Tool	30.73%
	Non-ML	3.68%
JUnit	ML Applied	2.81%
	ML Tool	3.76%
	Non-ML	1.64%
Gradle	ML Applied	2.02%
	ML Tool	2.06%
	Non-ML	2.77%
Clang	ML Applied	2.06%
	ML Tool	6.63%
	Non-ML	0.65%
Ant	ML Applied	1.72%
	ML Tool	5.02%
	Non-ML	0.54%

Table 2.5: Usage rates of Build Tools (Tools with 1% or more usage rates)

2.4.1.3 DevOps' historical Adoption Rates

We analyzed the historical adoption trends of DevOps tools to get a better understanding of the evolution of their adoption rates over time. The results of our analysis are illustrated in Figure 2.5. When analyzing the growth of Non-ML projects overall in comparison to that of Non-ML projects with one or more DevOps tools, it's clear that they both have similar

Tool Name	Project Category	Adoption Percentage
Pylint	ML Applied	2.02%
	ML Tool	4.75%
	Non-ML	0.17%
Flow	ML Applied	0.96%
	ML Tool	1.88%
	Non-ML	0.39%
Flake8	ML Applied	1.78%
	ML Tool	3.32%
	Non-ML	0.05
ESLint	ML Applied	1.58%
	ML Tool	1.34%
	Non-ML	2.55%
Coverage	ML Applied	3.50%
	ML Tool	7.89%
	Non-ML	0.29%
Codecov	ML Applied	2.64%
	ML Tool	5.38%
	Non-ML	0.25%
CodeClimate	ML Applied	0.48%
	ML Tool	1.08%
	Non-ML	0.27%
Clang	ML Applied	1.58%
	ML Tool	6.00%
	Non-ML	0.37

Table 2.6: Usage rates of Code Analysis Tools (Tools with 1% or more usage rates)

Tool Name	Project Category	Adoption Percentage
testthat	ML Applied	1.03%
	ML Tool	2.78%
	Non-ML	0.12%
Pytest	ML Applied	2.81%
	ML Tool	6.45%
	Non-ML	0.39%
JUnit	ML Applied	1.34%
	ML Tool	2.42%
	Non-ML	2.04%
Cassert	ML Applied	0.34%
	ML Tool	1.08%
	Non-ML	0.37%

Table 2.7: Usage rates of Test Tools (Tools with 1% or more usage rates)

Tool Name	Project Category	Adoption Percentage
Travis	ML Applied	17.94%
	ML Tool	33.24%
	Non-ML	10.30%
Jenkins	ML Applied	0.58%
	ML Tool	1.97%
	Non-ML	0.05%
AppVeyor	ML Applied	2.44%
	ML Tool	6.09%
	Non-ML	0.76%

Table 2.8: Usage rates of Continuous Integration Tools (Tools with 1% or more usage rates)

Tool Name	Project Category	Adoption Percentage
Docker	ML Applied	13.17%
	ML Tool	15.59%
	Non-ML	1.67%
Chef	ML Applied	0.10%
	ML Tool	0.36%
	Non-ML	0.64%

Table 2.9: Usage rates of Deployment Automation Tools (Tools with 1% or more usage rates)

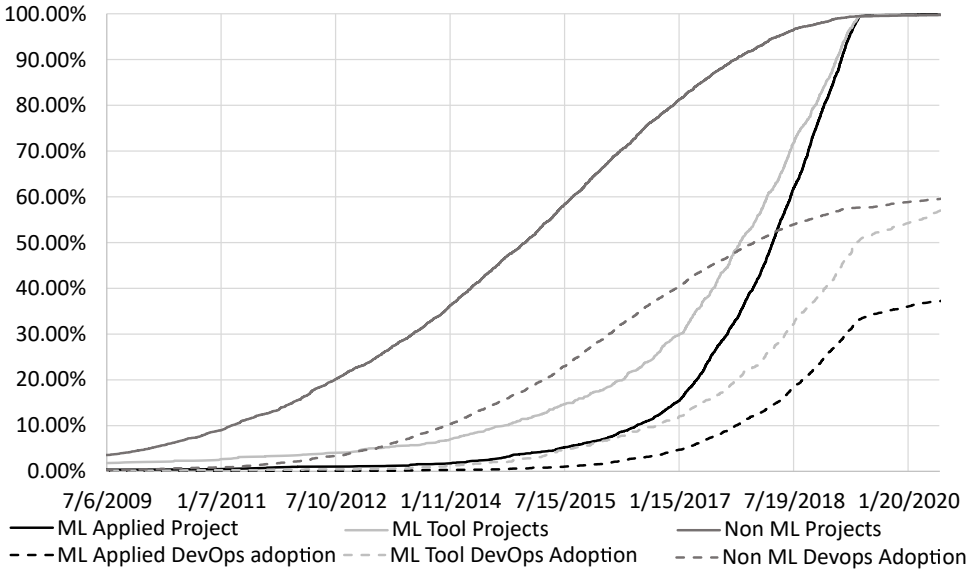


Figure 2.5: Historical project amounts and their DevOps tools' adoption (normalized to percentages)

trends over time, signaling a healthy adoption growth of DevOps among this type of projects.

Focusing on ML project types, both ML Tool projects' growth and ML Applied projects' seen a near exponential increase starting from 2017.The explosion in the projects' total amount can be attributed to the advances in ML fields and gains in their popularity. Focusing on the amount of ML Tool projects with DevOps tools, it shows similar growth trends as the total number of the ML Tool projects. This similarity in growth trends is also observed for Non-ML projects growth. However, while ML Applied projects have seen a similar in amount to ML Tool projects due to analogous reasons, their DevOps adoption growth has stalled in comparison. DevOps tools' in ML Applied projects had a slower and lower adoption rate overall in comparison to both Non-ML and ML Tool projects, and we were able to partially link them to the smaller team sizes of these projects in Section 2.4.1.1 to their lower DevOps adoption. Overall, these results indicate that the current adoption rates are consistent with the historical rates across project categories, and there are no abrupt changes of DevOps adoption.

Finding 1:

ML Tool projects and Non-ML projects have significantly higher current and historical DevOps tools' adoption rates than ML Applied projects. This adoption is most influenced by a project's age, team-size or both factors, depending on the project's category.

2.4.2 DevOps Maintenance Efforts and Goals

RQ2

What are the maintenance efforts and goals associated with DevOps tools across the different categories of projects ?

Having determined the historical and current adoption rates, we wanted to investigate the differences in the effort that developers are putting into maintaining their DevOps configuration files and the correct functioning of DevOps tools within their repositories, and to explore the different goals of updates to DevOps configuration files.

2.4.2.1 Ratio of DevOps configuration files' updates

We used the Commit Ratio metric to estimate the share of updates that affect DevOps tools out of all the updates that affect a repository. As illustrated by Figure 2.6, Tool projects tend to update their DevOps configuration files less overall, while Applied and Non-ML

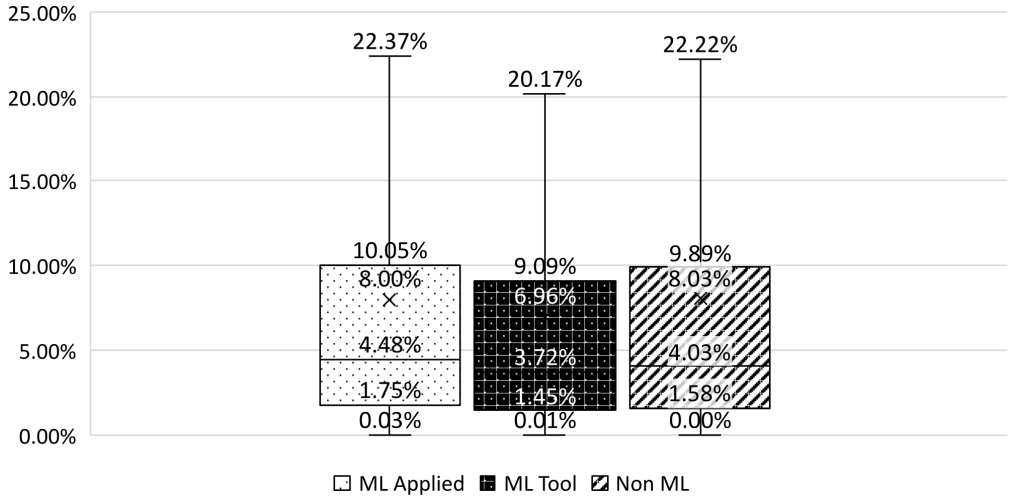


Figure 2.6: Commit Ratios of DevOps configuration files

projects had higher and similar ratios of updates. The projects with the highest DevOps commits ratio are generally those with the majority of their updates affecting their Build, CI or Deployment automation files. One such example is the ML Applied project ROSETTE-API/ROSETTE-ELASTICSEARCH-PLUGIN, with 78.46% of its commits modifying its Maven and Travis file. The majority of these updates are comprised of version or dependency and configuration changes for the project overall or its docker image and the plug-ins it provides. Another example is the CLARITYCAFE/IVY repo, which has frequent commits which almost always change its Travis CI and Docker file. Upon closer inspection, we identified that this project's Docker and Travis files are mostly changed to fix CI and Deployment problems. These examples and our statistical findings stand in contrast with the concept of "write-once-and-forget-it" for DevOps configuration files and indicate that they evolve frequently for different aspects of software maintenance.

To further investigate whether these project-specific trends are a widespread phenomenon, we performed, the ANCOVA analysis illustrated in Table 2.10, we found that CI adoption, and the adoption of CI, Build and Test tools at the same time to be among the strongest factors leading to a higher commit ratio in Applied projects.

However, after performing the same analysis on the other two project categories, we found no statistically significant link between the adoption of specific DevOps tool categories and the commit ratio in ML Tool and Non-ML projects. This allows us to deduce that specific categories of DevOps tools, such as CI, Build and Testing tools in ML Applied projects need more frequent updates in comparison to other types of tools. Yet, ML Tool and Non-ML projects do not show this correlation. A summary of our ANCOVA analyses is found within Table 2.11

Source	Sig.	Partial Eta Squared	Details
Intercept	<.001	.022	Intercept of the model
CI	.006	.007	Adoption of CI tool(s)
Build * CI * Analyzer	.007	.007	Adoption of Build, CI and CA tool(s)
Build * Deployment * Test	.024	.005	Adoption of Build, DA and Test tool(s)
Build * Analyzer * Test	.035	.004	Adoption of Build, Code Analysis and Test tool(s)
CI * Deployment * Analyzer	.045	.004	Adoption of CI, DA and CA tool(s)
CI * Deployment * Analyzer * Test	.045	.004	Adoption of CI, DA , CA and Test tool(s)

R Squared = .098 (Adjusted R Squared = .063)

Table 2.10: ANCOVA analysis of Commit Ratio for Applied projects (Only statistically significant variables are shown)

Category	Most important variables affecting DevOps churn	Interpretation
ML Applied	CI, Build * CI * Analyzer, Build * Deployment * Test, Build * Analyzer * Test, CI * Deployment * Analyzer, CI * Deployment * Analyzer * Test	An ML Applied projects' adoption of certain DevOps tool categories or a combination of these categories is linked to an increase in its DevOps configuration files commit-ratio
ML Tool	None	An ML Tool projects' DevOps configuration files commit-ratio is not linked to its adoption of a tool of a certain DevOps category.
Non-ML	None	A Non-ML projects' DevOps configuration files commit-ratio is not linked to its adoption of a tool of a certain DevOps category.

Table 2.11: Summary of ANCOVA analyses results for DevOps Commit-ratio

Finally, we were able verify the statistical dissimilarity between the different projects categories via the one-way ANOVA test [174], a test developed to allow the comparison of the means of three or more different groups based on one property. The p-value obtained was 0.032 implying significant statistical difference between the three groups regarding their

Commit Ratios.

2.4.2.2 DevOps Coding Efforts

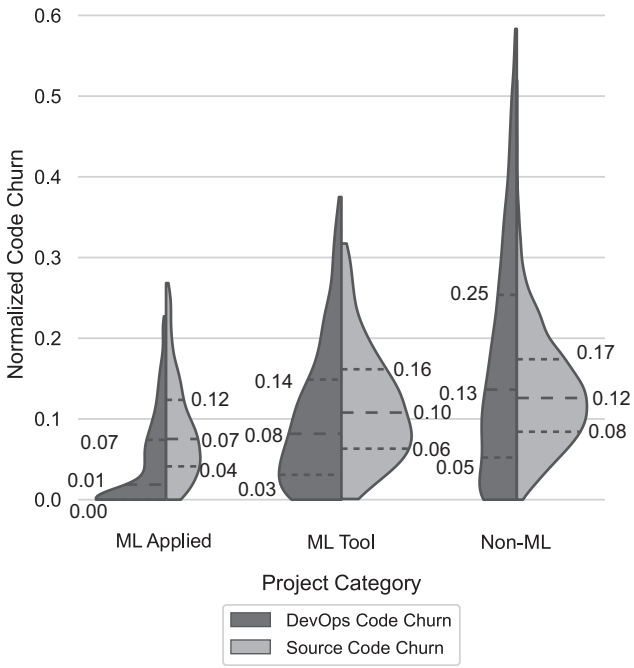


Figure 2.7: Average Normalized Code Churn(Outliers removed with IQR [273])

To estimate the effort that developers put into DevOps configuration files in comparison to Source files between different commits, we used the Average Normalized Code Churn metric. As illustrated by the results in Figure 2.7, a comparatively higher relative churn of DevOps configuration files is noted in ML Tool projects in comparison to ML Applied projects. This is made clearer with the higher quartiles and median values of this metric for ML Tool DevOps churn in comparison to those of ML Applied projects. Non-ML projects had a bigger churn overall on both file-types, yet its DevOps churn shows a more even distribution across its value range, reflecting more diverse DevOps maintenance practices within these projects. With a more detailed analysis, we identified that both Source and DevOps churn values are generally high at the beginning of a project's history, matching the intuition regarding changes being done to a large number of files as the project's initial code and configuration are being defined across a variety of them. These rates tended to quickly drop in value during the following months. Regarding DevOps Churn specifically, it tended to increase across all project categories whenever a new DevOps tool was added to a project, and it can take several development periods to drop again. This signifies a possible adoption barrier due to the time and effort required to establish and configure correctly

working DevOps tools in a project.

Focusing on some interesting cases, the ML Applied project with the highest Avg. Normalized DevOps Churn and Source code churn was the INDIX/WHATTHELANG project with the respective values of 1.0 and 0.43. This project provides a language prediction application usable via a CLI or an API. It employs Travis CI For continuous integration. Within this repository, 23 total commits over the period of one month were made. The only DevOps file within this project was a .travis.yml file, and it was updated more than once during that month, but not all of the source files were updated during this period following their creation.

The ML Tool project with the highest Avg. Normalized DevOps Churn and Source code churn was the YINCHUANDONG/SENTIMENT-ANALYSIS project with values of 1.0 and 0.2 respectively. It is a Deep Learning Workflow for Sentiment Analysis, and the only DevOps tool it uses is Docker for Deployment Automation. It also has a relatively low activity with 36 commits over the duration of one month, during which the Docker file was frequently updated. These two specific cases aside, ML projects of both types had DevOps churn values close to their Source churns. This implies that DevOps configuration files require development effort similar to that of Source files, along with the accompanying time and resource investments. Our intuition is confirmed within the ANCOVA analyses of DevOps code churn across the different project categories, which are illustrated and discussed in the following paragraphs.

Focusing on ML Applied projects, the results of which are illustrated in Table 2.12, we found that their adoption of a DevOps tool, or a combination of tools, such as Build or CI tools, is strongly correlated with an increase in their DevOps churn. Furthermore, the varying effect size values (represented by the Partial Eta Square) imply that different DevOps tools have different effort-requirements, with CI Tools being the ones that are most effort-intensive for ML Applied projects.

Moving on to ML Tool projects, the ANCOVA of which is illustrated in Table 2.13, we also found that their adoption of one or more DevOps tools is correlated with an increase in their DevOps churn. In their case, the adoption of Build, Deployment Automation, Continuous Integration, and Code Analysis tools at the same time had the largest effect size, and thus the highest consequential increase in DevOps Churn. This implies that an ML Tool project's adoption of multiple DevOps tools categories at the same time is more likely to result in an increase of its DevOps configuration files churn and this increase is likely to be more substantial than that resultant of the adoption of DevOps tools of one category.

Finally, focusing on Non-ML projects' ANCOVA, illustrated in Table 2.14, we find similar

Source	Sig.	Partial Eta Squared	Details
Intercept	<.001	.022	Intercept of the model
Team Size	<.001	.021	Project's team size
Age In Days	<.001	.011	Project's age
N_Pr_Merged	.021	.005	Number of Pull requests merged
CI	.031	.004	Adoption of CI tool(s)
Deployment * Analyzer * Test	.032	.004	Adoption of DA, CA and Test tool(s)
Build * Deployment * Analyzer * Test	.037	.004	Adoption of Build, DA, CA and Test tool(s)
Analyzer * Test	.041	.004	Adoption of CA and Test tool(s)
N_Pr_Core_Merged	.048	.004	Number of Pull requests by core developers merged

R Squared = .213 (Adjusted R Squared = .182)

Table 2.12: ANCOVA analysis of DevOps Code Churn for Applied projects (Only statistically significant variables are shown)

results to those of ML Applied and ML Tool projects, establishing that the phenomena of increased DevOps configuration files Churn is true across project categories. It's interesting to note that the adoption of a different mix of DevOps categories, more specifically Build, Code Analysis and Test tools, which is different from that of ML Tool projects', is the variable with the largest effect size and hence the biggest effect on DevOps Churn of Non-ML projects. It is especially interesting that Test tools are within this group of category, as they do not rely on any specific configuration file. As mentioned in Section 2.3.3.3, we do not consider Test files as DevOps configuration files.,

The summary of our ANCOVA analyses in relation to DevOps churn is within Table 2.15. Notably, across all project categories, the number of issues does not seem to affect DevOps code churn, signaling a lack of correlation between the reporting of issues within a project and the churn of DevOps configuration files. Applying the one-way ANOVA test across the different categories, we obtain a p-value of 3.61e−18 for the Source Code Churn and 6.49e−18 for the DevOps Code Churn, implying significant statistical difference between the three groups of projects.

Source	Sig.	Partial Eta Squared	Details
Age In Day	<.001	.019	Age of the project
Intercept	<.001	.018	Intercept of the model
Build * CI * Deployment * Analyzer	<.001	.005	Adoption of Build, DA, CI, and CA tools
CI	.002	.004	Adoption of CI Tools
Build * CI * Analyzer	.002	.004	Adoption of Build, CI, and CA Tools
Deployment * Test	.006	.003	Adoption of DA and Test tools
N_issues_Open	.007	.003	Number of Issues open
CI * Analyzer	.020	.002	Adoption of CI and CA tools
Build * CI * Deployment	.021	.002	Adoption of Build, CI and DA tools
Build * Test	.042	.002	Adoption of Build and Test tools

R Squared = .106 (Adjusted R Squared = .090)

Table 2.13: ANCOVA analysis of DevOps Code Churn for Tool projects (Only statistically significant variables are shown)

Source	Sig.	Partial Eta Squared	Details
Intercept	<.001	.082	Intercept of the model
Build * Analyzer * Test	.009	.011	Adoption of Build, CA and Test tools
Age In Days	.010	.011	Age of the project
Build	.031	.008	Adoption of Build Tools
N_Pr_Open	.040	.007	Number of Pull requests opened
Team Size	.043	.007	Size of the project's team
CI	.049	.006	Adoption of CI Tools

R Squared = .148 (Adjusted R Squared = .091)

Table 2.14: ANCOVA analysis of DevOps Code Churn for Non-ML projects (Only statistically significant variables are shown)

Category	Most important variables affecting DevOps churn	Interpretation
ML Applied	Team Size, Age In Days, N_Pr_Merged, CI, Deployment * Analyzer * Test, Build * Deployment * Analyzer * Test, Analyzer * Test, N_Pr_Core_Merged,	An ML Applied projects' Team Size, Age, reliance on PR-based development, and its adoption of certain DevOps tool categories or a combination of these categories are linked to an increase in its DevOps configuration files churn
ML Tool	Build * CI * Deployment * Analyzer, Build * CI * Analyzer , Deployment * Test, N_Issues_Open, CI * Analyzer, Build * CI * Deployment, Build * Test	An ML Tool projects' DevOps configuration files churn is not linked to its adoption of certain DevOps tool categories, and its number of issues open.
Non-ML	Build * Analyzer * Test, Age In Days, Build, N_Pr_Open, Team Size, CI	A Non-ML projects' DevOps configuration files churn is linked to its adoption of certain DevOps tool categories, its age, its reliance on PR-based development, and its team size.

Table 2.15: Summary of ANCOVA analyses results for DevOps Churn

2.4.2.3 DevOps Change goals

After uncovering the efforts invested by developers in DevOps configuration files, we wanted to explore the goals developers were trying to achieve by changing one or multiple DevOps configuration files. To achieve this, we analyzed the different commits that affect DevOps configuration files and determined the commits' main goals, within 1437 ML projects and 1942 Non-ML projects which adopted Build and CI Tools, via a process detailed in Section 2.3.3.3. The results are illustrated in Figure 2.8.

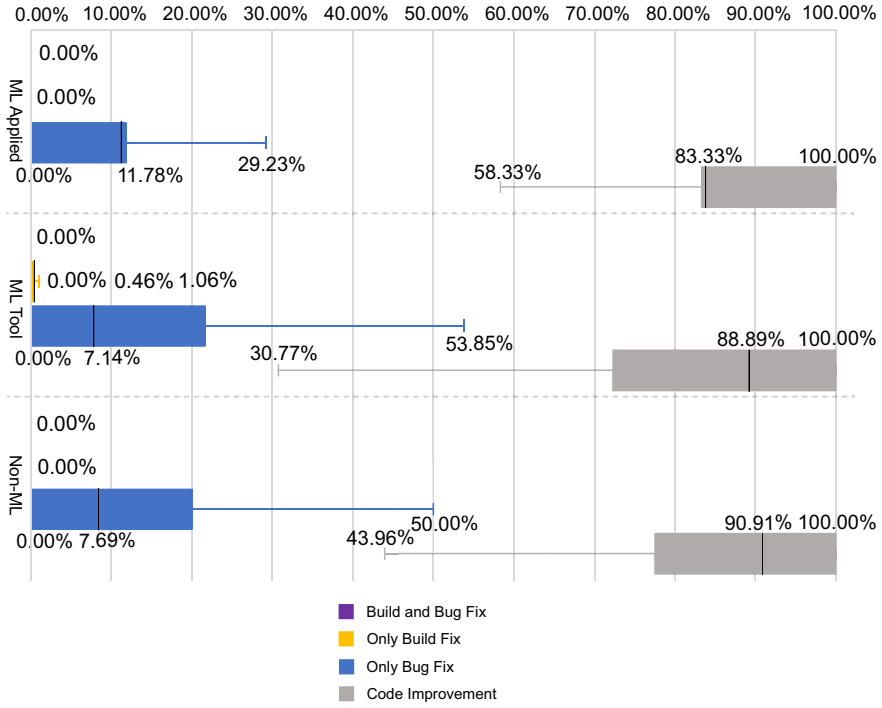


Figure 2.8: Goals of DevOps-changing Commits (Outliers points hidden, 3 quartile-values shown if different)

In a typical development cycle, bugs and problems may be detected directly by the developer through local unit testing, or be reported externally by either customers or testers. In a project that adopts CI tools, program bugs, test failures, DevOps tools' misconfigurations and other problems may be detected and reported by the CI system.

For ML Applied projects, the lower percentages of bug-fixes shown in Figure 2.8 may imply that these projects are experiencing less build breakages and bugs. But in reality, the ANCOVA analysis for ML Applied projects in Table 2.16 indicates that there is no correlation between the adoption of Test and Code Analysis tools and a reduction in the percentages of these fixes. This indicates that ML Applied projects are not using these tools

Source	Sig.	Partial Eta Squared	Details
CI * Deployment * Analyzer	.068❄	.004	Adoption of CI, DA and CA Tools
Intercept	.087❄	.004	Intercept of the model
Build * CI	.112❄	.003	Adoption of Build and CI Tools

R Squared = .059 (Adjusted R Squared = .007)

Table 2.16: ANCOVA analysis of bug-fix commit goal for Applied projects (❄ marks statistically non-significant variables, table is shown for illustrative purposes)

efficiently in order to remedy the bugs that may arise in their code. In addition, we did not find any correlation between team size or other covariates considered and bug-fixes. This implies that this misuse of Test and Code Analysis tools is present within the majority ML Applied projects, regardless of a project's properties.

Source	Sig.	Partial Eta Squared	Details
Intercept	<.001	.060	Intercept of the model
Build * Analyzer * Test	.016	.012	Adoption of Build, CA and Test tools

R Squared = .136 (Adjusted R Squared = .062)

Table 2.17: ANCOVA analysis of bug-fix commit goal for Tool projects (Only statistically significant variables are shown)

Moving on to ML Tool projects, a clear correlation is found between the adoption of Build, Code Analysis and Test tools within a project and bug-fixing commits being performed within it. Since the goals of Code Analysis and Test tools is to allow developers to find bugs and issues with their code-base, we interpret the increase of bug-fixing commits of ML Tool projects that adopted them as a sign of efficient use of these tools by these projects.

Concerning Non-ML projects, a correlation is found between the adoption of Build, CI, Code analysis, Test and Deployment Automation tools within a project and bug-fixing commits being performed within it. Similar to ML Tool projects, we interpret the increase of bug-fixing commits of Non-ML projects that adopted the different categories of DevOps tools, especially those designed to allow bug-detection as a sign of efficient use of these tools by these projects.

Across all projects categories, no correlation between Build fix percentage and Code

Source	Sig.	Partial Eta Squared	Details
Analyzer	<.001	.006	Adoption of CA tool(s)
Intercept	.005	.004	Intercept of the model
Deployment * Test	.020	.003	Adoption of DA tool(s)
Build * CI	.025	.003	Adoption of Build,CI tool(s)
Build * Analyzer	.033	.003	Adoption of Build,CA tool(s)
Build * Test	.047	.00	Adoption of Build, Test tool(s)

R Squared = .045 (Adjusted R Squared = .023)

Table 2.18: ANCOVA analysis of bug-fix commit goal for Non-ML projects (Only statistically significant variables are shown)

Analysis tool adoption or any other variable was found within the ANCOVA analysis. This indicates that Build failures and the corresponding Build fixes are not affected by variability within projects or project categories, and that there is no evidence that the adoption of a specific tool or tool type such as code analyzers will influence build failures and subsequent build-fixes. A summary of the analyses we performed for DevOps change goals is within Table 2.19.

2.4.2.4 Interpretation of results

Using these findings, it's evident that developers working on ML Applied projects make numerous updates to their DevOps configuration files that are also smaller than those of ML Tools project. By comparison, developers behind ML Tool projects overall did a smaller number of updates to their DevOps configuration files, that were larger in size. Non-ML projects had frequencies of DevOps-files updates similar to those of ML Applier projects, with a bigger variance in update-size in comparison to both ML categories. The frequency and size of updates, measured through the commit-ratio and DevOps code churn of ML Applied DevOps updates was linked to their adoption of certain DevOps tools categories, while no such correlations were found for ML Tool and Non-ML projects. The majority of DevOps updating commits of all projects categories had concerns that are not immediately related to the CI infrastructure which are in turn configured by DevOps configuration files. However, through the ANCOVA analyses we performed, we found that the adoption of Code Analysis, Test and other DevOps tools by ML Tool and Non-ML projects correlates with an increase in their bug-fixes. This signals that these tools are being efficiently used within these projects to detect bugs and the large effect size in the ANCOVA model signify this effect has important consequences on the number of bug-fixing commits. However, while adopting

Category	Most important variables affecting DevOps bug-fix commit	Interpretation
ML Applied	None	An ML Applied projects' adoption of certain DevOps tool categories or a combination of these categories is not linked to an increase in its Bug fixes
ML Tool	Build * Analyzer * Test	An ML Tool projects' commits which modify DevOps-files and fix bugs increase when Build, Code Analysis, and Test tools are adopted by them. This implies that these tools are being efficiently used to find and subsequently fix bugs.
Non-ML	Deployment * Test, Build * CI, Build * Analyzer, Build * Test	A Non-ML projects' commits which modify DevOps-files and fix bugs increase when combination of Build, Code Analysis, Test, Deployment, CI tools are adopted by them. This implies that the tools from these categories which facilitate bug-locating are being efficiently used to find and subsequently fix bugs.

Table 2.19: Summary of ANCOVA analyses results for DevOps change goals

these tools is linked with larger and more frequent updates to DevOps configuration files within ML Applied projects, it is not linked with an increase in bug-fixing commits. This hints at a less efficient adoption of these tools which requires more frequent updates with more effort but no noticeable results on bug-fixes within ML Applied projects

Finding 2:

While ML Applied DevOps configuration files updates are more frequent, they are smaller in size than those of ML Tool DevOps configuration files, are less concerned with CI Build fixes, and imply that DevOps tools are being used less efficiently within these projects.

2.4.3 DevOps Adoption Advantages

RQ3

What are the advantages of adopting DevOps tools across the different types of projects?

2.4.3.1 Commit Frequency

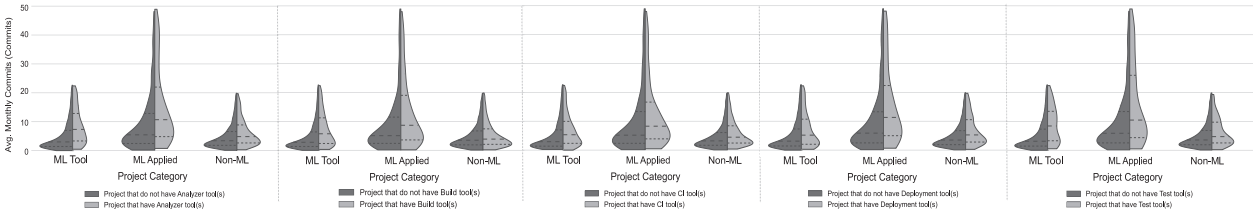


Figure 2.9: Commit Frequency in correlation to Project Type and DevOps tool adoption (Outliers removed with IQR [273])

Among the goals of the adoption of DevOps tools and practices within software projects is to increase the rate at which developers share their code with other stakeholders within their teams, which in-turn is measured with the frequency of commits that developers make during a specific development period. As illustrated in Figure 2.9, the projects that adopted 1 or more specific types of DevOps tools had generally higher monthly commit frequencies. This was especially true for projects that adopted CI, Deployment Automation and Testing tools, where the increase in commit frequencies was significant across all types of projects. In addition, ML Tool projects tend to see more frequent commits than ML Applied projects, which in turn have more frequent commits than Non-ML projects.

Source	Sig.	Partial Eta Squared	Details
Intercept	<.001	.043	Intercept of the model
DevOps	<.001	.013	DevOps tool(s) adoption
N_Pr_Rejected	<.001	.004	Number of Pull requests rejected
N_Pr_Core_Rejected	.002	.003	Number of Pull requests by core developers rejected
Age In Days	.003	.003	Project's age
Team Size	.004	.003	Project's team size
N_Stars	.013	.002	Number of stars
N_Pr_Core_Open	.036	.002	Number of Pull requests by core developers opened
N_issues_Open	.044	.001	Number of issues opened

R Squared = .185 (Adjusted R Squared = .182)

Table 2.20: ANCOVA analysis of Commit frequency for ML Applied projects (Only statistically significant variables are shown)

When statistically analyzing the Commit frequency through ANCOVA for ML Applied projects, as illustrated in Table 2.20, it's clear that DevOps tool adoption has a significant and important effect on the increase of monthly commit averages, especially since DevOps adoption is the variable with the largest effect size within the ANCOVA model.

Source	Sig.	Partial Eta Squared	Details
Team Size	<.001	.061	Size of the project's team
Intercept	<.001	.037	Intercept of the model
N_issues_Open	<.001	.011	Number of Pull requests opened
DevOps	.018	.005	Adoption of DevOps tool(s)
N_Forks	.029	.005	Number of Forks

R Squared = .198 (Adjusted R Squared = .189)

Table 2.21: ANCOVA analysis of Commit frequency for ML Tool projects (Only statistically significant variables are shown)

For ML Tool projects, the ANCOVA analysis in Table 2.21, shows that DevOps tools adoption by these projects also has an important effect on the increase of their monthly commit averages. However, the size of a project's team and the number of open issues it has seem to have a larger effect than its DevOps adoption on its commit averages.

For Non-ML projects, the ANCOVA analysis in Table 2.22, shows that DevOps tools

Source	Sig.	Partial Eta Squared	Details
Team Size	<.001	.015	Project's team size
Intercept	<.001	.007	Intercept of the model
N_Pr_Core_Rejected	<.001	.005	Number of Pull requests by core developers rejected
N_Pr_Rejected	<.001	.003	Number of Pull requests rejected
N_Pr_Merged	.004	.002	Number of Pull requests merged
DevOps	.006	.002	Adoption of DevOps tool(s)
N_Pr_Core_Merged	.016	.002	Number of Pull requests by core developers merged

R Squared = .057 (Adjusted R Squared = .054)

Table 2.22: ANCOVA analysis of Commit frequency for Non-ML projects (Only statistically significant variables are shown)

adoption by Non-ML projects positively affects its monthly commit averages. However, the size of a project's team and other variables related to its pull requests have a larger effect than its DevOps adoption on its commit averages.

The summary of our findings through the ANCOVA analysis linked to the average monthly Commits metric is illustrated in Table 2.23. Applying the one-way ANOVA test on this metric across the different categories, we obtain a p-value of s 7.29e−13, implying significant statistical difference regarding the average monthly commit frequency metric between the three groups of projects.

2.4.3.2 Merging Frequency

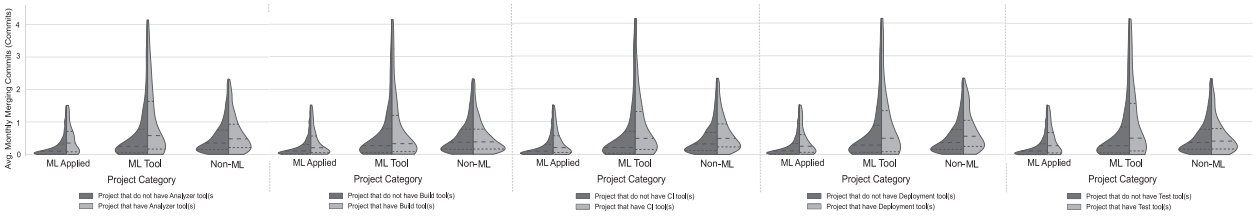


Figure 2.10: Merging Commit Frequency in correlation to Project Type and DevOps tool adoption(Outliers removed with IQR [273])

Increasing the rate at which developers merge their code with other code branches, thus increasing their code integration, is also a crucial goal of DevOps practices and tools. Merges

Category	Most important variables affecting Commit Frequency	Interpretation
ML Applied	DevOps, N_Pr_Rejected, N_Pr_Core_Rejected, Age In Days, Team Size, N_Stars, N_Pr_Core_Open, N_issues_Open	An ML Applied projects' adoption of DevOps has the largest effect on its monthly commits. Other factors such as its Number of rejects PRs and Team-size also affect this metric.
ML Tool	Team Size, N_issues_Open, DevOps, N_Forks	An ML Tool project's adoption of DevOps has an important effect on its monthly commits, however, other factors such as its Team-size have a larger effect on this metric.
Non-ML	Team Size, N_Pr_Core_Rejected, N_Pr_Rejected, N_Pr_Merged, DevOps, N_Pr_Core_Merged	A Non-ML project's adoption of DevOps has an important effect on its monthly commits, however, other factors such as its Team-size have a larger effect this metric.

Table 2.23: Summary of ANCOVA analysis results of Commit frequency

are represented with merging commits in a Git repository, and the frequency of branch merges is measured with the frequency of merging commits that developers make within a specific development period. As represented in Figure 2.10, the projects that adopted a specific type or more of DevOps tools had generally higher monthly merge commit frequencies. This was especially true for projects that adopted Analyzer, CI, and Deployment Automation tools, where the increase in merge frequencies was significant across all types of projects. ML Tool projects tend to have more frequent merges than Applied projects, which in turn have more frequent commits than Non-ML projects.

To examine the relationship between DevOps tools' adoption and the frequency of merge commits, we built ANCOVA models for the different project categories. For ML Applied projects, this model is represent in Table 2.24. Similar to the results found within Section 2.4.3.1 it's clear that adopting DevOps tools has a statistically-significant and important effect on the increase of monthly merge averages for ML Applied projects. DevOps adoption is the variable with the largest effect size within the ANCOVA model, indicating that DevOps adoption has the highest positive influence on Merge commit rates within ML Applied projects.

Moving on to ML Tool projects, Table 2.25 shows that adopting DevOps tools also has

Source	Sig.	Partial Eta Squared	Details
Intercept	<.001	.018	Intercept of the model
DevOps	<.001	.011	Adoption of DevOps tool(s)
Age In Days	.005	.003	Age of the project
N_Stars	.016	.002	Number of stars
N_Pr_Merged	.019	.002	Number of Pull Requests merged

R Squared = .258 (Adjusted R Squared = .255)

Table 2.24: ANCOVA analysis of Merge Commit frequency for Applied projects (Only statistically significant variables are shown)

Source	Sig.	Partial Eta Squared	Details
Team Size	<.001	.047	Size of the project's team
Intercept	<.001	.022	Intercept of the model
DevOps	.021	.005	Adoption of DevOps tool(s)

R Squared = .143 (Adjusted R Squared = .133)

Table 2.25: ANCOVA analysis of Merge Commit frequency for Tool projects (Only statistically significant variables are shown)

a statistically-significant and important effect on the increase of monthly merge averages for ML Tool projects. However, it's important to note that an ML Tool project's team-size has a much larger effect on Merge commit rates within ML Tool projects.

Source	Sig.	Partial Eta Squared	Details
Team Size	<.001	.052	Size of the project's team
Age In Days	<.001	.006	Age of the project
Intercept	<.001	.004	Intercept of the model
N_Forks	<.001	.003	Number of forks
N_Stars	<.001	.003	Number of stars
N_Pr_Rejected	<.001	.003	Number of pull requests rejected
N_Pr_Core_Rejected	.003	.002	Number of pull requests by core developers rejected

R Squared = .086 (Adjusted R Squared = .083)

Table 2.26: ANCOVA analysis of Merge Commit frequency for Non-ML projects (Only statistically significant variables are shown)

Through the ANCOVA analysis on Non-ML projects, shown in Table 2.26, it seems that DevOps adoption has no effect on Non-ML merge rates. To better investigate this contradiction with existing findings regarding DevOps tool adoption on merge frequency [109], we performed a detailed analysis on the effects of the adoption of the different categories of DevOps tool categories, such as Build Tools, CI Tools, etc., on Non-ML merge frequency, which is illustrated within Table 2.27.

Source	Sig.	Partial Eta Squared	Details
Team Size	<.001	.042	Size of the project's team
CI * Analyzer * Test	<.001	.012	Adoption of CI, CA and Test tools
Build * CI * Deployment * Analyzer	<.001	.006	Adoption of Build, CI, DA and CA tools
Build * CI * Analyzer	<.001	.006	Adoption of Build, CI and CA Tools

R Squared = .144 (Adjusted R Squared = .135)

Table 2.27: Detailed ANCOVA analysis of Merge Commit frequency for Non-ML projects (Only statistically significant variables are shown)

In this model, the statistically significant variable with the second largest effect size is the adoption of CI tools, Analyzer tools and Test tools, implying that these specific tool categories are more likely to increase the merge frequency of Non-ML projects, versus the adoption of any combination of tools, which apparently has no effect on the number of monthly merges.

Category	Most important variables affecting Merging Commit Frequency	Interpretation
ML Applied	DevOps, Age In Days, N_Stars, N_Pr_Merged	An ML Applied projects' adoption of DevOps has the largest effect on its monthly merging commits. Other factors such as its number of stars and Number of Pull requests merged also affect this metric.
ML Tool	Team Size, DevOps	An ML Tool project's adoption of DevOps has an important effect on its monthly commits, however, Team-size has a larger effect this metric.
Non-ML	Team Size, CI * Analyzer * Test, Build * CI * Deployment * Analyzer, Build * CI * Analyzer	An Non-ML project's adoption of certain DevOps tool categories at the same time, such as adoption CI, Code Analysis and Test tools, has an important effect on its monthly merging commits. However, its Team-size has a larger effect this metric.

Table 2.28: Summary of ANCOVA analyses results for Merging Commits frequency

The summary of our ANCOVA analyses in relation to the Average Monthly Merging Commits metric is detailed in Table 2.28. Applying the one-way ANOVA test on the Average Monthly Merging Commits metric across the different project categories, we obtain a p-value of s 1.61e−13, implying that there is a significant statistical difference between the three groups of projects.

2.4.3.3 Issue Duration

Allowing the quick resolution of problems and shortening down-time are also some of the purported goals of adopting DevOps within a software project. To measure the effectiveness of teams at resolving such problems, we used the average issue duration metric to approximate the duration an issue takes to be resolved after it's opened within a specific project, in

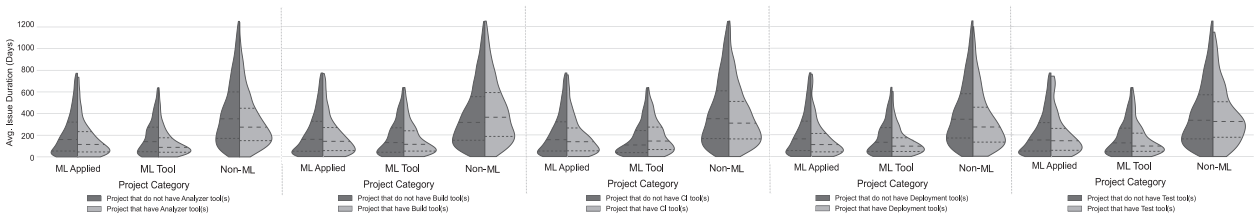


Figure 2.11: Average Issue Duration in correlation to Project Type and DevOps tool adoption(Outliers removed with IQR [273])

accordance to a project's category and its adoption of one or more types of DevOps tools. As illustrated in figure 2.11, adopting any type of DevOps tools corresponds to a quicker resolution of issues, especially the adoption of Analyzer, CI, Deployment Automation and Testing tools. Furthermore, ML Tool projects tend to have quicker resolution of issues than Applied projects, which in turn have a quicker resolution then Non-ML projects.

Source	Sig.	Partial Eta Squared	Details
N_issues_Open	<.001	.033	Number of issues open
Intercept	<.001	.032	Intercept of the model
Age In Days	<.001	.021	Project's age
DevOps	<.001	.012	Adoption of DevOps tool(s)
N_Pr_Rejected	<.001	.007	Number of pull requests rejected
N_Pr_Core_Rejected	<.001	.006	Number of pull requests by core developers rejected
Team Size	.003	.003	Project's team size
N_Pr_Core_Open	.004	.003	Number of pull requests opened by core developers
N_Pr_Merged	.009	.003	Number of pull requests merged
N_Pr_Core_Merged	.033	.002	Number of pull requests by core developers merged

R Squared = .096 (Adjusted R Squared = .092)

Table 2.29: ANCOVA analysis of Average Issue duration for Applied projects (Only statistically significant variables are shown)

When analyzing the effect of the adoption of DevOps tools on issue durations of ML Applied projects, as illustrated in the ANCOVA analyses in Table 2.29, it's clear that it has a statistically significant and important effect on decreasing the average issue durations

across all project categories. However, the number of issues open and the age of the project seem to have larger effects than DevOps adoption.

Source	Sig.	Partial Eta Eta	Details
Intercept	<.001	.095	Intercept of the model
N_issues_Open	<.001	.022	Number of issues open
Age In Days	<.001	.017	Age of the project
N_Pr_Core_Open	.019	.006	Number of pull requests opened by core developers
DevOps	.045	.004	Adoption of DevOps tool(s)
N_Pr_Open	.046	.004	Number of pull requests open

R Squared = .094 (Adjusted R Squared = .083)

Table 2.30: ANCOVA analysis of Average Issue duration for Tool projects (Only statistically significant variables are shown)

Moving on to the ANCOVA analysis regarding issue durations of ML Tool projects illustrated in Table 2.30, it's clear that it has an important effect on decreasing the average issue durations. However, similar to ML Applied projects, the number of issues open and the age of the project seem to have larger effects than DevOps adoption.

By observing the ANCOVA analysis of the issue durations of Non-ML projects illustrated in Table 2.31, it's clear that it has an important effect on decreasing average issue durations. However, other factors, such as the number of pull requests open and the age of the project seem to have larger effects than DevOps adoption.

A summary regarding the ANCOVA analyses linked to the average issue duration metric is illustrated in Table 2.32. Applying the one-way ANOVA test on the Average Monthly Merging Commit metric across the different categories, we obtain a p-value of s 1.02e−174, implying significant statistical difference between the three groups of projects.

2.4.3.4 Code Quality

In addition to positively influencing the code sharing rates and issue resolution durations, DevOps is also posed as a method of improving the quality of development processes of a project as well as its code base. To evaluate the validity of this claim, we used the state-of-the-art tool SonarQube [299] via the method described in Section 2.3.3.3 in order to evaluate the quality of the projects within our dataset. We were able to successfully generate code quality reports for 2566 ML Applied projects, 969 ML Tool projects and 3320 Non-ML projects, forming respectively 88.02%, 86.82% and 81.45% of the total number of projects

Source	Sig.	Partial Eta Squared	Details
N_Pr_Open	<.001	.101	Number of Pull requests opened
Age In Days	<.001	.076	Age of the project
N_issues_Open	<.001	.057	Number of Issues opened by core developers
Intercept	<.001	.049	Intercept of the model
N_Pr_Core_Open	<.001	.043	Number of Pull requests opened by core developers
Team Size	<.001	.027	Size of the project's team
N_Pr_Rejected	<.001	.012	Number of Pull requests rejected
DevOps	<.001	.008	Adoption of DevOps tool(s)
N_Stars	<.001	.007	Number of stars of project
N_Pr_Core_Rejected	<.001	.006	Number of Pull requests by core developers rejected
N_Forks	.001	.003	Number of forks of a project

R Squared = .327 (Adjusted R Squared = .325)

Table 2.31: ANCOVA analysis of Average Issue duration for Non-ML projects (Only statistically significant variables are shown)

Category	Most important variables affecting Issue Duration	Interpretation
ML Applied	N_issues_Open, Age In Days, DevOps, N_Pr_Rejected,N_Pr _Core_Rejected, Team Size, N_Pr_Core_Open, N_Pr_Merged, N_Pr_Core_Merged	An ML Applied projects' DevOps adoption helps it reduce its issue duration, however, other factors such as its numbers of issues open and its age have a larger effect on these durations.
ML Tool	N_issues_Open, Age In Days, N_Pr_Core_Open, DevOps, N_Pr_Open	An ML Tool project' DevOps adoption helps it reduce its issue duration, however, other factors such as its numbers of issues open and number of PRs open have a larger effect on these durations.
Non-ML	N_Pr_Open, Age In Days, N_issues_Open, N_Pr_Core_Open, Team Size, N_Pr_Rejected, DevOps, N_Stars, N_Pr_Core_Rejected, N_Forks	A Non-ML project' DevOps adoption helps it reduce its issue duration, however, other factors such as its age and number of PRs open have a larger effect on these durations.

Table 2.32: Summary of ANCOVA analyses results for Average Issue Duration

from their categories. SonarQube was unable to process some projects due to problems such as software incompatibility, as the free version is not compatible with C and C++ projects, missing dependencies, internal memory management issues, among other reasons.

Source	Sig.	Partial Eta Squared	Details
Intercept	0.000	0.435	Intercept of the model
DevOps	<0.001	0.08	Adoption of DevOps
Age In Days	0.002	0.004	Age of a project
N_issues_Open	0.006	0.006	Number of Issues Open
Team Size	0.011	0.003	Age of a project

R Squared = .065 (Adjusted R Squared = .059)

Table 2.33: ANCOVA analysis of Reliability for ML Applied projects

Source	Sig.	Partial Eta Squared	Details
Intercept	0.000	0.992	Intercept of the model
DevOps	<0.001	0.004	Adoption of DevOps

R Squared = .011 (Adjusted R Squared = .004)

Table 2.34: ANCOVA analysis of Maintainability for ML Applied projects

Through the ANCOVA analyses within Table 2.33, it's clear that an ML Applied project's reliability is correlated and most improved by its DevOps adoption. It's also interesting to note that a project's age, team size, and number of issues have a significant effect on improving a project's reliability. Longer-lived projects with larger teams, who are more capable at keeping track of bugs, are more likely to have better Reliability metrics. Focusing on ML Applied project's Maintainability, it's clear through Table 2.34 that DevOps adoption is the only project property that is statistically correlated to this quality metric. Overall, through these two analyses, it's clear that DevOps adoption is the number one factor influencing an ML Applied project's code quality.

Moving on to ML Tool projects, it's clear through Table 2.35 that DevOps is the only statistically significant variable that affects these projects Reliability metric. However, no such correlation was found concerning the Maintainability metric, as no statistically significant variables were found within its ANCOVA analysis. This allows us to deduce that DevOps adoption only affects certain aspect of an ML Tool project's code quality, yet it is the only variable that seems to affect it, regardless of an ML Tool project's team size, age, etc.

Source	Sig.	Partial Eta Squared	Details
Intercept	0.000	0.479	Intercept of the model
DevOps	0.001	0.013	Adoption of DevOps

R Squared = .089 (Adjusted R Squared = .077)

Table 2.35: ANCOVA analysis of Reliability for ML Tool projects

Source	Sig.	Partial Eta Squared	Details
Intercept	0.000	0.476	Intercept of the model
DevOps	0.000	0.010	Adoption of DevOps
N_issues_Open	0.000	0.009	Number of Issues Open
Age In Days	0.000	0.005	Age of a project
NBForks	0.013	0.002	Number of Forks
Team size	0.005	0.002	Size of project's team

R Squared = .059 (Adjusted R Squared = .055)

Table 2.36: ANCOVA analysis of Reliability for Non-ML projects

Concerning Non-ML projects, it's clear through Table 2.36 that DevOps is the single biggest contributor to a project's improved Reliability metric. In addition, a project's number of issues open, age, number of forks and Team size all correlate to this metric, signaling that multiple factors can influence a Non-ML project's reliability. However, it's also important to note that no statistically significant variables were found within the ANCOVA analyses of the Maintainability metric.

A summary regarding the ANCOVA analyses linked to the reliability and maintainability metrics is illustrated in Table 2.37. Applying the one-way ANOVA test on these two metrics across the different categories, we obtain a p-value of 5.22e−6 for Reliability, and 0.98 for Maintainability. This is surprising as it implies significant statistical difference between the three groups of projects for the first metric, but similarity regarding the second metric, even though both are code quality metrics.

2.4.3.5 Interpretation of results

Using these five metrics and their associated statistical analyses, it's evident that employing DevOps tools of different categories has mostly correlated with an increase in the frequency of code commits, an increase in the merges across different branches, a reduced duration leading up to issue resolution, and an increase in code quality across the three different

Category	Most important variables affecting Reliability	Most important variables affecting Maintainability	Interpretation
ML Applied	DevOps, Age In Days, N_issues _Open, Team Size	DevOps	An ML Applied project's DevOps adoption, age, and Number of issues open are the most important factors that affect its code quality
ML Tool	DevOps	None	An ML Tool project's DevOps adoption is the only statistically significant factors affecting its code quality
Non-ML	DevOps, N_issues _Open, Age In Days, NBForks, Team Size	None	A Non-ML project's DevOps adoption, Number of issues open, age, Number of forks and Team size are the most important factors that influence its code quality

Table 2.37: Summary of ANCOVA analyses results for Reliability and Maintainability

types of projects. These advantages are especially prevalent when using CI and Deployment automation tools across all categories of projects.

Focusing more on ML Applied projects, it's evident that employing DevOps tools has an important and generally positive effect on the development activities, issue resolution, and code quality within these projects, thus signaling that while these projects may have a harder time employing DevOps tools, as per the findings in Section 2.4.2, they also have the most to gain from using DevOps tools within their code bases.

ML Tool and Non-ML projects that employ DevOps show mostly similar improvements in comparison to their non-DevOps counterparts, however, the improvements are not as drastic as those of the Applied ML projects.

Finding 3

All categories of projects that employ DevOps show improvements in their development, code quality and issue resolution metrics in comparison to their non-DevOps counterparts, especially in the case of ML Applied projects, supporting the claim that DevOps tools can improve the development processes of most projects they are used in.

2.5 Implications of the Proposed Study

In this section, we discuss the implications of our empirical analysis. The following is a list of actionable items we identified:

- Our analysis on DevOps adoption rates and trends, detailed in section 2.4.1, identified that ML Applied projects were slow in adopting DevOps. They also had a lower adoption across different DevOps tool categories such as Build, CI and Code Analyzer. While analyzing the exact reasons behind the barriers to adoption of DevOps tools is by ML projects is not within this work's scope, our results shed a light on the necessity for researchers to study the barriers to adopting DevOps in ML projects and identify possible improvement scopes. These may include ML DevOps task automation, DevOps tools for ML models evaluation and monitoring, etc. On the other hand, tool developers can employ program analysis [192] techniques to automatically generate ML DevOps configuration files which can lower the barriers of entry for data scientists who might be unfamiliar with DevOps concepts and practices.

- Our DevOps tool maintenance effort analysis, detailed within Sections 2.4.2.1 and 2.4.2.2, reveals that even though ML Applied projects much less adoption of DevOps than the other two categories (ML Tools and Non-ML projects), their developers are changing DevOps configuration files more frequently. This highlights the necessity of working on support for automatic synchronization of DevOps configuration files. This may be provided via change recommendation tools [345], safe refactoring tools [336], and others. These tools can help reduce maintenance overhead, and can provide technical support to developers and data scientists who may not be very familiar with DevOps tools.

- Our analysis on events that trigger DevOps file changes, within section 2.4.2.3, identified that bug-fixing commits within Tool project that alter DevOps configuration files were much more prevalent in comparison to ML Applied and Non-ML projects. This

indicates that the software maintenance research community should invest more heavily in co-evolution analysis [160] of functional code and DevOps configuration files to facilitate early bug-detection. In turn, this will save both time and resources and allow teams to invest them in improving their software product's quality and reputation, rather than resolving problems within it.

- Our analysis on DevOps adoption advantages, within section 2.4.3, identified that for all project types, adopting DevOps has positive consequences on the code sharing and code integration speed and frequency and helped decrease the duration necessary for issue resolution and improve its quality. Even though using DevOps tools for all types of projects, including ML projects, introduced adoption and maintenance overhead, it appears that the benefits of DevOps outweigh the associated costs. Thus, data scientists and ML developers should adopt DevOps tools within their projects. Furthermore, we believe that adopting DevOps tools present these benefits for all ML projects, even for those with smaller teams. This is especially prevalent in the case of ML Applied projects, which had smaller team sizes overall but generally saw larger improvements resultant of DevOps adoption than ML Tool projects.

- Software engineering educators lack concrete ideas on ML DevOps integration trends, benefits, and tools, preventing them from training students with ML DevOps skills that would allow them to build industry-ready ML-based systems. This study helps educators understand the current trends, benefits, and tools of ML DevOps in order to include up-to-date pedagogical material on ML DevOps.

2.6 Threats to Validity

Our empirical analysis has some limitations that we would like to discuss:
Construct validity: We used the code churn and commit ratio metrics to estimate DevOps configuration files maintenance efforts. However, while these metrics may not reflect maintenance effort 100% correctly, they remain representative work items for maintaining source code and other files.
Internal validity: During DevOps tools detection, we used a file name patterns list which we manually constructed. To mitigate bias, one of the co-authors performed a manual checking of DevOps configuration files and file naming patterns in both ML projects and Non-ML projects. In addition, most tools have highly specific naming conventions, so the probability of false positives is minimal. Some tools, such as logging tools, may be hosted on third-party servers and do not need to have any configuration files within a repository, but they remain

the minority among DevOps tools. Furthermore, DevOps tools that do not leave traces in files within the code repository, such as communication tools, can not be detected via our approach.

External validity: Our analysis is based on public repositories on GitHub. These results might differ for private GitHub repositories and closed repositories, including projects developed by companies. However, our project set does contain projects developed by companies, such as `tensorflow/tensor2tensor` which is backed by Google. We also estimate that at least 30% of ML Tool projects are backed by major organizations such as Microsoft and IBM. Furthermore, since we used popular organization and user-managed projects within our analysis, we expect many similarities of behavior.

2.7 Conclusion

In this study, we conducted an empirical study on 4031 ML projects and a comparative set of 4076 Non-ML projects hosted in GitHub for ML DevOps adoption, maintenance effort and benefit analysis. Through our analysis, we found evidence of a lower adoption of DevOps tools within ML Applied projects, as well as different development practices and efforts in relation to these files that tended to be less efficient than those of ML Tool and Non-ML projects. In contrast, this type of projects has the most to gain from adopting these tools, and with similar advantages for both ML Tool and Non-ML projects. To the best of our knowledge, this is the first large scale empirical study on ML DevOps adoption, maintenance effort and benefit analysis.

CHAPTER 3

Characterizing the usage of CI tools in ML projects

This work was published in ESEM 2022, the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. This work was performed in collaboration with Microsoft Research.

3.1 Introduction

"The whole point of Continuous Integration is to provide rapid feedback". This is how Martin Fowler [96], who helped popularize CI, describes it.

Similar to traditional software, ML projects rely on iteration within their development. Indeed, the automation prowess of a CI system may be a great fit for ML projects' need for iteration. However, it's notable that most CI tools were conceived before ML project development became mainstream, and that both CI tools and ML projects have their specific problems. For example, debugging CI build failures and errors can be non-trivial due to complex logs [334], and ML projects require new development processes and practices such as data engineering and model management [196], or require a different approach to existing processes in comparison to traditional software, such as the example of traditional testing being ineffective on ML projects [170]. Yet, there is a gap in research concerning the adoption of CI within ML projects, the tasks performed by CI within them, as well as the problems CI tools face when they are used in these projects. In this paper, we aim to fill these knowledge gaps by identifying the adoption rates, current practices, and common failures and errors related to CI within ML projects. We used a triangulation-based [48] method to estimate the adoption rate of CI on a set of 4031 ML projects and 4076 Non-ML projects. Then, for our detailed CI analysis, we selected 476 ML and 202 Non-ML projects out of the larger set, using the same criteria of CI adoption and main programming language on both project

categories. Using `TraVanalyzer`, our Travis CI AST analyzer, we determined the CI build goals of these projects. Using our CI log analyzer, we analyzed these ML and Non-ML projects' builds, and their associated job failure and error logs, from which we determined the CI problems these projects encountered. With our analysis, we answered 3 key questions:
RQ1: *What is the adoption rate of CI among ML projects?* Around 37.22% of ML projects have adopted CI, which is below CI's adoption rate by our set of Non-ML projects, estimated at 45.12%, as well as that of Open Source Software (OSS) overall, estimated to be between 45% and 68% [73]. We also found that Travis CI, the most popular CI tool for our Non-ML project-set and Open-source software on GitHub [148], is the most popular tool for ML projects as well, with no sign of adoption of ML-projects-specific CI tools.
RQ2: *What tasks does CI perform for ML projects?* Similar to traditional software [77], Testing and building software are the most common tasks. Code analysis is the third most common, and deployment is the least common. Surprisingly, these tasks are used more often within CI of ML projects than CI of Non-ML projects.
RQ3: *How often and Why do CI builds break in ML projects?* On average, 23.87% of the CI builds of a project fail, and 14.09% of the builds are errored, both of these build types are considered non-successful. Build breakages in general occur at similar levels between our sets of ML and Non-ML projects, but are more common in both than in Open Source Software (OSS) overall. The most common failures were caused by failed tests, errored tests, and failures related to Code Analysis, which are also common within our comparison set of Non-ML projects. However, ML failures show more variability in terms of causes linked to failure of a project's build.
In summary, we make the following contributions:

- The first comprehensive analysis of CI adoption by ML projects on GitHub.
- The first Travis CI configuration AST analyzer `TraVanalyzer` which determines CI tasks.
- The first CI log analyzer specifically-designed for the detection and classification of CI problems within Python-based ML and Non-ML projects.
- A comprehensive taxonomy of CI problems in Python-based ML and Non-ML projects, that can facilitate the fix pattern analysis.

3.2 Background

Continuous integration (CI) was first introduced by Grady Booch in 1991 [42], and this concept began to gain popularity in the early 2000s partially due to support from Martin Fowler [96]. The founding principle behind CI is frequent integration of code from the different developers of a shared repository, arising from the time-consuming and difficult task

of code integration that software projects without CI need to perform [96, 42]. In general, Continuous Integration servers and tools automate integration by automatically validating newly-pushed commits via the execution of building and testing processes. CI can also include other automated tasks like code analysis tasks, such as linting and code coverage, or deployment tasks. A variety of CI tools and services exist, and more recently, a few CI tools and services have been specifically designed for ML projects or their components. Tools and services that provide testing and versioning for ML projects, such as Kuberflow [177] and Amazon Sagemaker [292], are generally only concerned with ML models, and the corresponding ML project's code base is managed via a traditional CI tool such as Travis CI or GitHub Actions. As a result, the majority of these aforementioned tools are generally used in conjunction with traditional CI tools and cannot fully replace them. Furthermore, we found no evidence of their usage within our project set. In fact, Travis CI is also the tool that enjoys the highest usage within our set of ML projects, as detailed in Section 3.5.1.

Indeed, The majority of CI-related processes, such as building and testing, and their corresponding tools were established and designed for traditional software projects, and may not be well-suited for ML projects. For example, Unit Testing is a well-established practice of testing functional code by comparing its results against the expected results as defined by the test's author [236]. Applying this same approach to ML projects by evaluating their results on the same testing set repeatedly can cause problems with their accuracy [170]. Another example is automatic deployment, where for traditional projects, the software and its configuration are bundled into a deployable archive, such as a JAR, or deployable image, such as a Docker-image, and then are pushed to their respective endpoints. For ML projects, the most frequently updated component is the ML model [226], the deployment strategies of which may differ from those of other components. In spite of these differences, Travis CI, one of the most widely used CI tools [148], is the most popular CI tool of the ML and Non-ML projects we analyzed.

A Travis CI workflow is described via a *.travis.yml* file, written in YAML-based [60] Domain Specific Language (DSL), where certain settings-keywords can configure the environment or execute a certain process. A workflow is generally referred to as a build and can be composed of one or more stages that run sequentially, and each stage forms a specific subset of the overall build. Stages can be configured with the `stage` keyword, and by default, a build is composed of only one stage. Each stage may be composed of one or more Jobs that run in parallel, each executing the same sub-script in a specific environment different from the other jobs. This is configured via the *build matrix*, where two or more jobs can be set to run in parallel in each stage, by specifying a different environment for each. The keywords `OS` and `language`, which respectively set the Operating System of a job's container and

prepare it by installing the tools of a specific programming language, can be used multiple times and with different values. For example, `OS:linux` with `language:Java` and `OS:linux` with `language:ruby` will configure two Jobs that run in Linux containers, one of which is configured for Java projects and the other one is configured for Ruby projects.

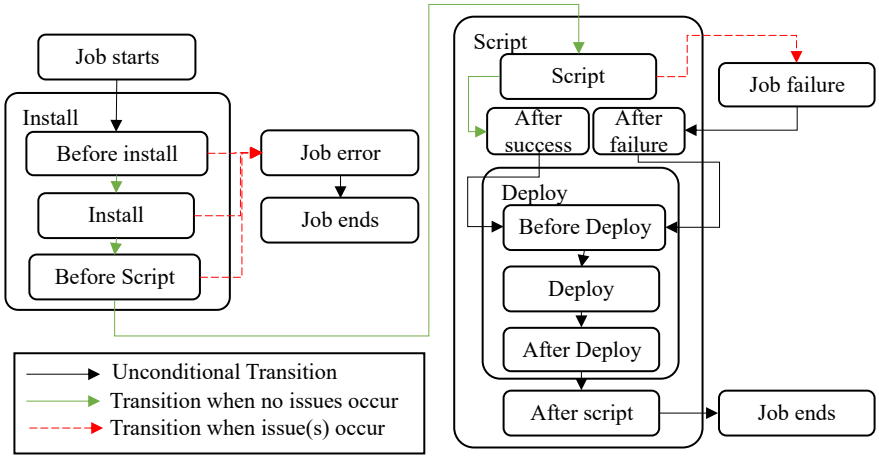


Figure 3.1: Travis CI job state machine

In further detail, a generic job's state machine is illustrated in Figure 3.1. A job has two phases, the install phase, meant for the installation of any dependencies and preparation of the environment, and the script phase, which runs the build, test, and other tasks specified by the developer. An **errored job** is a job that experiences an issue during the install phase of its execution, after which it immediately stops executing. A **failed job** is a job that experiences an issue during its script phase, after which it executes its "after failure" section, and the rest of its script phase. A **failed build** is a build with 1 or more failed jobs and no errored jobs. An **errored build** has one or more errored jobs. If no problems occur, the job(s) and the build are considered **passed**. A notable exception is that if a job experiences an issue only during its `Deploy` or `After Script` phase, it's still labeled as passed. A build can be manually canceled by the developer, giving it and all its associated jobs the **canceled** state.

3.3 Research Methodology

3.3.1 Dataset

In this work, we used two data sets: the larger one is referred to as the *Breadth Corpus*, on which we performed analysis to inform us about the state of CI adoption in ML projects, and the smaller one is referred to as the *Depth Corpus*, a subset of the breadth corpus on

which we performed detailed analysis to inform us about CI usage goals, and CI problems within ML projects.

3.3.1.1 Breadth Corpus

Our goal was to analyze CI adoption within a set of open-source and active Machine Learning (ML) projects, and a similar comparison set of Non-ML projects. We define ML projects, also referred to as ML-enabled systems, as those with the goal of producing systems that have ML capabilities as part of their features. For this task, we initially chose to analyze the data set of projects proposed by Gonzalez et al. [129]. This data set is composed of 5224 ML projects and 4101 Non-ML projects. However, we found several problems with it via manual inspection, such as the inclusion of toy projects and study guides among others. To resolve this problem, two of the authors re-curated the ML projects by reading the descriptions on their main GitHub pages and any websites linked to by those pages, and they removed 1193 projects. These projects either did not use ML, or were toy projects, or study guides, or another type of repository which did not constitute a software project. We were unable to obtain 25 Non-ML projects from the original dataset due to delisting. The new set of projects which forms our *Breadth Corpus* contains:

- **4031 ML projects:** These projects are composed of ML frameworks and libraries such as Tensorflow, as well as ML applications such as Faceswap. All of these projects employ Machine Learning based techniques or components, and they either meet a specific need for the user, or have a general-purpose usage and can be used by other developers for a specific goal.
- **4076 Non-ML projects:** These projects are considered traditional software applications, such as websites, desktop or mobile applications etc., which do not contain or use ML-based components or technologies.

3.3.1.2 Depth Corpus

After estimating the adoption rates of different CI tools as detailed in Section 3.3.2.1, we found that Travis CI was the most popular CI tool for ML projects, as detailed in Section 3.5.1, which is also true for the Non-ML projects we analyzed and open-source software in general [148]. Furthermore, for the ML projects that used Travis CI, Python is the main language for 51.06% of them, as reported by the GitHub API [111], and no other language was the main language for more than 9% of them. This aligns with the results found by Gonzalez et al. [129] concerning Python being the most popular language for ML projects. As a result, we selected Python-based ML projects with one or more Travis CI

builds since they represent the majority of CI-using ML projects. We then applied the same selection criteria of Python as a main programming language and CI-usage to our Non-ML set of projects to obtain a comparison set, which produced a smaller number of projects. This is expected since Python is a main programming language of only 14.95% of Non-ML CI-using projects. Our depth corpus is composed of:

- **476 ML** Travis CI-using Python ML projects.
- **202 Non-ML** Travis CI-using Python Non-ML projects.

3.3.2 Approach

In this section, we illustrate the different steps we took to select and analyze our project set. An overview of our approach is in Figure 3.2.

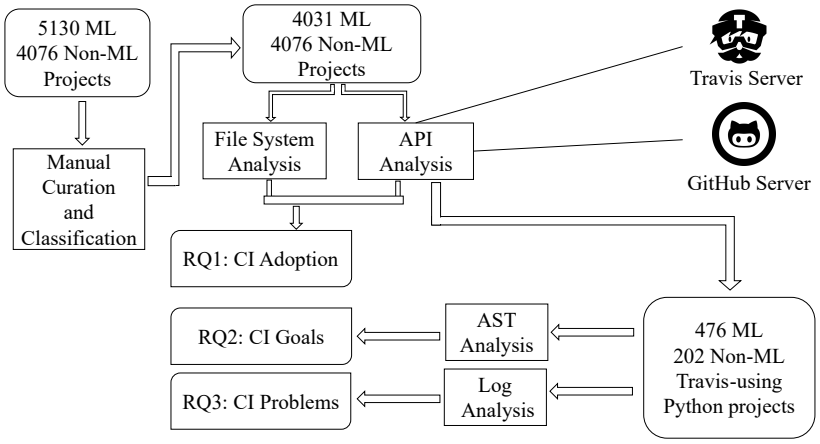


Figure 3.2: Overview of Research Methodology

3.3.2.1 CI Adoption Analysis

Currently, there is no standardized approach to determine if a GitHub repository is using a CI tool. In order to determine the adoption of different CI tools within our set of ML projects, we developed a two-pronged approach to estimate the adoption rates of the different continuous integration tools, based on methods followed by Hilton et al. [148] and Gallaba & Mcintosh [100]. First, we considered a list of CI tools proposed by Leite et al. [181], but this list included only 4 tools and was developed by examining traditional software projects. To enrich it, we selected the top 1000 ML and the top 1000 Non-ML projects in our *breadth corpus* and listed their filenames which had non-source-code extensions. Then, 2 authors attempted to match their names with the naming conventions of configuration files of CI tools in order to identify any other CI tools in our project set. We also added support for

tools such as ease.ml\ci, that rely on the same configuration files of other tools by introducing their own segments, since the filename-based approach would not allow us to detect their usage. Second, we scanned the projects' repositories [1] to determine if they had files or file segments indicative of the usage of a specific CI tool, which we collected in the previous step. Finally, we selected the CI tools with an adoption rate of 5% or more as measured by our first File-system-based approach, which was GitHub Actions and Travis-CI, and then queried their APIs and used the existence of one or more builds to establish a project's adoption of a specific CI tool. However, we found conflicts between the two data sources: some projects had CI configuration files without any builds in the corresponding CI tool's server, or vice-versa. For example, while we found 852 ML projects with Travis files in their repositories, only 559 of them also had builds on Travis's server, and we found an additional 376 ML projects which had Travis builds but did not currently have a Travis configuration file in their repository. Even more notable differences were found in the case of GitHub actions, where 858 ML projects had configuration files for this tool, but only 477 of them had GitHub Actions builds, and we did not find any ML projects which had GitHub Actions builds but no configuration files. To resolve these conflicts and avoid false positives or false negatives regarding CI adoption, we applied a triangulation-based method [48] and defined two types of CI adoption:

Historical Adoption: A project that has builds on a specific CI tool's server, but does not have the CI tool's configuration files in its current repository, is assumed to have used the CI tool in the past.

Current Adoption: A project that has builds on a specific CI tool's server and has the tool's configuration file(s) within its current repository, is assumed to be currently using the CI tool.

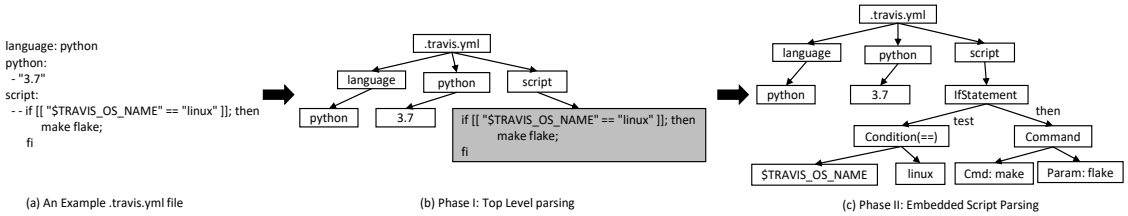


Figure 3.3: Overview of Parsing an example .travis.yml file in AST Format

This allowed us to resolve any data conflicts and answer **RQ1** regarding the popularity of CI within ML projects. The complete list of CI tools we considered is: AppVeyor [22], Buildbot [47], CircleCI [63], Cloud Build [67], CodeBuild [3], GitLab CI [121], Jenkins [1], Travis CI [2], GitHub Actions [110], VSTS [212] and ease.ml\ci [81].

[1] Last updated on 08/13/2021

3.3.2.2 CI Task Analysis

CI configuration files (e.g., .travis.yml) define continuous integration tasks such as build, test, deployment, etc. Travis CI adopted a DSL that is based on YAML. However, analyzing a Travis CI configuration file is not trivial as it can invoke system commands, external shell scripts, Python scripts, among others, and can also include steps to integrate ML components (e.g., data, model, etc.). To overcome these challenges, we developed the first .travis.yml AST [225] analyzer, `TraVanalyzer`, to parse Travis configuration files from our *Depth Corpus* projects, and applied a command clustering approach to group commands to allow for manual annotation and analysis. Details of the parser and the AST-based command clustering approach are discussed in the subsequent paragraphs.

AST Parsing of CI Configuration File:

Since Travis CI configuration files are written in a domain-specific language (DSL) language extended from YAML, we chose to extend the Java-based YAML parser `DocConverter` [24] to extract top-level entities of the configuration file. Figure 3.3 shows such an example where, in Phase I, top-level configurations are parsed in an AST format. However, based on Travis CI Documentation [58], `install, script, before_install, before_script, after_script, after_success, after_failure` job phases can invoke external system commands and bash syntax that includes if-else conditions, looping, variable usage, etc. Figure 3.3 Phase I shows an example of that with a script-block that contains an embedded bash script that uses an if-condition to invoke the `make flake` command. We developed a Bash script parser that can parse and extract such embedded scripts, and generate their ASTs. Since Bash scripts support variable assignment and usages, we applied a data-flow analysis [94] on the extracted AST to analyze its condition checks. However, in many cases, the scripts use system environment variables such as `TRAVIS_OS_NAME` for if-else conditions. Since system variables cannot be determined by just analyzing the embedded scripts, we considered these conditions as always true. After generating the AST for the embedded script, we extended the Phase I AST to generate the Phase II AST with Bash annotations. We used the Phase II AST as depicted in Figure 3.3(c) for the CI task analysis of ML projects. On a programmatic level, `DocConverter` allows us to ingest the file into an object, which we then convert to a Tree object, as represented in Phase I. We then apply the embedded script parsing to generate the object represented in Phase II.

Command Clustering and Task Analysis:

With the ASTs generated for the Travis CI configuration file, we can analyze the different stages and properties described in it, such as the language and OS. However, the purpose of external tools and commands invoked is difficult to determine. To solve this problem, we extracted the commands from the Travis AST objects, generated from the .travis.yml

files belonging to the ML and Non-ML projects within our depth corpus, and applied AST-clustering based on the commands' names. We chose to omit the commands' parameters as they are often project-specific. In total, we extracted 258 distinct commands and/or tools, corresponding each to one cluster. Two of the authors then manually reviewed the tools and commands documentations' to categorize them into build, test, code analysis, and deployment tools. This categorization is used in order to answer **RQ2** regarding the CI tasks of our *Depth Corpus* projects.

3.3.2.3 CI Problem Frequency and Taxonomy Study

Continuous integration workflows may experience failures or errors due to a variety of reasons. To determine the build breakage[2] and success rates of the projects in our *Depth Corpus*, we used the Travis CI API via PyTravisCI [254] to obtain information about their respective builds and jobs, as well as the logs pertaining to their failed and errored jobs. Within this analysis, we opted to use the CI information of each project within one year of a project's most recent build[3]. We opted for this moving window of dates in order to consider only the most recent builds of each project, and we chose this relatively large window to minimize the chance of excluding information from less active projects. We found a total of 79868 builds of our ML projects and 7519 builds of our Non-ML projects using this moving-window criterion. The higher total number of builds for ML projects can be explained by the inclusion of projects such `RasaHQ/rasa_core` and `Cloud-CV/EvalAI` with builds as high as 4715 and 1680 in the selected window, while the highest number of builds for a single project was 733 for Non-ML projects within `numenta/nupic`. We then applied the filtering process proposed by Gallaba et al. [99] to avoid including duplicate builds in our analysis, which resulted in the removal of 6157 passed, 2144 failing, and 1163 errored builds of ML projects, as well as the removal of 694 passed, 708 failing, 357 errored builds of Non-ML projects. Finally, we generated the average percentages of successful, failed, errored, and canceled builds for each project, considering only their filtered builds.

Examination of existing taxonomies:

When it comes to classifying the issues causing a job failure or a job error, a number of taxonomies exist: Beller et al. [33] classify build failures into two categories depending on whether or not a build failed because of a test, but this has limited usage since it does not clarify the reasons behind a build failure when tests are not the cause. Durieux et al. [78] analyzed build failures to extract the issue causing a job failure. But some of their issue categories were overly specific, such as the "Gem file not found" category that does not

[2]a broken build refers to a failed or errored build in the context of CI

[3]before August 13th, 2021

generalize to projects not using Ruby, and some of them lacked detail, such as the "Travis Limitation" category which can be a log file length restriction, a timeout, or another Travis-related limitation. Rausch et al.'s work [261] classifies failures into a variety of categories related to Java projects but doesn't generalize to other languages or projects other than those studied. For example, their "androidsdk" category would only occur in projects using the Android SDK. Finally, none of these aforementioned taxonomies focus on ML projects, as they were developed for traditional software. When testing the preexisting log analyzers associated with the aforementioned taxonomies, we found none of them reached a satisfactory saturation rate, which in our context is the percentage of failure logs within which a failure type was detected. For example. Travis Listener's Log Parser [78] failed to detect the failure type within 71% of 7655 randomly selected job failure logs pertaining to projects from our *Depth Corpus*, and Rausch et al.'s [261] classifier would not work since it is intended for Java projects, while the projects we are analyzing are Python-based. Overall, we found these existing taxonomies either have a very broad categorization that lacks detail, or a narrow categorization that does not directly relate to the failure and error classifications of the Travis CI builds that we specified. Furthermore, none of these taxonomies or their associated tools were designed for Python-based ML projects, and we found them inadequate for the job logs we aimed to analyze.

Creation of a new taxonomy:

In order to resolve the aforementioned problems and answer **RQ3** regarding the underlying reasons behind job failures and job errors, we created a new taxonomy and its associated log analyzer. We used open coding [173] to build our taxonomy of failures and errors and their associated log analyzer composed of regexes and scripts. During this step, we considered the following builds, selected via the filtration process described earlier: 1262 failed builds and 1144 errored builds belonging to Non-ML projects, and 19621 failed build and 7014 errored builds belonging to ML projects. Then, we attempted to obtain the logs pertaining to job failures connected to these broken builds, and succeeded at downloading 35965 and 7153 job failure logs of ML and Non-ML projects respectively. In the following step, we randomly selected our first set of 100 failure log files, which belonged to 71 different projects[4], from the set of all the failure logs. Then, we manually analyzed them to extract regular expressions belonging to different failure types, each of which is associated with one CI task. Fourth, we wrote a log analysis script that uses these regexes for classifying failures within their respective sub-types. Finally, we tested our log analyzer on the remaining set of failure logs to estimate our saturation rate. The rate was lower than 90% for both ML and Non-ML

[4]no uniqueness criterionwas applied regarding projects or their categories during the process of log-selection as to not influence the randomness of the process

projects, so we repeated the previously-described process on a second set of 100 randomly-selected failure log files from 33 different projects to enrich our initial set of regexes. This allowed us to reach a saturation rate of 91.59% and 91.15% on ML and Non-ML job failure logs, respectively. In total, we used 2 sets each containing 100 log files to construct our failure taxonomy. For the error taxonomy, we repeated this same process by analyzing 2 sets of 100 error log files from 90 different projects, randomly selected from 18857 error logs, 14121 of which belonged to ML projects and 4736 of which belonged to Non-ML projects. This allowed us to reach a saturation rate of 95.05% and 96.16% on ML and Non-ML on job error logs, respectively.

This process has proven complex and time-consuming due to the variety of tools being used across Python projects, and their different logging conventions. For example, a test failure may be reported in different ways across different projects all using the same Pytest framework. It may be a line showing a test summary such as `=== 2 failed, 91 passed ===`, or instead, listing each failed test e.g: `FAIL Test1, FAIL Test2`, or a combination of both outputs or others. This output variability is also noted within other tools and other processes, such as linting and code coverage. To resolve these problems, our log analyzer had to rely on a high number of regexes to identify a large amount of different output-patterns, it relies on 110 regexes to analyze job failure logs, and 33 regexes to analyze job error logs. These regexes correspond to popular python tools that are used by both the ML and Non-ML projects we studied, making it possible to reuse the log analyzer for other Python-based CI-using software projects.

To better organize the different types of failures we identified via our log analyzer, we constructed our job failure taxonomy via the following process: 2 authors followed the card sorting [92] method of grouping similar failures and errors as identified by the regexes into multiple main and subgroups. Then, in the case of job failures, this hierarchy was simplified by merging most similar sub-groups leaving only the ML-specific failure types at the 3rd level of the taxonomy tree, to help increase the ease of understanding and generalization of this taxonomy, after which 6 main and 18 subgroups remained. We constructed the error taxonomy using the same method, but we chose to only leave 4 main categories as they contained less internal variability within them in comparison to those of the failure taxonomy.

3.4 Evaluation of Analysis Tools

Evaluation of TraVanalyzer: To evaluate `TraVanalyzer`'s AST generation, 2 authors manually evaluated 100 ASTs generated from 100 randomly selected .travis.yml files belong-

ing to projects from our *Depth Corpus*, and found that the files were parsed correctly. This confirms the robustness of our tool regarding the generation of accurate ASTs. Regarding `TraVanalyzer`'s efficacy in correctly allowing us to determine the CI usage goals, 3 of the co-authors manually inspected Travis CI configuration files separately to categorize different tasks executed by the CI. Fleiss's kappa coefficient [93] was used to find inter-annotator agreement prior to discussion, and was on average 0.78 across the different categories, indicating substantial agreement. The disagreements of analysis were resolved by discussion. After evaluating the performance of `TraVanalyzer` on the manually labeled data, we found an average Precision, Recall, and F1-score of 98.44%, 95.45%, and 96.92%, respectively, for identifying the different build, test, code analyzer, and deployment tasks configured within the CI configurations file(s). These results confirm the correctness of the proposed tool for our purposes.

Evaluation of the Log analyzer: We tested our CI Log analyzer on a set of randomly selected logs. Two authors manually labeled 100 errored job logs and 100 failed job logs, belonging to jobs from our Depth Corpus. These logs were not used to construct the log analyzer. Fleiss's kappa coefficient [93] was on average 0.73 across the 4 Job Error types, and 0.78 across the six main Job Failure types, indicating substantial agreement, and any disagreements were then resolved by discussion. Our log analyzer achieved an average Precision of 95.42%, average Recall of 92.84%, and F-1 score of 94.11% across the main Job Failure types. It also achieved an average Precision of 99.1%, an average Recall of 96.4%, and an F-1 score of 97.73 % across the Job Error types.

3.5 Results

3.5.1 CI Adoption rates

> ***Research Question 1:*** What is the adoption rate of CI among ML projects?

To determine the prevalence of CI within ML projects, we used the triangulation-based method detailed in Section 3.3.2.1, and applied it to our **Breadth Corpus**. This allowed us to estimate the adoption rates of the CI tools outlined in Section 3.3.2.1. The first step of our triangulation process was the File-System (FS) based process. Overall, the CI adoption rate through this method is estimated at 37.22% for all ML projects, and the CI adoption for Non-ML projects is estimated at 45.12%. Focusing on individual tools, the top 3 tools adopted by ML projects were Travis CI at 21.14%, GitHub Actions at 21.29%, and Circle CI as a distant third at 3.28%. For Non-ML projects, the top 3 tools adopted by ML projects

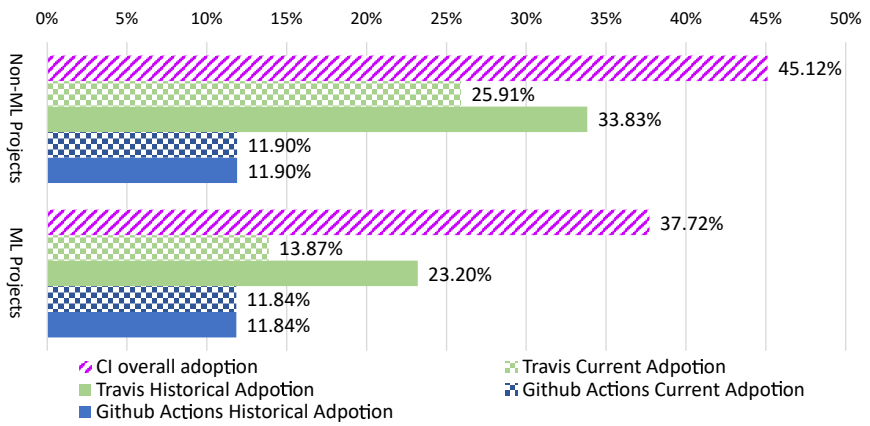


Figure 3.4: CI Tools Adoption rates (Excluding CI Tools with less than 5% adoption)

were Travis CI at 33.98%, GitHub Actions at 13.62%, and AppVeyor as a distant third at 3.04%.

However, when applying the triangulation-based method, via querying the APIs of the top 2 tools per the FS-based method for both categories of projects, Travis CI and GitHub Actions, and then consolidating them with the FS-based findings, we found that there is a mismatch between the two data sources. Details about this mismatch and the definitions of *Historical Adoption* and *Current Adoption* we chose to resolve it are in Section 3.3.2.1. The current and historical adoption rates, illustrated within Figure 3.4, reflect that the popularity of Travis CI for open-source software [148] is also evident in ML projects and our comparison set of Non-ML projects, with GitHub Actions being a close contender in terms of current adoption for ML projects. This is surprising given the age of GitHub Actions, as support for CI was only added to it in public beta in August 2019 [110]. Overall the adoption of CI by ML projects is between 24.46% per the triangulation-based approach, and 37.22%, per the FS-based approach, which trail behind those of our comparison set of Non-ML projects, estimated at 38.84% per the triangulation-based approach, and 45.12% per the FS-based approach. The adoption rates of ML projects are also less than those of Open source projects in general, which is estimated at 40.27% by Hilton et al. [148] and more recently between 45% and 68% by Digital Ocean [73], which are consistent with our findings regarding CI adoption by our Non-ML comparison set.

3.5.2 CI Task Analysis

> ***Research Question 2:*** What tasks does CI perform for ML projects?

We applied `TraVanalyzer` on the subset of projects from our **Depth Corpus** with the

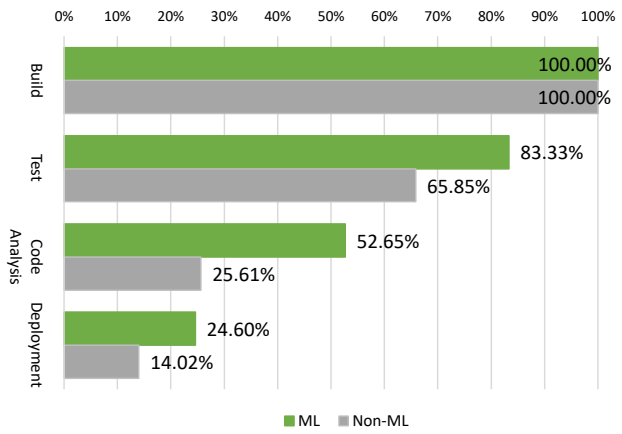


Figure 3.5: CI Task Adoption Percentage

current adoption of Travis: 378 ML projects and 164 Non-ML projects in order to categorize the CI tasks of ML projects and compare them with those of Non-ML projects. We classified the tasks into four categories: build, test, code analysis, and deployment. The build category includes both build environment preparation and build configuration and execution commands. All the ML projects we analyzed include configuration and commands related to build environment preparation and execution in their Travis configuration files. However, only 83.33% of all of the ML projects we analyzed adopted testing in their CI process, and surprisingly, only 65.85% of Non-ML projects adopted testing, even though recommended testing practices are well established for these projects [252]. For code analysis, which includes static analysis, linting, and code coverage tools, overall adoption is lower than build and test adoption. Another surprising result is that while 52.65% of ML projects adopted code analysis in their CI workflow, only 25.61% of Non-ML projects adopted it. Concerning deployment, its overall adoption rate is only 24.6% by ML projects, even though automatic deployment is considered a key component of their workflows [226]. Meanwhile, only 14.02% of Non-ML projects adopted automatic deployment. In addition, as will be illustrated within the rest of this section, along with the higher adoption of the different functions of CI by ML projects, there is a higher diversity of tools being used by these projects to achieve those goals in comparison to Non-ML projects. Comparing these results to those found by Durieux et al. [77] when analysing Travis jobs of open source software in-general, building and testing are also the most common concerns of CI within OSS.

Build Tasks: Travis CI allows the definition of build environment in its configuration. For example, `language:python` prepares an environment for Python-based projects. Other properties can be set, such as the Python version, OS, etc. With `TraVanalyzer`, we extracted build configuration features. The 10 most frequently-used Tavis CI configurations for ML

projects are: `language`, `python`, `dist`, `sudo`, `env`, `cache`, `include`, `os`, and `apt`. These are also the most popular keywords for Non-ML build configuration. Developers can also prepare their build environments by using bash commands to download dependencies, set environment variables, etc. Based on our approach detailed in Section 3.3.2.2, the 10 most frequently used external commands for preparing ML projects' build steps are: `pip`, `export`, `wget`, `apt-get`, `setup.py`, `conda`, `hash`, `cd`, `make`, and `curl`. The top 10 commands used for Non-ML projects are similar, but instead of `hash` and `curl` we find the `git` and `npm` commands.

Test Tasks: Testing confirms the correctness of the code before its integration. Since Travis CI does not provide built-in test configuration, test cases are generally executed by external tools and scripts invoked from the Travis CI configuration file. Based on our analysis, `pytest`, `activate`, external shell scripts, external Python scripts, `nosetests`, `py.test`, `coverage`, `tox`, `unittest` and, `green` are the 10 most frequently used tools and scripts used for ML projects' testing. Most projects invoke python unit test frameworks directly, but some projects invoke external scripts to execute their tests. No usage of ML-specific testing frameworks was noted. The same list was found for Non-ML projects, substituting `green` with the `make_test` command.

Code Analysis Tasks: Some ML projects use code analyzers for code quality and code style checking. These analyzers are generally configured through the Travis CI configuration file to ensure continuous code quality checking. `coveralls`, `codecov`, `flake8`, `coverage`, `pylint`, `bandit`, `mypy`, `autopep8`, `python-codacy-coverage`, and `black` are the 10 most frequently used code analysis tools and commands. Most of these are code coverage analyzers, but some are used for code style checking and other types of static analysis, such as `bandit`, a tool for checking security vulnerabilities of Python code. The first 5 tools were also among the most frequent Non-ML tools for code analysis, however,`ninja`,`pep8`,`pyflakes`,`codeclimate-test-reporter`,`luacheck` were the other 5 most-frequently-used coverage tools.

Deployment Tasks: Most CI services including Travis CI provide built-in support for deployment automation. Developers can also invoke external tools such as Docker [74] to automate deployment. Based on our analysis, `provider:pypi`, `docker`, `provider:pages` [59], `provider:script`, external shell script,`provider:releases` [59], `twine`,
`provider: pages:git`, `doctr`, and `provider:aws` make up the top 10 providers and tools used for deployment. Meanwhile, the only deployment tool we found was beind used by Non-ML projects was `docker`

3.5.3 CI Problem Frequency and Taxonomy

Research Question 3: How often and Why do CI builds break in ML projects?

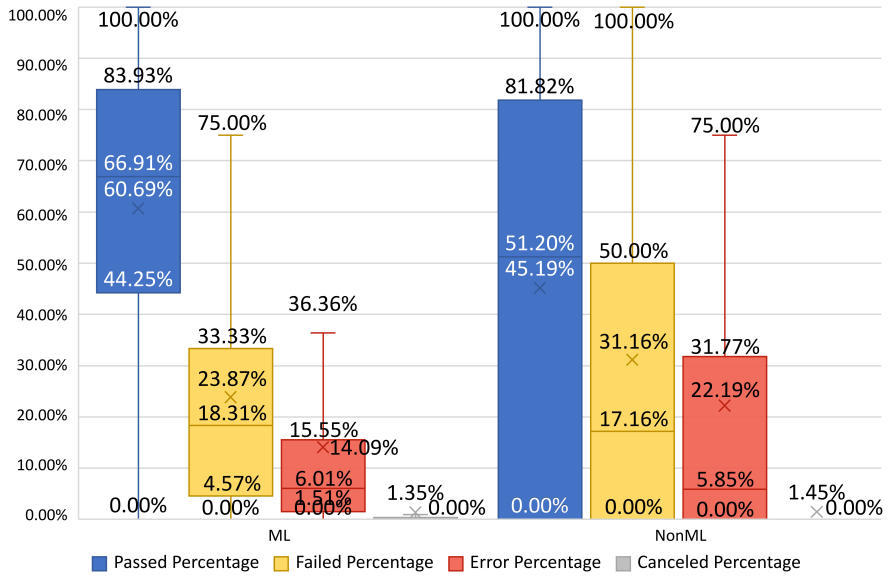


Figure 3.6: Build status average percentages

Figure 3.6 shows the average percentages of the different types of build-outcomes per project within our **Depth Corpus** of projects. While ML projects seem to have a higher average rate of passing builds in comparison to Non-ML projects, it's important to also note the higher internal variability in terms of build-outcomes shown by the Non-ML projects within our **Depth Corpus**, when comparing their quartiles to those of ML projects. The results for Non-ML comparison set are surprising, especially when comparing our findings to Beller et al.'s [33] findings regarding open-source software, where an average of 82.4% of the builds for Java projects and an average of 72.7% of the builds for Ruby projects are passed, but it's important to note that our comparison set is composed of Python-based projects. Delving deeper into the different factors behind build breakage, we chose to analyze the logs corresponding to the failed jobs and errored jobs which compose the non-duplicate broken builds of the projects within our Depth Corpus, that we identified using the process detailed in Section 3.3.2.3. Gallaba et al. [99] showed that logs from Travis CI are an imperfect source of information since they can be incomplete, malformed, or not present within the Travis CI server. Indeed, we encountered problems obtaining the failure logs corresponding to the failed jobs we specified. For ML projects, this totaled 45590 jobs, 13.87% out of which were either empty or not found on Travis CI's server. From the 35965 obtained job failure logs, we achieved a 91.59% saturation rate of detecting at least one job failure type. For Non-ML projects, we succeeded at obtaining 7153 logs pertaining to the job failures

we identified, on which we obtained a saturation rate of 91.15%, and only 11 logs were unobtainable. Moving on to error logs, We also attempted to download 15748 of the errored jobs selected from the jobs of ML projects. 10.33% of them were empty or not found on the Travis server. We achieved a 95.05% saturation rate of detecting the error type of the 14121 job error logs we obtained. For Non-ML projects, we attempted to download 4737 error logs only 1 of which were empty or not found, and we achieved a saturation rate of 96.16% on classifying the error types.

Description of Error Taxonomy: Moving on to the results we obtained, specifically the error taxonomy, the categories of the job errors which we determined are:

Script Error: in 306 jobs of ML projects and 0 jobs of Non-ML projects, constituting respectively 2.17% and 0%. They contain one or more errors within the shell script being executed. For example, an error occurs during copying or deleting a specific item or trying to execute a command that's not available.

Dependency Install problem: in 9989 jobs of ML projects and 3369 jobs of Non-ML projects, constituting respectively 70.74% and 73.96%. They contain problems directly related to the installation of dependencies. For example, if the package manager does not find a package, or there's a problem cloning a git repository required by the project.

Travis CI Error: in 2728 jobs of ML projects and 1157 jobs of Non-ML projects, constituting respectively 19.32% and 25.40%. They contain errors specific to the CI environment. In the case of Travis CI, logs exceeding the maximum size are an example of such errors.

Install phase misuse: in 2732 jobs of ML projects and 864 jobs of Non-ML projects, constituting respectively 19.35% and 18.97%. They contain the usage of the install phases for purposes other than installing dependencies. For example, running Testing, Code analysis, Deployment, or other processes within this phase.

Interpretation of Error Taxonomy results: It is evident that Dependency Install problems are the most common type of problems within the install phase, since they are detected within 70.74% and 73.96% of ML and Non-ML job errors. In fact, all the job error categories we identified, except for the install phase misuse category, are linked directly or indirectly to issues that can occur during dependency installation. This is unsurprising as the main goal of the `install` phase of a Travis job is to install the dependencies needed in order to run the configured scripts for the `script` phase correctly. It's important to note that an errored job can be linked to one or more of the issues within the taxonomy, hence a job can belong to one or more error categories. The frequency of job errors confirms that some of the approaches that developers use to install their dependencies, such as cloning from git or installing a specific version from package managers, are not 100% reliable. A git repository may change location or be removed, and a specific version of a tool may be

removed from the package manager. Similar to Durieux et al.'s work [78] on traditional software, we found git cloning for dependency installation is a problem-causing practice, with 1065 jobs linked to ML projects failing due to a git-related error. While CI Tool errors may be due to limitations with Travis itself, the Install phase misuse being present in 19.35% of ML errored jobs and 18.97% of Non-ML errored jobs is concerning, as it possibly indicates a disregard of developers for Travis conventions, which can make diagnosing and resolving subsequent issues harder. These findings are similar to those of Gallaba et al.[100] concerning the prevalence of misuse of Travis files in open-source software.

Overall, the different job error categories occur with similar percentages in our ML set of projects as well as our Non-ML comparison set. It's important to note however that script errors as well as install phase misuses are present within a larger percentage of ML job errors than Non-ML ones, but Non-ML job errors are more likely to be related to Dependency Install problems and Travis CI related errors than ML job errors. Our findings align with those of Pinto et al. [248] concerning CI build breakages for traditional OSS. Indeed, they also found that issues related to dependency management were common reasons behind build errors. Furthermore, other works such as those of Sulír & Porubän [305], Tufano et al. [315] and Seo at al. [291] who respectively estimated that dependency-related issues accounted for 39% , 58% , and 65% of OSS build breakages.

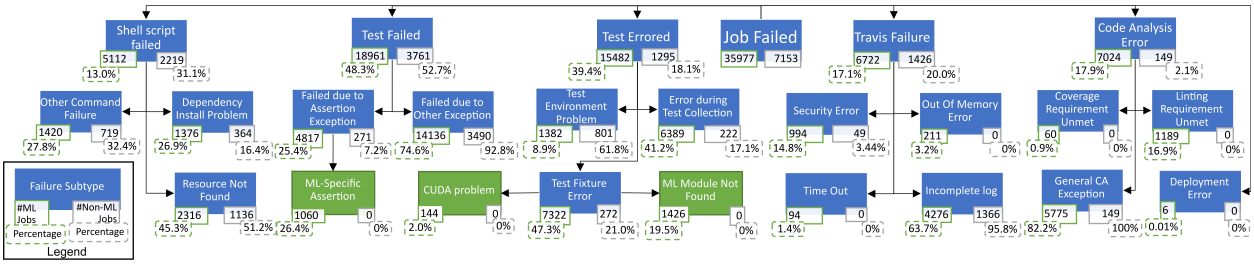


Figure 3.7: Job Fail Taxonomy. We show the count of Failed ML and Non-ML jobs of each sub-type in each block, along with the relative percentage of failed jobs of each sub-type in relation to its direct super-type

Description of Failure Taxonomy: Following the `install` phase, the `script` phase which involves multiple CI processes is executed. A failed job may belong to 1 or more of the main categories or sub-categories. The main types of failures are:

Script Failure: This failure is the result of an error during the execution of the shell script or the python script as it attempts to execute tasks related to preparing the environment for CI processes and execute each one of them. This failure's sub-types are:

⇨*Dependency Install Problems*: this type of failure is encountered when Travis encounters a problem while trying to acquire and install a certain dependency. This is a sign of misuse of

the script phase, as the best practices recommend that all dependencies be installed during the install phase of a Travis job [100].

⇨*Resource Not Found*: If the script tries to access a module, file, or program during its execution (outside of the other CI processes) and is unable to find it, this type of error occurs.

⇨*Other Commands Failure*: A general command failure resulting in the failure of the execution of a command, which in turn may stop the execution of the entire script or the execution of a specific CI task.

Test Failure: One or more tests ran correctly, but identified problems within the functional code being tested. The functional code needs to be modified to resolve this problem. Its sub-types are:

⇨*Assertion Exception*: An exception occurs when an assertion fails within a test execution. This exception occurs when the code fails to establish the functional requirements specified by the developer and is indicative of good test-writing practices being followed. A sub-type of this exception is the **ML-specific Assertion**, which indicates that an exception specific to the context of ML projects has occurred, such as the following example where the model prediction result does not meet the accuracy threshold.

```
@Example@: FAIL: test_recalculate_user (tests.als_test.ALSTest)
AssertionError: 1.0 != 0.0 within 0.0001 delta
```

⇨*Other Exception*: This type of exception occurs when an exception unforeseen by the test occurs. It indicates that the test did not account for this specific type of failure, and thus is not following the best practices of test-writing [252].

Test Error: One or more tests did not run correctly, due to problems within the test build up or tear down, or other processes related to preparing the environment to execute a test. Its sub-types are:

⇨*Error During Test Collection*: This indicates an error occurred while the testing framework was attempting to collect the tests across the different test-script files. This is usually due to an error in the tests which prevents them from being loaded into Pytest, such as missing modules or indentation errors within the test files.

⇨*Test Fixture Error*: An error that occurs during the execution of a pre-test or a post-test method, also known as fixtures, due to a programmatic problem during the execution of these methods such as a resource not being found. An ML-specific sub-type of this exception is

the **CUDA Problems**, which indicates an exception related to the CUDA ML framework, and the other one is **ML Module Not found**, which specifies that the test fixture was unable to import a module generally associated with ML projects such as Tensorflow.

@Example@:
Could not load dynamic library 'libcuda.so.1'; dlerror: libcuda.so.1: cannot open shared object file: No such file or directory

⇨*Test Environment Problem*: This indicates a problem linked to the testing environment was found by the testing framework. For example, it occurs when certain environment variables or dependencies that were expected by the testing framework were not found, or the testing framework was unable to find or create the testing environment.

@Example@:
$tox
ERROR: unknown environment 'vulture'

Code Analysis Error: A failure during the process of Code Analysis or caused by its result. For example, if an unexpected exception occurs during the code coverage phase, or if severe code formatting issues are reported, this type of error occurs. Its sub-types are:

⇨*Code Coverage*: This type of failure occurs when the testing coverage does not meet the minimum criteria set by the developer.

@Example@:
FAIL Required test coverage of 100% not reached

⇨*Linting*: This type of failure occurs when the coding practices of the software do not conform with the conventions set by the developer.

⇨*Other Code Analysis Fail*: this occurs when an exception is thrown during a code analysis process. In general, it's due to an unexpected exception during the process of Code Coverage or Linting.

Deployment Error: This error occurs when a problem is encountered when the CI process attempts to deploy artifacts to a certain destination, for example, due to connection or authentication issues. The reason behind the low number of deployment-related job failures is the fact that the failure of the deployment phase built into Travis does not affect the build outcome status. Hence even if a deployment fails, if the job has not encountered any problems beforehand, it will still be considered passed.

Travis Failure: This error occurs due to problems related to the CI tool being used within a project. For instance, in the context of Travis, this could be a security error related to certificates within the Travis container, the container running out of memory, etc. Its sub-types are:

⇨*Security Error*: This error occurs when the Travis instance faces a problem related to the verification of the signatures of certain resources, such as a package manager.

⇨*Out of Memory Error*: This error occurs when the Travis CI instance runs out of memory and can no longer load needed resources into its memory.

⇨*Incomplete Log*: This type of error occurs when the Travis Log is unexpectedly incomplete (for example, stopping in the middle of an output line). This is due to a known issue with Travis CI [108].

Interpretation of Failure Taxonomy results: While a job may fail for multiple reasons, it's clear that most of the job failures of ML projects are linked to testing. Focusing on test failures, 4817 jobs of ML projects, 25.4% of the reported failures, were a result of an assertion exception, thus 74.6% of the reported test failures did not follow the best practices of testing for the Python language [252]. Furthermore, only 26.4% of the test failures of ML projects which followed these practices were ML-specific, such as failing to meet accuracy criteria, indicating that the majority of the tests performed were not necessarily ML-specific. Regarding test errors, the second most common failure reason, 88.5% of them were due to programmatic problems related to test collection or test fixtures, which are problems linked to the coding of the tests themselves and to their frameworks' configuration. Comparing these results with those of Non-ML projects, it's clear that ML projects have different frequencies of job failure types. One surprising results is that ML projects are better at following testing practices than Non-ML projects, since only 7.2% of the latter's failures followed the recommended practices, especially considering their relative novelty [129]. Focusing on build failures in OSS in general, Pinto et al. [248] found similar build failure reasons as we did, ranging from inadequate testing to missing edge cases, when analyzing build breakages within OSS via interviews. Vassallo's work [322] illustrates a similar picture for both open source and Industrial Java software in the context of build failures. Beller et al.'s work [33] illustrated that most build failures are due to failed tests, where 70.89% of Java build failures and 67.13% of Ruby build failures were due to failed Tests. In comparison, 48.3% ML build failures were due to Test failures. The similarities between OSS job failures, our comparison set of Non-ML job failures, and ML build failures confirm that the same problems that affect CI in OSS also affect CI in ML projects, with some variance in frequency. For example, static analysis failures were present in 17.9% of ML projects, but only affected 4.2% of builds of OSS in Vasallo's results [322], and 2.1% of the job failures of our Non-ML set.

None of the test failures or test errors we detected through our semi-automatic log analysis or the manual methods we used in constructing it revealed the usage of ML-specific testing frameworks or tools, even though a few of these tools have been introduced, such as that by

Karlaš et al. [170], or Amazon [292], or the usage of recommended practices [170], such as changing the test set with different builds to avoid the overfitting of models. This indicates that even though ML-specific tools and practices were introduced to the software engineering process, their adoption is still lagging.

3.6 Implications

For Researchers. First, while CI in ML projects has not received much attention from the research community, it's adopted by up to 37.22% of ML projects, showcasing its importance as a subject of study. We provide researchers with a set of CI-using ML projects, in order to guide their work in developing and adjusting CI servers and tools for ML projects. Second, Travis CI is the best source of information regarding CI practices of ML projects, and based on our study, there is no evidence of the widespread adoption of CI platforms specifically designed for ML projects. Thus, `TraVanalyzer` is a great tool to further investigate CI practices in ML projects and can be easily extended to support other types of Travis-using projects, especially since we were able to successfully use it in our set of Non-ML projects. Third, we found some of the most frequent CI problems of ML projects were related to test failures, test errors, and code analysis failures. Our Failure taxonomy and corresponding log analyzer can help with the automatic detection and debugging of these problems and guide the development of Travis CI configuration repair tools. **For ML Developers**. While adopting a CI tool is a step in the right direction, simply building the software with it is insufficient. Testing is a basic tenant of Continuous integration [96], and code analysis procedures, especially code coverage, are highly recommended. ML projects' developers should invest in implementing these processes in their CI workflows and adapting them to the context of ML projects and that of their project. For example, during our manual log-analysis step for the construction of our CI Log analyzer, we found that some projects are executing their tests on the same test set in every build, increasing their risk of ML model over-fitting. To avoid this problem, ML projects are recommended to vary their test sets frequently [170]. Another example is that some ML projects had job failures due to a restrictive 100% code coverage requirement, which can be harder to achieve as the code-base grows. Relaxing this requirement or monitoring CI status frequently can help minimize unnecessary disruptions.

3.7 Related Work

Hilton et al.'s work [148] contains one of the most exhaustive CI adoption estimates for open-source projects on GitHub, as it considered a larger amount of open-source CI tools and projects than many other works examining this issue [132, 33, 317]. When it comes to CI goals, Durieux et al.'s [77] work is a good indicator of the tasks CI in OSS performs, It classified the tasks being performed by Travis CI into categories ranging from testing to communication. Moving on to CI problems, Gallaba & McIntosh [100] did an excellent job at analyzing a set of .travis.yml files, extracting configuration anti-patterns from them Focusing more on Travis in action, Beller et al.'s [33] work analyzes Travis CI jobs with problems resultant from testing and presents important information about their frequency per language. Focusing also on the testing aspect of CI, Karlaš et al. [170] outline a few problems concerning CI tool-support for ML projects and discuss some of the problems that using the CI tools' traditional testing practices may engender in regards to an ML model's accuracy.

What distinguishes our work from these is scope: it is one of the few that specifically focus on CI within ML projects in practice, as well as analyzing CI in Non-ML in Python-based projects as a baseline for contextualization. Furthermore, to estimate CI adoption within ML projects, we apply a triangulation-based approach on more recent data, rather than the API-only approach that Hilton et al. [148] employ, which may give false positives in case a project is no longer using a CI service. Concerning the goals of CI, Durieux et al's work's [77] granularity with its focus on jobs may be skewing the results in the direction of the CI tasks being performed by more active projects that generate more builds, and thus more jobs, or projects which are configured to run multiple jobs in different environments, thus artificially increasing the count of the tasks they are trying to perform. Our work attempts to investigate the multitude of CI problems ML projects experience in practice, ranging from dependency installation problems to deployment errors.

3.8 Threats to Validity

The major threat to this study's internal validity is the correctness of the classification of the ML projects dataset. To reduce this threat, we have manually inspected the breadth corpus to ensure that the studied projects are actual software projects, not study guides or toy projects, and that they are in fact using ML. Moreover, the Travis CI AST-based task analyzer and CI log analyzer developed for automatic analysis can have internal threats to the correct analysis of the tasks, failure types, and error types. To minimize such threats,

we also evaluated the correctness of these tools with manually labeled data as described in Section 3.4. The major threat to the study's external validity is that we analyzed open-source ML projects available on GitHub and mainly focused on Python-based projects. So, the CI adoption by ML projects findings can be different for closed-source projects and projects developed in other programming languages. However, our findings are still significant, since our dataset included large-scale ML projects such as `tensorflow/tensor2tensor`, which was developed and open-sourced by organizations like Google Brain. We also focused on Python, the most popular programming language of ML projects.

3.9 Conclusion

In this work, we have shown that CI has been less widely adopted among ML projects in comparison to Non-ML projects. We also analyzed their different CI tasks, and extracted knowledge about common problems of CI in ML projects. To the best of our knowledge, this is the first work that has analyzed ML projects' CI usage, practices, and issues. Furthermore, we have also contextualized these results by comparing them with similar Non-ML projects, and summarized useful findings for researchers and ML developers to identify possible issues and improvement scopes for CI.

CHAPTER 4

Empirical Analysis on CI/CD Pipeline Evolution in Machine Learning Projects

This work is currently under submission in the Journal of Empirical Software Engineering (EMSE).

4.1 Introduction

Continuous Integration (CI) [32] establishes an automated way to build, test,and package software applications and encourages developers to commit code changes more frequently [80, 150, 34]. Continuous Delivery (CD) is an extension of CI, which additionaly automates the delivery of applications to selected environments in short cycles [157]. CI/CD pipelines help create an automated and consistent process that helps reduce human errors, increase productivity in teams, and accelerate release cycles [80, 150, 319]. CI/CD has become the industry standard of modern software development [295] and has been widely adopted in Open-Source Software (OSS) projects [150, 303] and in Machine Learning (ML) projects [277] which have gained widespread popularity and significance in recent years [127, 256].
Machine Learning shares a lot of common ground with traditional software development, especially with their the need for multiple development iterations to enhance their quality. However, ML projects introduce a unique set of challenges due to their inherent complexity [14, 139]. For instance, regular testing methods generally used in CI can cause model overfitting, rendering accuracy measures unreliable for evaluating models [168]. Additionally, ML projects encounter challenges in version control and dependency management, leading to manual interventions during model experiments and deployments to address these issues [195]. Furthermore, limited knowledge exists on software maintenance and evolution in the context of ML systems [202]. A recent study by Zampetti et al. [351] explored how CI/CD configurations evolve over time in open-source projects and focused mostly on the restructuring actions occurring within these files. Other studies [126, 176, 335] on CI/CD

focused on the goals and challenges of utilizing CI/CD in traditional software systems. To the best of our knowledge, understanding the nuances of this co-evolution in ML projects between source code and CI/CD configuration file changes remains uncharted territory. This research gap makes it challenging to discern the necessary adjustments in CI/CD configurations to effectively accommodate the changes in ML systems.

Building upon this context, this paper presents an in-depth analysis of the evolution of CI/CD configurations in ML projects. By examining the intricate relationships between changes in ML source files and corresponding adjustments in CI/CD configurations, analyzing the patterns of change, and evaluating developers' expertise, this study aims to unravel the complexities of sustaining and maintaining CI/CD setups for evolving ML models. We collected 578 open-source Python-based ML-enabled projects on GitHub having Travis CI or GitHub Actions as their CI/CD infrastructure. We opted for these filtering criteria since Python is the most popular language for ML-enabled projects, and Travis and GitHub Actions are their most popular CI tools [277]. We extracted 38,982 commits from these projects modifying the different CI configuration files for Travis CI and GitHub Actions. We filtered those commits to include only ones modifying both ML source files and CI/CD configurations and then we applied random sampling to obtain 701 commits which will be used for manual analysis. Through this work, we answer the following research questions:

- **RQ1**: *How do CI/CD pipelines evolve in ML projects?* We observe that changes to the Build Policy in ML CI/CD configurations occur in a sizeable chunk of the commits, mostly due to updates in the the installation policy. We found that Performance ajd Maintainability are not major concerns when it comes to updating CI/CD configurations, which may lead to technical debt and prolonged builds.
- **RQ2**: *How do CI/CD pipelines co-evolve with ML code?* We devise a taxonomy of 14 co-evolution categories and we find Testing and Dependency Management as the most prominent categories of change whereas Deployment and Data Versioning are infrequent.
- **RQ3**: *What are the common patterns of change occurring in CI/CD configurations?* We generated a comprehensive list of change patterns occurring in CI/CD pipelines in ML projects. The AST analysis supports our earlier findings and shows that there are minor adjustments related to deployment and failure handling. This is worrisome because it means that ML developers often resort to manual interventions for debugging build failure and for deploying models.

  We identified four bad practices performed by ML developers: adding dependencies directly to the CI Configurations file for Travis CI and GHA, not using the automatic discovery feature of testing frameworks in both CI tools,using deprecated Travis CI settings, and relying on a generic build language configuration in Travis CI.

- **RQ4**: *How skilled are the developers changing CI/CD configurations in ML projects?* Our analysis revealed a robust and statistically significant positive association between developers' project knowledge and expertise, and their involvement in modifying CI/CD configurations. This indicated that the more active and knowledgeable developers are more inclined to modify CI/CD configurations.

In summary, our study makes the following contributions:

- The first quantitative and qualitative study of CI/CD configuration evolution in open-source ML projects.
- A taxonomy of 14 categories of co-changes between CI/CD configurations and ML source code.
- A list of common change patterns in CI/CD configurations in ML projects that can be used to mitigate challenges associated with ML CI evolution.

Furthermore, our code scripts and empirical dataset are publicly available for future research [17].

The remainder of this paper is organized as follows: Section 4.2 discusses the background of the project, and Section 4.3 outlines the research methodology employed in our empirical study. Section 4.4 provides a detailed analysis addressing the four research questions. Potential threats to the validity of our study are mentioned in Section 4.5. The research implications are discussed in Section 4.7. Finally, Section 4.6 delves into related works to our study, while Section 4.8 concludes this study.

4.2 Background

CI/CD pipelines have been used by different types of projects and have been adopted by a fair amount of ML projects [277, 278]. In recent years, some CI/CD tools designed explicitly for ML projects have emerged, including KubeFlow [178], Amazon Sagemaker [12], and Azure ML [213]. However, these ML-specific tools can not be used alone and typically complement traditional CI/CD tools for managing the codebase of ML-enabled projects. Despite the differences between traditional software development and ML projects, some CI/CD tools designed for the former, like Travis CI [61], amd GitHub Actions [120] remain the most popular in both domains.

Travis CI [61] is one of the largest and most popular CI/CD services [351]. It supports a variety of programming languages and provides a cloud-based infrastructure, relieving developers of the burden of maintaining their own environments. GitHub Actions [120] is another popular CI/CD service that automates workflows for GitHub repositories. It is integrated with GitHub, allowing developers to define CI/CD workflows directly in the

repository. GitHub Actions also supports a wide range of programming languages. What differentiates GitHub Actions from Travis CI provides a variety of pre-built actions that can be used to automate common tasks. Both GitHub Actions and Travis CI can automatically detect changes to the repository on version control systems like GitHub and triggers a build based on predefined events. The build process is configured by the *.travis.yml* file, which resides in the root of the project repository. GitHub Actions, on the other hand, uses a similar configuration files under the folder *.github/workflows* to define the workflow.

The *.travis.yml* file defines the build process through a series of sequential stages. Each stage is composed of multiple sequential phases, each phase contains multiple jobs running in parallel and executing different in a virtual environment. The build environment can be configured through the `os` keyword, which sets the Operating System of a job's container, and the `language` keyword, which installs the tools and dependencies of a specific programming language. This configuration can be used multiple times and with different values to configure various environments for jobs, each executing the same sub-script in its designated environment.

The job executes a series of phases which are shown in Figure 4.1. The typical phases include an `install` phase for dependency installation, a `script` phase for running tests, an `after_success` phase to handle post-test actions such as coverage reporting, and a `deploy` phase for project deployment.

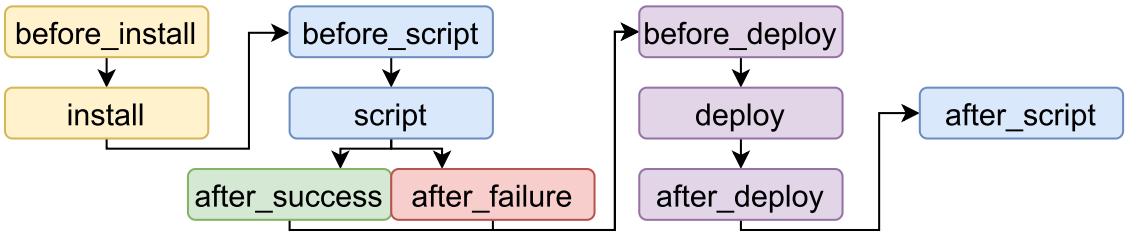


Figure 4.1: Travis CI job Lifecycle.

GitHub Actions relies on multiple *ymls* files to define the workflow. Each file contains a series of jobs that run in parallel or sequentially. Each job can have multiple sequential steps, which are the individual tasks that run in the job's environment. These steps can rely on commands or actions, where the latter are reusable units of code that can be shared across workflows. Actions can be used to automate common tasks like building, testing, and deploying code.

GitHub workflow can be triggered by various events, such as push, pull request, or issue creation. The workflow can also be scheduled to run at specific times or triggered by external events like a new commit or pull request. In comparison to Travis CI, GitHub Actions

workflow rely on a more modular and flexible structure, allowing developers to define complex workflows with custom life cycles. By default, jobs run in parallel. However, they can be configured to run sequentially via the `needs` keyword, which allows a job to depend on the completion of another job before it can start. An example of a custom lifecycle, where the build and test jobs run in parallel, is shown in Figure 4.2.

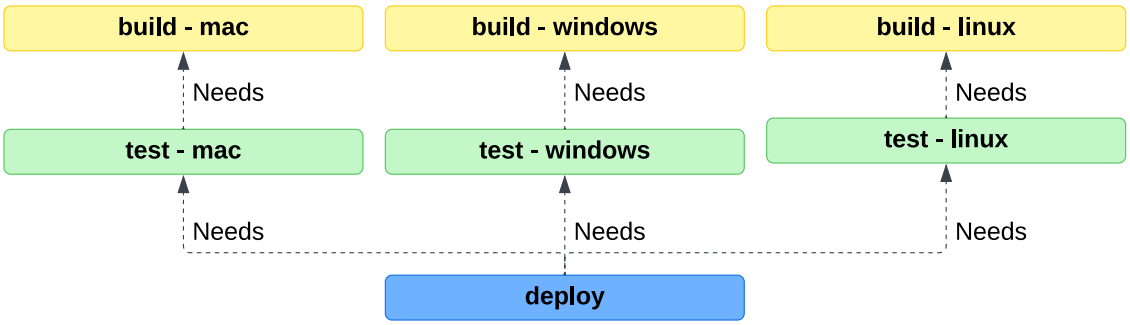


Figure 4.2: An example of a custom GitHub Actions job Lifecycle.

4.3 Research Methodology

Figure 4.3 shows an overview of the research methodology. In this section, we begin by describing the dataset used in this study and how we acquired it. Then we will move on to explaining the approach by specifying the steps needed to answer each of the four research questions.

4.3.1 Data Collection

In order to conduct our qualitative and quantitative analysis, we utilizied the ML Project dataset proposed by Rzig et al. [278]. The dataset was curated to eliminate toy or study projects and ensured projects were indeed using ML. The dataset contained 4031 ML projects hosted on GitHub, one of the most popular platforms for hosting software repositories [165]. Our set is composed of 578 open-source ML projects extracted from this set based on the following criteria. They have Python as the main programming language since it was found to be the most popular language for ML projects [127], and use Travis CI or GitHub Actions, which are the two most widely adopted CI/CD infrastructures in OSS projects [351] and in ML projects [277]. From these projects, we extracted 15,634 that modify Travis CI configuration files, which we refer to as TravisCI- -modifying commits, and 23,348 that modify GitHub Actions configuration files, which we refer to as GitHub Actions-modifying commits. These commits will be used for the quantitative study to answer RQ3 and RQ4.

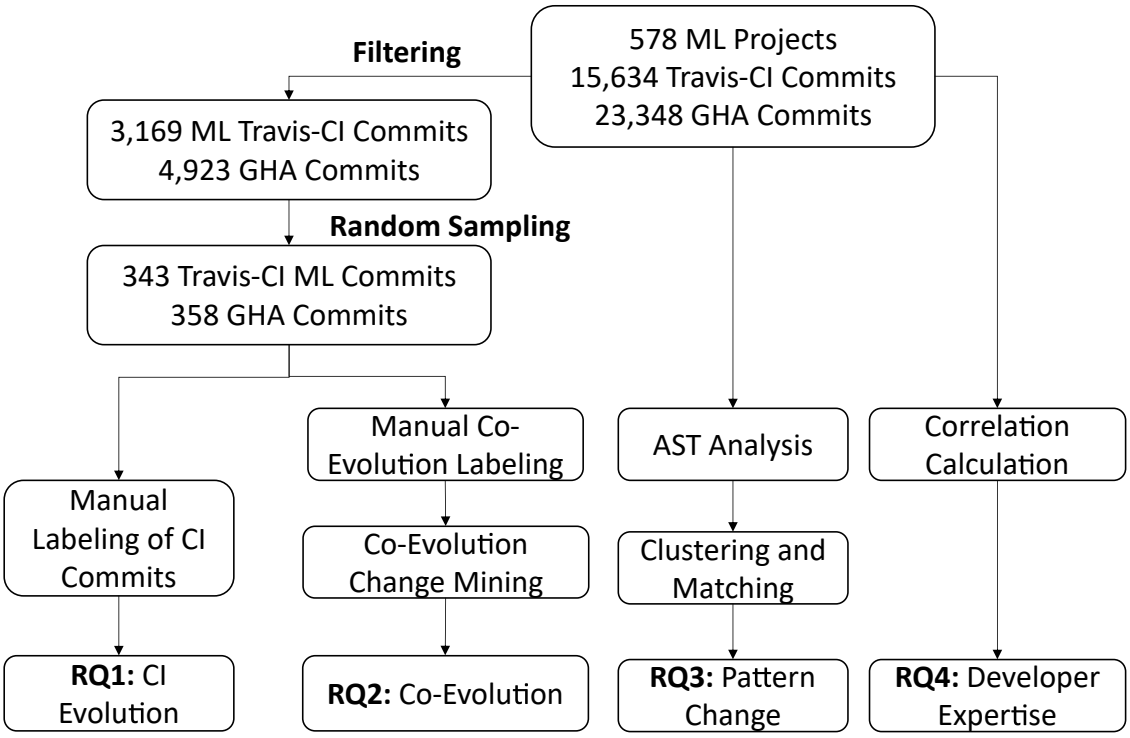


Figure 4.3: Overview of Research Approach

Then, we filtered those commits based on two criteria: At least one Python file should be in the list of modified files and at least one of the modified source code files should have one of these keywords related to ML in their name or path: 'data', 'model', 'train', 'training', 'test', 'pipeline', 'predict', 'correctness', 'deploy', 'inference', and 'preprocess'. Prior works in Machine Learning [40, 79] have also utilized keyword-based searching. We applied this filter in order to obtain commits that are impacting both the pipeline and ML source code, which represents one of the main of focuses of this study. We ended up with 3,169 Travis CI-modifying commits, and 4,923 GitHub Actions-modifying commits. We applied pure random sampling with a 95% confidence level and a 5% margin of error on each sample. We obtained 343 commits for Travis CI and 358 Commits for GitHub Actions for manual analysis in RQ1 and RQ2.

4.3.2 Approach

4.3.2.1 CI Evolution Analysis

To understand the evolution of CI/CD pipelines, two authors manually labeled the 343 Travis CI commits and the 357 GitHub Actions commits using Zampetti et al.'s [351] taxonomy of restructuring actions applied in the pipeline configuration file. Both of these authors have

extensive expertise in DevOps and Software engineering. They focused on the second level of the taxonomy to simplify our analysis. Inter-rater agreement was measured using Cohen's kappa [205] at 0.72 for labeling Travis CI commits, and 0.71 for labeling GitHub Actions commits indicating substantial agreement. A reconciliation meeting was held afterward to resolve the differences.

4.3.2.2 Co-Evolution Analysis.

Studying the changes in the CI/CD configuration files can give us an idea of the most frequent actions performed on it. However, the over-arching intent of these changes is not obvious when the CI files are analyzed individually. To yield deeper insights, we analyze code changes in both ML source files and the CI configuration files of the tool, to better understand how CI co-evolves with other ML-related components.

Manual co-evolution labeling. Two authors individually labeled the changes happening in the sample commits in both ML source and CI files. Because there could be multiple changes occurring in the same commit, the labeling was not limited to one category per commit; rather, the raters were free to add as many categories as required to assess all the changes. Not only did the authors analyze the code changes happening in files, but they also observed the commit message and description. If any are found, Pull Request (PRs) discussions of the commit were also analyzed. The labels were created by following a cooperative card-sorting procedure [302, 214]. The list of categories defined by both raters was maintained in a shared file, ensuring consistent naming without introducing substantial bias. Then, the two raters discussed and resolved conflicts in a consensual agreement meeting. We calculated the Cohen's kappa [205] score, which we found to be 0.65 for Travis CI commits, and 0.66 for GHA commits, showing substantial agreement. In the end, we identified 14 categories of co-changes in CI/CD pipelines and ML code.

Co-evolution Change Mining. To further analyze the co-evolution between CI/CD configuration changes and ML-related changes in RQ2, we employed Association Rule Mining (ARM), which describes relationships between different phenomena. The rules consist of frequent subsets and take the form of X => Y, where X represents the antecedent and Y is the consequent. Metrics like "Support" and "Confidence" are used to measure the quality of these rules [10]. Support(X => Y) indicates the frequency of appearance of co-occurrence of X and Y, while Confidence(X=>Y) measures the conditional probability that Y is present given the presence of X, indicating the reliability of the association between X and Y.

$$Supp(X \Rightarrow Y) = P(X \cap Y) \tag{4.1}$$

$$Conf(X \Rightarrow Y) = \frac{P(X \cap Y)}{P(X)} \tag{4.2}$$

To generate Association Rules, we picked the Apriori algorithm [5], a widely used algorithm for studying co-change and co-evolution [327, 146, 200]. Given a set of transactions, the Apriori algorithm generates association rules satisfying user-specified minimum support and confidence criteria. It starts by generating large itemsets appearing in a minimum proportion of transactions using a minimum support threshold and then uses these itemsets to derive association rules. In RQ2, we use ARM to assess the relationship between the categories we labeled manually for co-evolution analysis. After a process of parameter tuning, similar to other co-evolution works [200, 323], we used a minimum support value of 0.1 and a minimum confidence of 0.6 for the Travis CI commits. Following the same process, we used a minimum support of 0.01 and a minimum confidence 0.2 for the GitHub Actions commits. These values were chosen to focus on relatively frequent and potentially more significant associations.

4.3.2.3 Change Patterns Analysis

To gain a better understanding of the the changes happening in CI/CD configuration files, we performed a change pattern analysis. We used Abstract Syntax Trees (AST), to represent the code to help mine the re-occurring changes. Then we cluster and match these ASTs to generate general and valuable insight.

Abstract Syntax Tree Analysis. We were interested in studying the patterns of changes happening in the CI/CD configurations in the context of ML projects. But first, we needed to parse the different CI configuration files and extract the Abstract Syntax Trees (AST) [225], that they contain. ASTs break down code into a tree-like structure, with nodes representing different language constructs, such as functions, loops, or variables, and edges denoting the relationships between them. To accomplish this, we used TraVanalyzer [277], a tool designed to parse Travis CI configuration files and generate the corresponding ASTs. Furthermore, we expanded this tool to support GitHub Actions configuration files. With it, we parse the CI/CD configuration file in all the CI-modifying commits and then apply AST diffing using the state-of-the-art GumTree [86] to compute fine-grained changes. GumTree can detect change types happening in the configuration files by comparing each node in the AST between two diffs, allowing us to pinpoint and categorize the structural changes that take place within the configuration files.

Command Matching and Clustering of CI changes. After generating the change patterns from all the commits, we now needed to find a way to group them into meaningful

clusters to analyze their different properties and stages. However, there was an abundant use of commands and scripts within the different CI configuration files. Following Rzig et al.'s [277] approach, we extracted the commands appearing in the file from each commit and applied a matching process where we matched AST nodes having the same command name and omitted the rest of the parameters in the commands since they are often project-specific and do not add value. We performed a more refined clustering based on parent node name similarity, which consists of a CI keyword [62, 118], and change types to generate the final list of change patterns. Finally, we apply a normalization procedure on the node labels to ensure the generalizability of our results. We identified 59,948 changes in the Travis CI files and 103,480 changes in the GitHub Actions files from the dataset of CI-modifying commits. Each change includes the action performed, the modified command, and the parent Travis CI keywords where the command was performed.

4.3.2.4 Developer Expertise Analysis

Finding reliable metrics for measuring developer expertise in software development is no easy feat, as previously reported [267, 30]. Previous studies have employed diverse approaches, including skill vectors [72, 25], ML techniques [20, 218], and, notably, Change History information [204, 217, 105]. The latter encompasses metrics such as commit frequency and the extent of modified lines of code, and has been proven effective by Anvik et al. [21]. In our study, we aim to investigate the role of developer expertise in the context of modifying CI/CD configurations for ML projects. We hypothesize that developers with a deeper knowledge and prolonged involvement in the project are more inclined to modify CI/CD configurations. In our dataset comprising 15,634 Travis CI commits, we identified 1951 developers as contributors modifying Travis CI pipeline configurations, with each developer uniquely identified by their email addresses. Concerning the 23,348 GitHub Actions commits, we identified 1808 developers as contributors modifying GitHub Actions pipeline configurations.

To assess expertise, we calculate the percentage of CI-modifying commits for each developer alongside the percentage of their total contributions to the projects. The objective is to explore how experience, manifested through active and substantial contributions to a project, influences the likelihood of developers being involved in CI/CD changes. To quantify and statistically assess the strength of the observed relationships, we calculate the correlation between these two measures using Spearman's rank-order correlation [301] and Kendall's correlation [289]. Spearman's correlation measures the strength and direction of the monotonic relationship between two variables. Kendall's correlation quantifies the strength and direction of the ordinal association between two variables by comparing the number of concordant and discordant pairs of data points. We used the p-value measure to assess the

statistical significance and strength of the observed correlations.

4.4 Empirical Evaluation

4.4.1 RQ1: Evolution of CI/CD pipelines

We begin by evaluating the percentage of CI/CD pipeline changing commits in the 578 ML projects. We find a median of 4.44% when examining projects that used Travis CI, and 4.04% when looking at projects that relied on GitHub Actions, revealing similar percentages for both CI tools.

To further understand how CI/CD pipelines evolve over time, two authors manually labeled the commits by adopting Zampetti et al.'s [351] taxonomy as described previously in Section 4.3.2.1. We give a brief description of each category here, but more details are provided in Zampetti et al's [351] study.

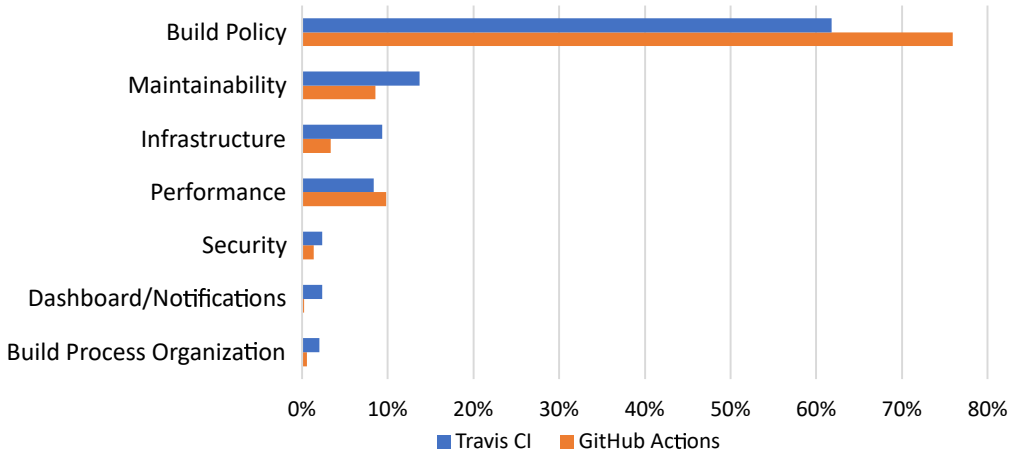


Figure 4.4: Distribution of CI/CD change categories.

Overall, comparing the different categories between Travis CI and GHA with a paired T-test [270], we find a p-value of 0.99, indicating a non-statistically significant difference between the two tools, thus implying that both tools show similar change category trends overall.

Build Policy: Actions in this category concern the build triggering strategy and dependencies' installation policy. We found many changes in this cateogry to be related to managing the dependency installation policy. This category represents a common concern for both Travis CI and GitHub Actions modifications, where it affected 61.8% and 75.91% of commits, respectively. It's clear that altering the build policy to adapt to ML-related changes is

a common practice, unlike the set-it-and-forget-it approach often promised as an advantage of CI/CD [80].

Maintainability: Maintainability refers to the ease with which the CI/CD pipeline can be modified, repaired, and understood. Common actions include improving the readability of code snippets, renaming jobs and scripts, and simplifying the build matrix. Our analysis reveals that maintainability is a less common concern, with only 13.7% and 8.68% of Travis CI and GitHub Actions modifying commits, respectively. This is a worrisome sign as ML systems have a higher susceptibility to incurring technical debt, as they not only inherit the typical maintenance challenges associated with traditional code but also face an additional set of ML-specific issues [286].

Infrastructure: This category revolves around changes to the overall infrastructure supporting the build process. The identified actions involve adopting Containerization to ensure a consistent environment for the build process, and the migration of the whole CI/CD pipeline from GitHub Actions to Travis or vice-versa. The two authors improved on this definition from Zampetti et al.'s [351] by adding the latter action to this category. The raters found a limited number of commit changes related to Infrastructure, 9.3% and 3.36% for Travis CI and GitHub Actions commits, respectively.

ML projects often involve a diverse set of dependencies and specialized hardware configurations, making it challenging to encapsulate all aspects of the ML environment within Docker containers effectively for both CI tools. The larger share of infrastructure commits for Travis is explained by the large number of migrations observed from it to GitHub Actions, consistent with findings in other works [125].

Performance The objective in this category is to minimize build time by removing unnecessary components from the workflow, adopting caching strategies, and introducing parallelization. These actions align with established practices, including recommendations from Duvall [80]. ML models usually involve extensive computations, large datasets, and numerous dependencies, thus requiring special attention to handling these complexities when configuring CI/CD builds. We find that 8.45% and 9.80% of Travis and GitHub Actions commits, respectively, are related to performance improvements.

This finding suggests that, despite the resource-intensive nature of ML development, optimizing CI/CD runtime performance has not been a main focus for ML developers.

Security This category includes actions with security implications. Examples include securing credentials/tokens in pipeline configuration and either introducing or removing `sudo` in commands. We found a low percentage of respectively 2.33% and 1.43% of Travis CI and GitHub Actions commits related to security compared to other categories.

Dashboard/notifications Improving the notification mechanism and enhancing build log

readability in CI/CD pipelines are the main goals for this category. Here, we found a percentage of 2.33% and 0.28% for Travis CI and GitHub Actions, respectively. Notifications are provided by default in GitHub Actions, and are also configurable via the GUI, thus explaining the much lower percentage of changes in this category for GitHub Actions compared to Travis CI. It is notable that not properly configuring build output for CI/CD is considered a bad practice [352]. Thus, ML developers, especially those using Travis CI, need to improve on this.

Build Process Organization: This category focuses on improving the overall organization of the CI/CD configurations through reordering the execution of workflow steps, restructuring of the different phases, and embracing parameterized builds for both tools. Here, we also find lower percentages of changes compared to other categories, at 2.04% and 0.56% for Travis CI and GitHub Actions commits, respectively.

RQ1 Findings: *a sizeable chunk of changes happening in CI/CD configurations are related to updating the build policy with minor changes to Performance, Maintainability, and Build Process Organization which may lead to technical debt and prolonged builds. Some Minor differences were found between Travis CI and GitHub Actions regarding these categories.*

4.4.2 RQ2: Co-evolution of CI/CD Pipelines and ML Code

We curated a taxonomy of 14 categories to describe ML CI/CD co-changes, as explained in Section 4.3.2.2. The distribution of the different categories amongst the 343 Travis CI and 357 GitHub Actions commits is shown in Figure 4.5.

We employed Association Rule Mining to assess the coupling between these different categories, which is detailed in Section 4.3.2.2. We found seven association rules concerning Travis Commits, which we present in Table 4.1, and 9 association rules concerning GitHub Actions commits in Table 4.2. Now we investigate the results of our analysis for each of the categories in the taxonomy.

Overall, comparing the different co-evolution categories between Travis CI and GHA with a paired T-test [270], we find a p-value of 0.98, indicating a non-statistically significant difference between the two tools, thus implying that both tools show similar co-change category trends overall, similar to the results found in Section 4.4.1.

Testing: Testing includes commits related to writing or updating tests for ML models. This category appeared in 21% of Travis CI-modifying commits and 12.52% of GitHub Actions-modifying commits. This high frequency supports our findings in RQ1 where we identified

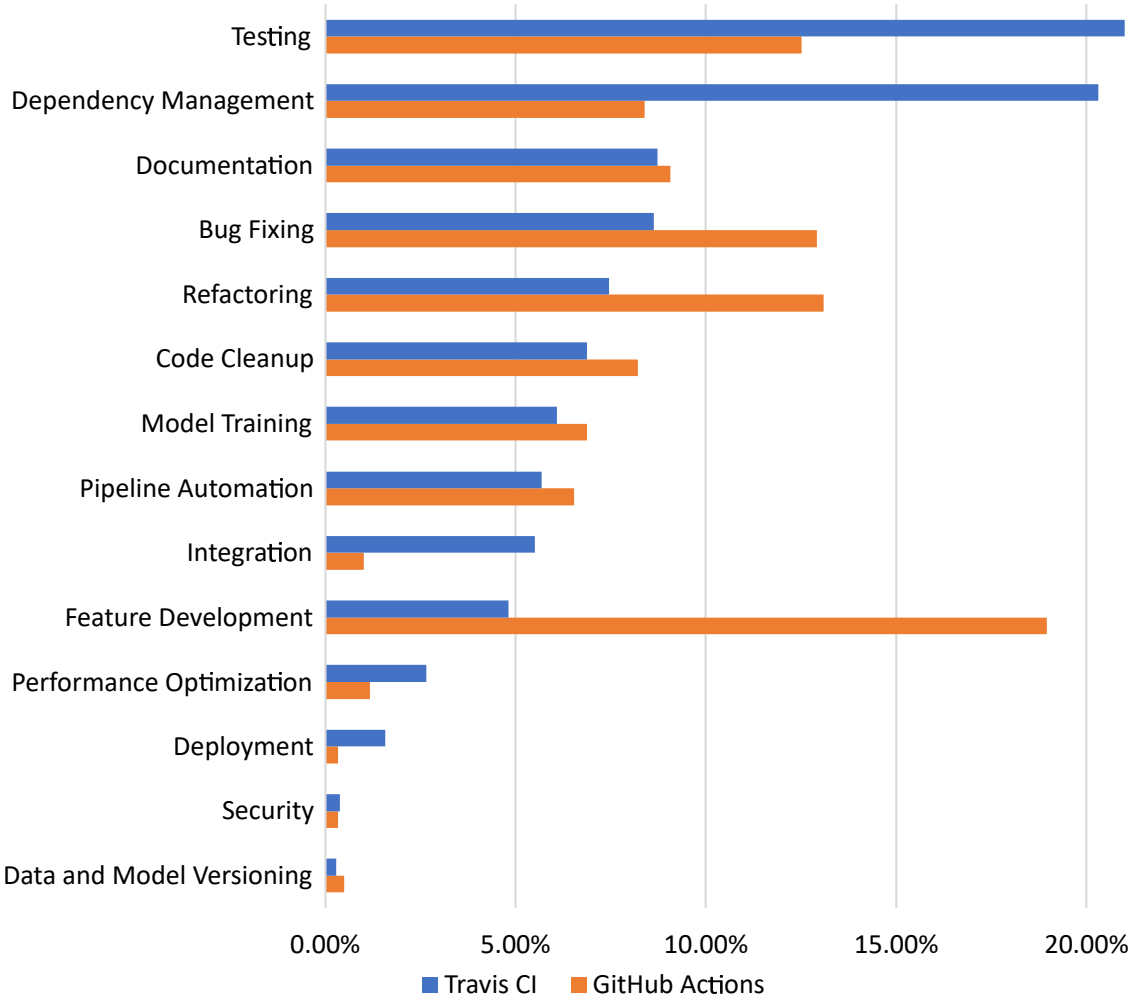


Figure 4.5: Distribution of commit categories

Table 4.1: Association rules mined for Travis CI commit analysis

Rule Antecedents	Rule Consequents	Support	Confidence
Documentation	Dependency Management	0.18	0.71
Pipeline Automation	Dependency Management	0.11	0.67
Refactoring	Dependency Management	0.13	0.61
Feature Development	Testing	0.1	0.75
Integration	Testing	0.11	0.71
Model Training	Testing	0.11	0.66
Pipeline Automation	Testing	0.12	0.72

Table 4.2: Association rules mined for GitHub Actions commit analysis

Rule Antecedents	Rule Consequents	Support	Confidence
Documentation	Feature Development	0.04	0.29
Code Cleanup	Feature Development	0.03	0.28
Pipeline Automation	Feature Development	0.03	0.28
Pipeline Automation	Testing	0.02	0.25
Pipeline Automation, Feature Development	Code Cleanup	0.01	0.36
Pipeline Automation, Code Cleanup	Feature Development	0.01	0.50
Feature Development, Code Cleanup	Pipeline Automation	0.01	0.28
Pipeline Automation, Feature Development	Documentation	0.01	0.36
Pipeline Automation, Documentation	Feature Development	0.01	0.44
Feature Development, Documentation	Pipeline Automation	0.01	0.25

Build Policy, which includes updating testing policy, as the most important change category of both CI systems. This is due to the fact that in many cases, we found that developers add the tests manually to the CI/CD configuration. An example is illustrated in Listing 4.1, where a new test was added to the *.travis.yml* file for the ray-project/ray repository, which has over 28,000 stars.

Listing 4.1: Adding a new test suit in *.travis.yml* (ray-project/ray/d06beac).

```
1 script:
2  ...
3 + - python test/trial_scheduler_test.py
```

However, this is a bad practice because each time a new test is added, the developers need would also need to update their CI/CD configuration to include it. A better approach is using a testing framework with automatic discovery, such as `pytest` [253] and `nosetests` [233], which scan for testing files and functions, making it easier to maintain and scale test suites

as the project grows.

Dependency Management (20.31%): Commits in this category involve adding or updating dependencies used in ML or CI components. 20.31% of Travis CI-modifying commits are related to dependency management, while 8.39% of GitHub Actions commits are related to this category. Many Travis projects manage their dependencies within the Travis CI configuration file instead of externalizing them into dedicated files, such as *requirements.txt* and *pyproject.toml*. This leads to frequent changes to the installation policy in *.travis.yml* as we discussed in RQ1. To a lesser extent, many GitHub Actions projects also manage dependencies within the CI configuration file, but the percentage is lower than Travis projects. We show an example of this in Listing 4.2 for the piskvorky/gensim repository, a well-known open-source Python library with over 14,000 stars, where dependencies are added directly in the *.travis.yml* file. Another example is shown in Listing 4.3 for the OpenMined/PySyft repository, a popular open-source library for privacy-preserving machine learning with over 9,500 stars.

Listing 4.2: Adding new dependencies directly in *.travis.yml* (piskvorky/gensim/7e74d15).

```
1  install :
2    ...
3  + - pip install tensorflow
4  + - pip install keras
5
```

Listing 4.3: Configuring dependencies directly in GitHub Actions configuration file (OpenMined/PySyft/60c0dae).

```
1    install :
2   run: |
3        pip install --upgrade tox tox-uv==1.5.1
4
5
```

We consider this a bad practice, as embedding dependencies in CI scripts can make it harder to maintain a consistent and reproducible environment.

Documentation (8.73%): This category involves updating project documentation, including *README* files, code comments, or API documentation. The raters found 8.73% of Travis commits and 9.06% updated documentation. Concerning Travis CI commits, as shown in Table 4.1 we find that if there is a documentation update within a commit, it is likely that there is also a dependency management change as well with a confidence of 0.71. This change, as observed by the two authors, is to ensure that the documentation reflects the correct dependencies and their versions. For instance, in the EducationalTestingService/skll

repository, they changed the `tabulate` package to `prettytable` and had to update both *.travis.yml* and the *README* file. Concerning GitHub Actions Commits, we found a different association in Table 4.2, where if there is a documentation update, it is likely that there is a feature development change as well with a confidence of 0.29. This is because when adding new features, developers need to update the documentation to document these features and ensure they can be used properly. An example of this is shown in Listing 4.4, where a new feature is added to the Espnet/Espnet repository, which relies on GitHub Actions, and the documentation is updated to note this new feature. Another example of this change is shown in Listing 4.5 for the EducationalTestingService/skll repository, which uses Travis CI. This category is also strong correlated with Pipeline Automation and Feature Development, as shown in the last two rules in Table 4.2.

Listing 4.4: Changing the 'prettytable' dependency to 'tabulate' and updating documentation (EducationalTestingService/skll/cd5bf73).

```
# .travis.yml
...
-     prettytable python-coveralls ruamel.yaml
+     tabulate python-coveralls ruamel.yaml
# README.rst
...
-   `PrettyTable <https://pypi.org/project/PrettyTable/>`
+   `tabulate <https://pypi.org/project/tabulate/>`
```

Listing 4.5: Adding documentation of new features to documentation (espnet/espnet/8eed522).

```
  # README.md
  ...
  +   - Transfer Learning :
  +  - easy usage and transfers from models previously trained by your group, or
     models from [ESPnet huggingface repository]
  ...
```

Bug Fixing

Commits in this category address and resolve issues or bugs identified in the code. 8.64% of Travis CI commits and c 12.92% of GitHub Actions Commits belonged to this category. These lowers percentages might indicate that CI configuration file and ML source code changes are not often related to bug fixing. Here, we observe that the bug fixes in ML files and CI/CD configurations are independent and are usually related to fixing syntax errors in

the CI/CD configuration files.

Refactoring Refactoring focuses on improving the code's structure, readability, and maintainability without altering its external behavior. ML developers might remove files, reorganize code, rename variables, or simplify complex functions to enhance the overall quality of the codebase. The category appears only in 7.46% of Travis and 13.09% of GitHub Actions commits. For Travis CI, we find that it leads to dependency updates with a confidence of 0.61. This is mostly due to removing deprecated libraries and updating Python versions, such as the example shown in Listing 4.6 which was taken from the marl/openl3 repository. In this commit, developers removed Python versions 2.7 and 3.5 and added versions 3.7 and 3.8 since as part of their refactoring process, they changed their models to use Tensorflow 2 and it only supports Python versions 3.6-3.8.

Listing 4.6: Removing python versions unsupported by Tensorflow 2 (marl/openl3/d593e2d).

```
python:
+#- "2.7" # byeeeee forever
+#- "3.5" # tensorflow 2 does not support
  - "3.6"
+ - "3.7"
+ - "3.8"
```

Code Cleanup: Commits in this category are related to removing dead code, unused variables, or deprecated functions to help maintain a clean and efficient codebase.

We find that 6.87% of the Travis CI commits and 8.22% of the GitHub Actions commits are related to code cleanup.

We show a code snippet reflecting this category from the chainer/chainercv repository, where they removed the disk attribute, a decorator that marks tests consuming a lot of disk space, from the *.travis.yml* file as well as source files as shown in Listing 4.7.

Listing 4.7: Removing the disk test decorator from test files and *.travis.yml* (chainer/chainercv/3eff205).

```
# \textit{.travis.yml}
- MPLBACKEND="agg" nosetests -a '!gpu,!slow,!disk' tests;
+ MPLBACKEND="agg" nosetests -a '!gpu,!slow' tests;
# tests/datasets_tests/ade20k_tests/test_ade20k.py
-    @attr.disk
     def test_ade20k_dataset(self):
```

Model Training: Model Training commits are related to training or fine-tuning ML models. Changes to hyperparameters, datasets, or training algorithms fall under this category. Model

training is essential for optimizing the performance of ML applications. However, with a percentage of 6.08% of Travis Commits and 6.88% of GitHub Actions, we realize that changing model behavior is not always correlated with updating CI/CD configurations. We also found that model training often requires adding new tests and/or updating older ones with a confidence of 0.66. An example of that is shown in Listing 4.8 taken from the OpenNMT/OpenNMT-py. Here, the developers decided to use training steps instead of epochs when training the models. Training steps provide more fine-grained control over the training process. A change in the testing command was performed in *.travis.yml* as well.

Listing 4.8: Using training steps instead of epochs for model training (OpenNMT/OpenNMT-py/4d17982).

```
1   - python train.py -model_type img -data /tmp/im2text/q -rnn_size 2 -batch_size 10
        -word_vec_size 5
2 -      -report_every 5 -rnn_size 10 -epochs 1
3 +      -report_every 5 -rnn_size 10 -train_steps 10
```

Pipeline Automation (5.69%): These commits introduce or enhance automation scripts or workflows within the CI/CD pipeline. Automated processes improve efficiency and reliability, enabling seamless integration, testing, and deployment of machine learning models. The percentage of 5.69% of Travis CI commits and 6.54% of GitHub Actions commits suggests minimal efforts for using automation scripts within these commits. Furthermore, We noted that adding automation scripts often coincides with updates in testing components and changing dependencies for Travis Commits as shown in Table 4.1, and with Testing, Feature Development, Documentation, and Code Cleanup for GitHub Actions commits as shown in Table 4.2 with a confidence of 0.72, 0.28, and 0.50 respectively.

.

Integration: Integration commits signify connections with other systems or services, aligning with a broader software ecosystem. Integration can include databases, web applications, or notification systems. We found a percentage of 5.5% of Travis CI, and 1.01% of GitHub Actions commits which is moderately low compared to other categories. We also found that these changes are frequently followed by updates in testing components in Travis CI with a confidence of 0.71 in Table 4.1. This indicates a strong correlation between integration efforts and ensuring comprehensive testing, possibly to validate the integrated systems. The raters observed that in most cases for Travis CI, the service is actually Docker and is used to run integration tests.

Feature Development (4.81%): Feature Development involves the addition of new ML features to the codebase. This could include implementing novel algorithms, data processing

techniques, or any other functionality that enhance the capabilities of the machine learning models. This was observed in 4.81% of Travis CI commits and 18.96% of GitHub Actions commits.

The lower frequency in Travis CI suggests that new features do not require updates in the Travis CI configuration file, where as GitHub Actions configuration files needs to be updated more frequently to support these features. In Travis CI, these changes are highly associated with Testing updates with 0.75 confidence as shown in Table 4.1, and is highly correlated to Documentation, Code Cleanup, and Pipeline Automation with a confidence of 0.29, 0.28, 0.28. .

Performance Optimization: Commits aiming to optimize the performance of machine learning models or CI/CD processes fall into this category. The techniques used are detailed under Section 4.4.1. We found that 2.65% of Travis CI commits and 1.17% of GitHub Actions commits are related to performance optimization.

Listing 4.9: Using the joblib [164] library for parallel computing (tslearn-team/tslearn/d3062d3).

```
1 # \textit{.travis.yml}
2   - conda create -q -n test-environment python=$TRAVIS_PYTHON_VERSION Cython numpy
      >=1.14 scipy tensorflow keras
3 -       scikit-learn numba nose
4 +       scikit-learn numba joblib>=0.12
5 # tslearn/clustering.py
6 def silhouette_score(X, labels, metric=None, sample_size=
7 -  None, metric_params=None, random_state=None,
8 +  None, metric_params=None, n_jobs=None, random_state=
9 +  None,**kwds):
```

A notable case involves the use of parallelization through `joblib` [164], a well-known Python library that provides tools for parallel computing and efficient caching. Listing 4.9 is a code snippet taken from the tslearn-team/tslearn repository where the developer added the joblib package to perform parallel computations within their models. They added the dependency to the installation command in the CI/CD files since it is now needed to build and test the models.

Deployment: Deployment commits involve activities related to deploying machine learning models to production environment updating deployment-related code. Surprisingly, we found a small number of commits that apply deployment changes: only 1.57% of Travis CI commits 1.57%, and only 0.34% of GitHub Actions commits. ML practitioners seem to prefer more manual and controlled deployment processes, reflecting a risk-averse attitude

toward automated deployment. The limited emphasis on CD could stem from the unique challenges posed by ML models, emphasizing the need for precision and careful consideration in better-tailored deployment processes for ML projects.

Security: This category addresses security vulnerabilities or improves the security aspects of machine learning models, data handling, or CI/CD processes. With only found 0.39% of Travis CI commits and 0.34% GitHub Actions commits related to security, it is obvious that security is not a major concern when updating both ML source code and pipeline configurations. This aligns with our finding from earlier in Section 4.4.1, where we also identified a small number of commits related to security in CI
CD configuration files.

Model And Data Versioning: These commits are related to data and model versioning and management, ensuring consistency and reproducibility in ML experiments. This category has the some of the lowest frequencies with only 0.29% for Travis CI and 0.50% for GitHub Actions. Although versioning is established as a good practice, ML versioning is still a young practice as observed by Lewis et al. [182], due to the lack of effective tools tailored for the complexity of ML models. Traditional code versioning tools like Github are unsuitable for data versioning due to large sizes and specialized tools like Data Versioning Control (DVC) are still not widely adopted in ML projects, as found by Barrak et al. [31]. Most of the commits we found simply used Github to specify datasets with different versions whilst only one commit uploaded the model to Amazon Web Services (AWS) [293] storage with their corresponding versions as a workaround for the large file sizes. The commit in Listing 4.10, from sorgerlab/indra, updates the version of the folder containing the models in the cloud storage and renames to reflect the version update from 1.2 to 1.3.

Listing 4.10: Versioning data in AWS storage (sorgerlab/indra/6bba5a2).

```
1 -  - mkdir -p $HOME/.indra/bio_ontology/1.2
2 +  - mkdir -p $HOME/.indra/bio_ontology/1.3
3 -  - aws s3 cp s3://bigmech/travis/mock_ontology.pkl
4 -    $HOME/.indra/bio_ontology/1.2/
5 +  - aws s3 cp s3://bigmech/travis/bio_ontology/1.3/
6 +    mock_ontology.pkl $HOME/.indra/bio_ontology/1.3/
7      bio_ontology.pkl --no-sign-request
```

RQ2 Findings: *We devised a taxonomy of 14 co-changes and identified Feature Development, Testing Dependency Management, Bug Fixing, and Refactoring as the most prominent categories. We found two bad practices in those two categories which are direct inclusion of dependencies in CI/CD configuration and a lack of usage of auto-discovery testing frameworks.*

4.4.3 RQ3: Change Patterns in CI/CD pipelines

To gain a deeper understanding of the evolution of CI/CD configurations, we wanted to identify the change patterns occurring of the build environment settings and the different job phases. To achieve this, we perform AST analysis on the different CI/CD configuration files and then apply clustering and matching to obtain a list of change patterns, as explained in Section 4.3.2.3. Due to the differences in structure and lifecycle between Travis CI and GitHub Actions configuration files, we conducted the analysis separately for each CI system and present them in two separate sections. We first discuss the results for Travis CI, followed by the results for GitHub Actions. The results for Travis CI are illustrated in Figure 4.6, and the results for GitHub Actions are illustrated in Figure 4.7.

4.4.3.1 Travis CI Change Patterns

Build Environment Configurations: As shown in Figure 4.6, we identified the top eight build environment configurations and the frequency of their changes. For some of them, we also specified the top three change patterns. Setting environment variables via the `env` key is the most changed configuration with 5,183 occurrences. By adjusting environment variables, developers can easily specify and control the test environment. Furthermore, it provides flexibility for testing since ML projects often involve diverse frameworks, data sources, or experimental conditions. The next most changed setting is related to updating Python versions. Developers may want to ensure their projects are tested across various Python environments to guarantee broad compatibility.

We also found 960 updates to the `sudo` key which is lower than the other patterns. This aligns with our results in RQ1 and RQ2 where we found that security is not a big concern for ML developers. It is notable that the `sudo` key is actually deprecated according to Travis CI documentation [62]. We consider the usage of this key as a bad practice, as the use deprecated code is a bad practice in software engineering overall. Additionally, we identified 661 changes related to caching, which is also lower than other keys. As explained in RQ1 and RQ2, performance-related changes are minimal in *.travis.yml* file within ML projects.

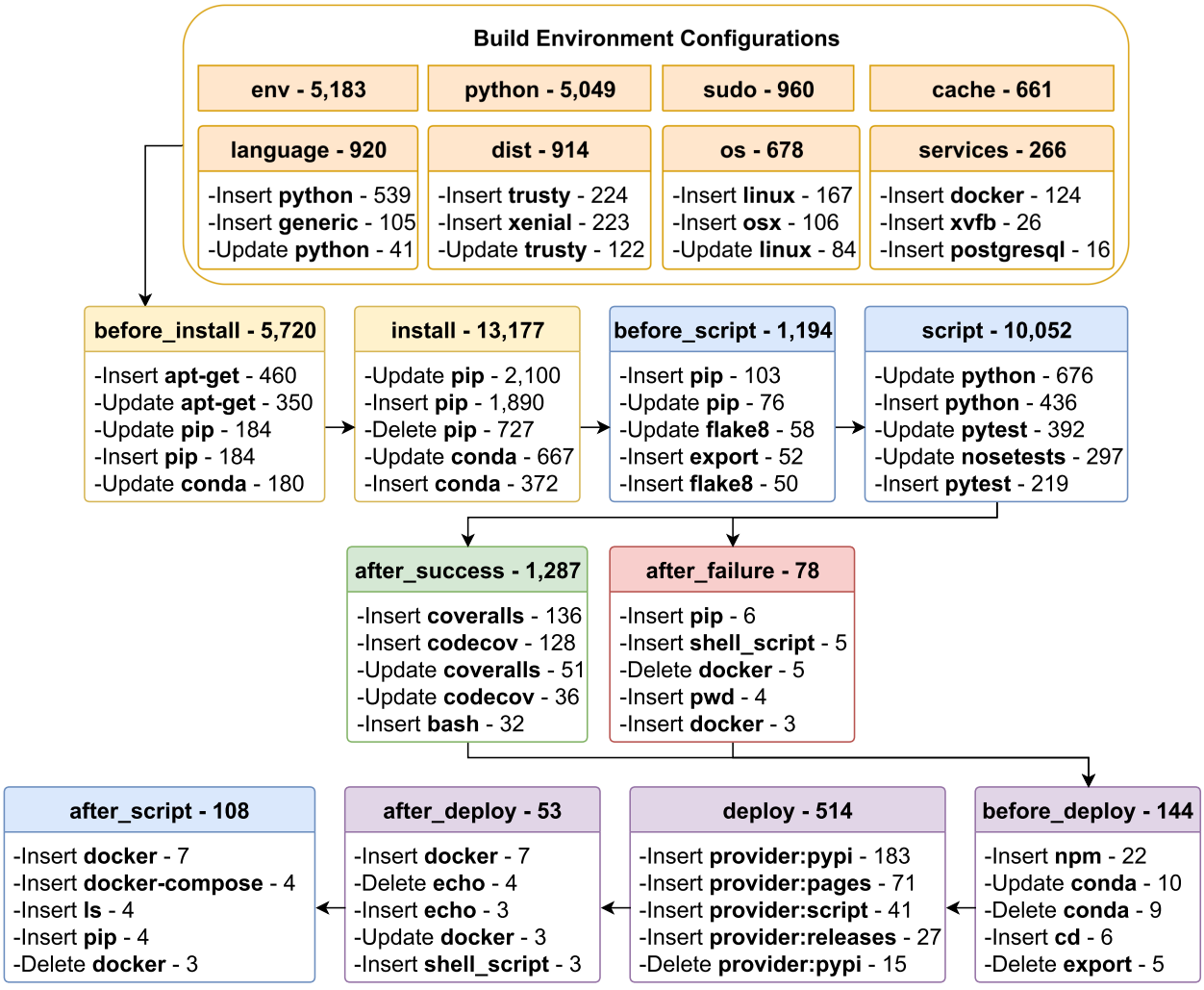


Figure 4.6: Change Patterns in Travis CI configurations lifecycle.

The `language` key has been updated 920 times in our dataset, and most changes are related to adding or updating the Python language or using a `generic` language. The latter means that the build environment should not assume a specific programming language and should provide a generic environment where developers can setup their own build tools. However, since all the projects we examined are written in Python, we consider this behavior as a bad practice. Using "python" as the language key provides a more standardized and optimized environment.

The `os` and `dist` keys were modified moderately with `linux` and `osx` being the most changed operating systems. As for distribution, most changes are related to adding either the `trusty` distribution or the `xenial one`. Both represent a version of the Ubuntu operating system with specific features and package versions.

Finally, we found minimal changes related to integration with services, a detail we also observed in RQ2. Similarly to our findings in RQ2, most changes are related to adding `docker`. However, as observed in RQ1, there are minimal changes related to Containerization due to the difficulty of encapsulating the complex aspects of the ML environment within Docker containers. Other added services include `xvfb`, a service that provides a virtual display server for running graphical applications, and PostgreSQL, a pre-configured environment that provides a running instance of the PostgreSQL database server for testing. The usage of the `xvfb` service in CI/CD configuration within ML projects appears somewhat unconventional due to the non-graphical nature of many ML tasks.

Job Phases: We observe that the `install` phase has undergone the most changes with a frequency of 13,177. The top five change patterns we found are related to `Pip` and `Conda` which are considered the most popular package managers [294]. Additionally, the `before_install` phase is the third most changed phase as well which is often used for pre-setup tasks before the main installation phase. This supports the results we found in RQ1 and RQ2, where we identified updating dependencies as the second most frequent category of change happening in CI/CD changing commits as well as updating installation build policy as the most occurring action in *.travis.yml* file.

The `script` phase is the second most frequently changed. This phase typically includes commands for running tests, as shown in the change patterns in Figure 4.6. The high number of changes indicates a significant amount of activity related to test scripts, which we observed in RQ2. `Pytest` [253] and `nosetests` [233] are popular testing frameworks for Python but surprisingly, the python command is changed more frequently, potentially indicating a tendency to run tests independently using the python command rather than utilizing standard testing frameworks, a bad practice that we also noted in RQ2. As for the `"before_script"`, it is also frequently changed. Surprisingly, we found that the that

are many changes linked to the pip commands, further confirming the prevalence of the bad practice of directly including dependencies in the CI/CD configuration file which we also noted in Section 4.4.1. We also found the flake8 command, often used to perform linting and static code analysis, and the export command which sets environment variables. Updating these commands aligns with the idea of performing necessary setup and checks before the main build or testing phases commence which is the intent of the `before_script` phase. This aligns with existing findings regarding the misuse of the different phases of the Travis CI configuration file [100].

The discrepancy between the frequency of changes in the `after_success` and `after_failure` phases may be attributed to the distinct nature of these phases within the CI/CD pipeline. `after_success` typically includes actions performed when the build and tests have passed successfully, indicating a stable state, whereas `after_failure` is executed in the event of test failures or build errors. Changes in `after_success` are mostly related to adding the `coveralls` and `codecov` commands, which report code coverage metrics to external services after a successful build. The `after_failure` less frequent usage can indicate that ML developers often resort to manual intervention or debugging outside the CI/CD pipeline when errors occur, thereby reducing the need for frequent adjustments.

Furthermore, we note that deployment-related phases, which are `before_deploy`, `deploy` and `after_deploy` have low frequencies compared to other phases which suggests a cautious approach to automated deployment in the ML community. We found similar results in RQ2 as well with Deployment being one of the least frequent categories of changes. The most commonly added provider is PyPI, followed by a few others such as GitHub Pages and releases, as well as custom script-based deployment strategies.

4.4.3.2 GitHub Actions Change Patterns

GitHub Actions does not have predefined phases like Travis CI, instead, every project maintainer defines their specific phases and steps through the use of `jobs` and `steps` keys. Furthermore, these jobs have custom names and are not standardized like Travis CI. Due to this flexibility of GitHub Actions, many more AST forms are possible in GitHub Actions compared to Travis CI. Hence the clustering process using only the predefined syntax has limited feasibility and results in GitHub Actions. Hence, we enrich this process by also examining some commonly used keys under the jobs, namely the common job names "build", "test", and "deploy". We also examine the "matrix" key which is used to define different environments/configurations for multiple jobs.

Build Environment Configurations: The most frequently changed key in GitHub Actions is the `on` key, which defines the triggers behind the GitHub Actions workflows, with 4662

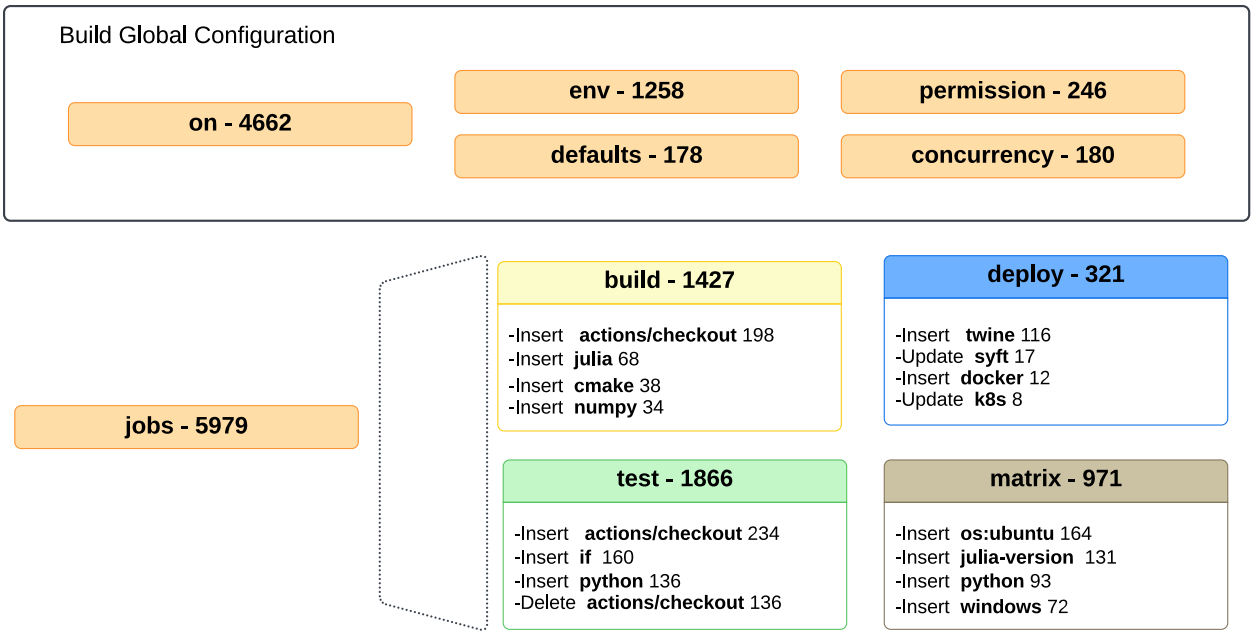


Figure 4.7: Change Patterns in GitHub Actions configurations lifecycle.

changes, denoting that changing a workflow's policy is a common concern behind GitHub Actions commits, confirming our findings in Section 4.4.1 regarding the frequency of Build Policy changes. Similar to Travis CI setting environment variables via the `env` key is also a common concern, with 1258 changes.

However, this is where the similarity between the two tools end. Indeed, GitHub Actions offers a `permissions` key, which can be used to modify the default permission granted to the GitHub token, thus changing what the token can access, whether for example it can access the repository's secrets or modify its contents, or perform other actions. This key was changed 246 times, showing that developers are concerned with the security of their workflows. However, consistent with our observations in Section 4.4.1, that Security-focused changes are of limited frequency.

The key `defaults`, which allows developers to set shared settings across multiple jobs, was changed 178 times, showing that some developers are taking advantage of this GitHub Actions feature to ensure consistency across their workflows. Finally, concurrency is a concern for 160 changes, showing that developers are concerned with changing the default behavior of GitHub Actions workflows to ensure that they run in parallel or serially as needed. However, the low frequency of this change indicates that most developers are content with the default behavior of GitHub Actions workflows, as we observed in Section 4.4.1 concerning the low frequency of Build Process Organization changes.

Jobs:

As jobs describe the different processes that are to be executed as part of the GitHub Actions workflow, they are the most frequently changed key in GitHub Actions, with almost 6000 changes. Under the jobs key, we found on three common job names to get a better understanding of the granular changes being performed to achieve the different processes.

Focusing on `build`, a job name commonly used for that jobs that compile code and prepare any artifacts necessary for the execution of the workflow, we denote that the insertion of the action "actions/checkout" was the most common operation. This action is a necessary addition to ensure a copy of the code from the repository is available to the job. The second most commonly inserted command was `julia` with 68 insertions. Julia is a fast-performing programming language that is often used in ML projects, and programs written in Julia can be called directly from Python. Hence, this change pattern reveals that some developers are taking advantage of Julia's superior performance for specific ML tasks even though they rely on Python for the majority of their codebase. Another common insertion we denote is `cmake`, a cross-platform build system commonly utilized with C Language, with 38 insertions. This change pattern reveals that some developers are using CMake to compile parts of their codebase, which is a common practice in ML projects that rely on C or C++ for performance-critical tasks. This change pattern and the latter change pattern reveal a trend in ML projects to use multiple languages to optimize performance and functionality, resulting in a heterogeneous codebase. Finally `numpy` was inserted 34 times as an explicit dependency, revealing that some developers are manually setting up the numpy library in GitHub Actions, which is a common practice in ML projects that require numerical computations and linear algebra operations, thus further confirming the bad practice of directly including dependencies in the CI/CD configuration file that we noted in Section 4.4.1.

Moving on to another common job name, the `test` jobs also contained numerous insertions of the `actions/checkout` action, at 234. The keyword `if` was inserted 160 times, denoting the common usage of conditional command to run specific tests or test-suites following specific conditions. The command `"Python"` was also inserted 136 times, mostly where a specific test file was denoted as a parameter for this command. This reveals an bad practice encountered in Section 4.4.3.1, where we noticed a lack of usage of auto-discovery-based test tools and their corresponding commands. Finally, the deletion of `actions/checkout` is a common change at 136 deletions. This is likely an optimization, as we noticed this generally occurs in cases where the code is copied in a previously executed job, thus making this action redundant.

We shift focus to the `deploy` key, which is generally used to deploy the codebase to a specific environment, was changed 321 times. It contained 116 insertion of the `twine` [316]

command, which is used to upload Python packages to the Python Package Index (PyPI). This reveals a common deployment strategy in ML projects, where the codebase is deployed to PyPI [95] for easy access and distribution. Another commonly inserted command was `syft` [16], with 17 insertions. Syft is a tool that is used to generate software Bill of Materials, thus revealing the packages and dependencies required for a certain ML project. The insertion of this command reveals that some developers are using Syft to identify the dependencies of their codebase before deployment, ensuring that the deployment environment is correctly configured. Finally, the 12 insertions of `Docker` [74] and the 8 updates of `K8s` [130] command reveal that some developers are deploying their codebase to Docker containers or Kubernetes clusters, which are common deployment strategies in ML projects that require scalability and flexibility.

Finally, we discuss the `matrix` key. Matrix does not represent another common job name, but rather, it is a key generally used to define different environments and configurations of multiple jobs. The Matrix key was updated 971 times. Out of these updates, the insertion of `os:ubuntu` was performed 164 times, reflecting that most developers rely on the Ubuntu to run their workflows, similar to our findings in Section 4.4.3.1, as it is a commonly used linux distribution that has a very active community and comes pre-installed with Python, making it a popular choice for ML projects. The insertion of `julia-version` was performed 131 times, revealing that developers are running certain jobs within an environment that supports Julia, as we observed in the `build` job. The insertion of `python-version` was performed 93 times, revealing that developers are running certain jobs within an environment specifically setup for a specific version of Python. Finally `windows` was inserted 72 times, revealing that some developers are running their workflows on Windows, which is a less common choice for ML projects, as most ML libraries and frameworks are optimized for Linux environments. However, this change pattern reveals that some developers are using Windows to evaluate the compatibility of their codebase with Windows environments, which is a good practice to ensure that the codebase is compatible with a wide range of environments, especially due to the widespread usage of Windows in the end-user market.

RQ3 Findings: *We generated a comprehensive list of change patterns. We reveal how these patterns those supports our findings in RQ1 and RQ2. We specifically uncover two more bad practices: usage of deprecated CI syntax, the reliance on a* `generic` *build language.*

4.4.4 RQ4: Developer Expertise for CI/CD Configuration Changes

Developer expertise has been a well-explored area in the context of recommendation systems, with substantial research highlighting its significance [204, 217, 215]. As explained in Section 4.4.4, we used change history, and specifically the frequency of commits, as a metric to evaluate the experience level needed to conduct changes in CI/CD configurations. We

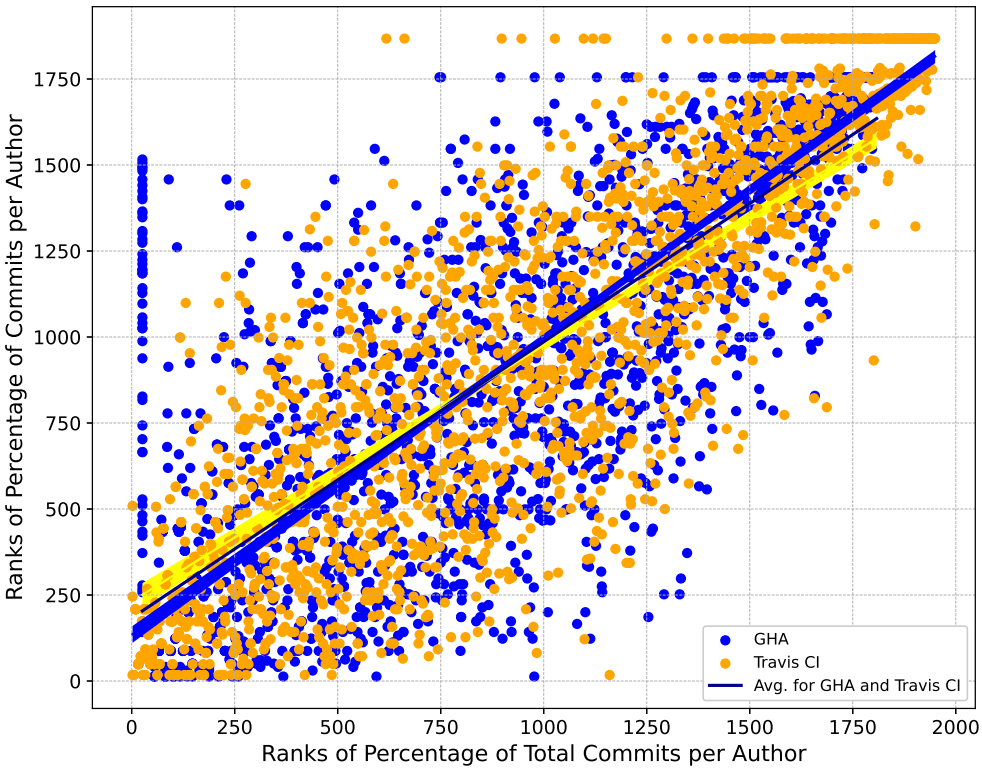


Figure 4.8: Ranks of percentage of commits per author (Spearman's)

calculate the correlation between the percentage of CI-modifying commits for each developer and the percentage of their overall contributions to the projects using Spearman's correlation [301] and Kendall's correlation [289]. We observed a Spearman's correlation value of 0.86 for Travis CI, and of 0.74 for GitHub Actions. The associated p-values were extremely small ($p < 0.001$), indicating strong evidence against the null hypothesis of no correlation. This significant correlation is visually represented in Figure 4.8 using a scatter plot of the ranked values for both Travis CI, and GitHub Actions, and we also illustrate the average for both tools with an additional line. We also found Kendall's correlation value of 0.68 for Travis CI and of 0.56 of GitHub Actions, with associated low p-values ($p < 0.001$). These results show that the developers responsible for modifying the CI/CD configuration file are likely to be the ones contributing the most to the ML projects and thus, should have the

most expertise in those projects.

RQ4 Findings: *Developers with deeper knowledge and prolonged involvement in the project are more inclined to modify CI/CD configurations.*

4.5 Threats to Validity

Construct validity. The manual analysis of RQ1 and RQ2 might be insufficient to evaluate the exact changes happening in a commit. Since there is no existing taxonomy to use as a reference for RQ2, the two authors independently reviewed and categorized the changes in the sample commits. The categorization process took into account various sources, including code diffs, commit messages, commit descriptions, and pull request discussions. Then, the authors held a meeting to discuss the conflicts and reach a consensus. Additionally, for assessing developer expertise in RQ4, as there is no exact mean of measurement, we leveraged change history information, a reliable metric well-documented in the literature [204, 217, 105, 21].

Internal validity. We acknowledge the potential for selection bias in our sampled dataset of CI-modifying ML commits, utilized in RQ1 and RQ2, which may not fully represent all changes in CI/CD commits. We reduce this threat by using random sampling with a 95% confidence level and a 0.05 margin of error when creating this sample. Furthermore, the observed coupling between commit categories may be due to randomness. To mitigate this threat, minimum support and minimum confidence thresholds were applied.

External validity. Our study focused on open-source ML projects using Python as their primary programming language, which is the most popular language for ML projects [127]. We acknowledge the limitation of generalizing our findings to closed-source projects and those developed in different programming languages. To address this limitation, our dataset was diverse, encompassing projects of varying sizes, ages, and commit frequencies. Furthermore, we only studied projects that have Travis CI or GitHub Actions as their CI/CD infrastructure, which is considered the most used CI/CD tool for open-source ML projects [277].

4.6 Related Works

4.6.1 CI/CD Bad Practices and Barriers

The challenges of adopting CI/CD pipelines have been highlighted in several research works. Duvall et al. [80] first identified several common barriers related to using CI/CD pipelines such as maintenance, managing dependencies, and handling different environments. Zampetti et al. [352] also defined a catalog of 79 bad smells encountered by developers, leveraging interviews with experts and analyzing Stack Overflow posts. Hilton et al. [147] studied the challenges faced by developers when moving to CI, which involve multiple aspects such as quality assurance, security, and flexibility. Similarly, Olsson et al. [238] examined the barriers of migration towards CD. Also notable is Gallaba & Mcintosh's work [100] that discussed the misuse of Travis CI in open-source projects.

4.6.2 CI/CD in Machine Learning Projects

There is limited literature studying CI/CD usage within ML projects. Some works [168, 265] found the traditional testing practices in existing CI services to be insufficient when it comes to ML applications and proposed new CI systems more tailored to the specifications of ML testing specifications. Rzig et al. [277] was one of the first researchers to empirically study and characterize CI adoption rate, tasks, and build failures in ML projects compared to general OSS projects. However, his work mainly focused on analyzing CI adoption without delving deeper into the changes occurring in CI/CD configuration and the developer expertise needed to perform those modifications.

4.6.3 Software Evolution

McIntosh et al. [208] empirically studied the evolution of build systems in open-source projects and found that build files have a high churn rate and are tightly coupled with source code and test files, which means that they need constant maintenance as the source files and test files changes. Jiang et al. [160] explored the co-evolution of Infrastructure-as-Code (IaC) files and found IaC files to be tightly coupled with other software artifacts. Barrak et al. [31] focused on the co-evolution of Data Versioning Control (DVC) files and ML source files, and found a tight coupling between DVC and software artifacts and a non-constant complexity trend for DVC files in 78% projects. Zampetti et al. [351] studied the evolution of CI/CD pipelines by evaluating the restructuring actions occurring in the CI/CD changes. Unlike Zampetti et al. [351], we want to understand the co-evolution between changes happening

in the CI/CD pipeline configuration and the ones happening in ML source code. We also analyzed the developer expertise needed to perform those changes.

4.7 Implications

For ML Developers: The study's findings underscore the critical importance of managing dependencies and testing procedures in ML projects, as these areas are likely to receive frequent changes and often need adjustments in build policies. ML developers need to pay extra care in these areas and try to avoid bad practices like managing dependencies directly in CI/CD configuration and not using testing frameworks with auto-discovery. These practices can lead to CI/CD maintenance overhead and bugs. Furthermore, as ML projects tend to use large datasets and complex computations, ML developers need to utilize caching mechanisms and job parallelization to enhance the performance of the CI build.

For ML Tool Builders: The study highlights a significant opportunity for tool developers to streamline the CI/CD process for ML developers. The limited adoption of continuous deployment among ML developers when updating CI/CD presents a valuable opportunity for tool builders to develop and provide solutions tailored to the specific needs and challenges of the machine learning development workflow. Furthermore, building upon our identified change patterns in RQ3, there is an opportunity for CI tools to become more tailored to ML projects. This could involve enhanced documentation features to assist in the creation of CI configuration files tailored to the specific needs of ML development. Additionally, incorporating prompts with commonly used commands could facilitate the onboarding of less experienced developers, thus mitigating the perception that only experts can effectively modify CI/CD files, as observed in our earlier findings. Our dataset of change patterns from RQ3 can also be used for improving static analysis tools.

For Researchers: Our findings reveal a prevalence of bad practices among ML developers in CI/CD processes, presenting an opportunity for researchers to delve into this domain. Researchers can leverage the existing list of change patterns to conduct in-depth investigations into code smells and bad practices within ML projects. This approach allows for the development of tailored guidelines and best practices aimed at improving the overall quality and efficiency of CI/CD workflows within the machine learning development ecosystem.

4.8 Conclusion

In this paper, we presented the first empirical analysis of how CI/CD configuration changes and co-evolves with ML code during the life cycle of ML projects. Moreover, we performed

CI/CD change pattern analysis and evaluated the expertise of ML developers who manage CI/CD configurations. Our analysis found that over half of commits include updates to the build policy and minor changes related to performance and maintainability. We also revealed several bad practices performed by ML developers which include managing dependencies directly in CI/CD files, using deprecated code, and not utilizing testing frameworks with auto-discovery. Moreover, the pattern analysis identified common integration and delivery features widely used in different CI/CD execution phases. At the same time, our developer expertise for CI/CD maintenance identified that the pipeline is mostly managed by experienced developers, which indicates limited knowledge of CI/CD among the ML development community. We hope that our findings on CI/CD change analysis on ML projects will allow future researchers to develop techniques for automatic incorporation and synchronization of the CI/CD pipeline for ML projects.

CHAPTER 5

CIMig: An Automated Approach of Migrating Continuous Integration (CI) System

This work is currently under submission in the Journal Transactons on Software Engineering and Methodology (TOSEM). This work was done in collaboration with Amazon Web Services.

5.1 Introduction

Continuous Integration (CI) is a widely used software engineering (SE) process for automatically integrating changes in shared repositories. It has enabled drastic change and improvement in SE processes and outcomes, such as quicker issue resolution and faster shipping [148, 350, 356]. There is a wide range of CI tools that help developers automate their development workflow. Many popular contemporary CI tools exist, such as GitHub Actions, TravisCI, GitLab CI/CD, Azure DevOps, CircleCI, and Jenkins. Each of these tools offers its own benefits to accommodate individual software projects' specific needs and constraints [298]. Soares et al. [298] discuss how CI tool-support has undergone significant shifts recently, driven by the emergence of new competitors, expanded operating system support in existing tools, alterations in billing policies, changes in the organizational or community structure of CI providers, improvements in service reliability and performance, among other factors. As a result, developer migration between CI tools has become a common and frequent phenomena [125]. Prior research [272] identified that CI-platform migration is a difficult task and that "*lack of developer familiarity with the new CI*" is one of the main reasons for CI-migration abandonment. However, these migrations are slow and error-prone due to various factors [272], and further complicated by a lack of tool support. The only official tool among prominent CI tools is GitHub Actions Importer [117], which only supports migrating to GHA, relies on hand-crafted mapping rules, and lacks support for features such as the migration of secrets like authorization tokens [112]. Moreover, this tool is technology-specific

and can not be applied to migrate to or between other CI systems. This highlights the need for a technology-agnostic approach to CI migration.

Most existing automatic migration research works focus on analyzing and migrating source code between programming languages [88, 83, 219, 15, 231], and some works focused on with the analysis and migration of configuration code [145, 124, 321, 260], but none tackled the automatic migration of CI configuration code. Many differences exist between source code and configuration code. Source code defines the behavior of software, relies on programming languages like Java that have logic and behavior-describing syntax, and is generally managed and documented by developers. Configuration code describes the parameters of a software application [37], relies on markup languages like YAML or domain-specific languages (DSLs) [325] with higher abstraction level than programming languages, and is generally maintained by DevOps engineers [325]. The migration of CI systems is challenging because of the many possible differences between Source and Target CI systems [272], owing to the aforementioned usage of DSLs with higher abstraction. Moreover, our analysis identified that these migrations require multiple iterations and a significant time span to achieve stability in the Target CI system.

To mitigate this difficulty, we propose a novel technology-agnostic approach *CIMig* that employs example-based mining to migrate CI configurations between CI systems. The proposed approach CIMig automatically learns rules from semantically-equivalent tuples of CI files originating from different tools, then applies its learnings to migrate CI files from a Source CI system to a Target CI system. Since Travis CI and GitHub Actions (GHA) are the most popular CI tools for Open Source Software (OSS) projects [125, 148, 276], and migrations occur frequently between these two tools [125], we evaluated CIMig for migration between GHA and Travis CI. We assessed the results of CIMig through automatic and manual evaluations, described its cost, and analyzed some of the cases where it failed. Through this paper, we answer the following research questions:

RQ1: What is the accuracy of the migration pipelines ?

RQ2: What is the cost of the migration pipelines?

RQ3: What are the limitations of our approach?

CIMig can translate 70.82% of a Travis CI file and 51.86% of a GHA file on average. Its translations from Travis to GHA are competitive with GitHub Actions Importer, where they had an average cosine [281] similarity of 0.51 to the developer's hand-crafted manual translations, versus 0.45 achieved by GitHub Actions Importer. Unlike the latter, CIMig also translates syntax in the opposite direction, where it generates files with an average 0.35 cosine similarity to the developer's versions.

Our main contributions through this work are:

- A novel technology-agnostic CI migration technique leveraging Apriori Rule Mining and Tree Association Rules.
- A comprehensive evaluation of the effectiveness of CIMig, and of a few important failure scenarios.
- A dataset of GitHub Actions and Travis CI configuration files from 30,543 real-world Java projects shared at [18].

We motivate our work within Section 5.2. We discuss its background in Section 5.3. We detail our approach in Section 5.4, and the quality, cost, and shortcomings of applying techniques to migrations between GHA and Travis CI within Section 5.6. Related works are detailed in Section 5.7, the threats to validity are discussed in Section 5.8, and finally, we conclude our work in Section 5.9.

5.2 Problem Contextualization

Prior research [272] has identified through qualitative analysis that migrating a CI infrastructure is a difficult process due to technical and human hurdles. A recent study [354] on CI migration pointed out that migration to GitHub Actions is complicated, with developer comments like "Migrating to GitHub Actions from Travis keeps failing." To further validate these findings, we performed an empirical study, where we analyzed 1252 projects that migrated from Travis CI to GHA, one of the most common migration patterns [125, 272]. These projects were collected through a process detailed in Section 5.4.1, and are manually confirmed to have created an equivalent GHA file.[1]. In GHA and Travis CI, some workflows may be reported as "No Data", such workflows are assumed to be failed unless they are preceded by a successful workflow, similar to other works that analyzed Build and CI systems [143, 276, 279].

Through our Git and API-based analyses, we uncovered that an average 71.20 days, and 2.75 commits are needed to reach a successful build that corresponds to the equivalent GHA file, with some projects needing up to 169 commits to reach this threshold. This implies that the migration process is not self-evident and requires multiple attempts over an important span of time. Furthermore, we find that 48 projects seemingly abandoned the migration process entirely even though they implemented equivalent GHA files, as they never achieved a successful GHA workflow.

[1]Defined as sharing 50% or more functionality, detailed in Section 5.4.1

To better illustrate the complexity of the CI migration process, we present an example of a migration from Travis CI to GHA from the project `VocableTrainer-Android` in Figure 5.1. The semantically-equivalent segments of the configuration are marked with a + sign and linked with an arrow in Figure 5.1. However, the sections marked with an x sign, have no direct equivalents between the two syntaxes.

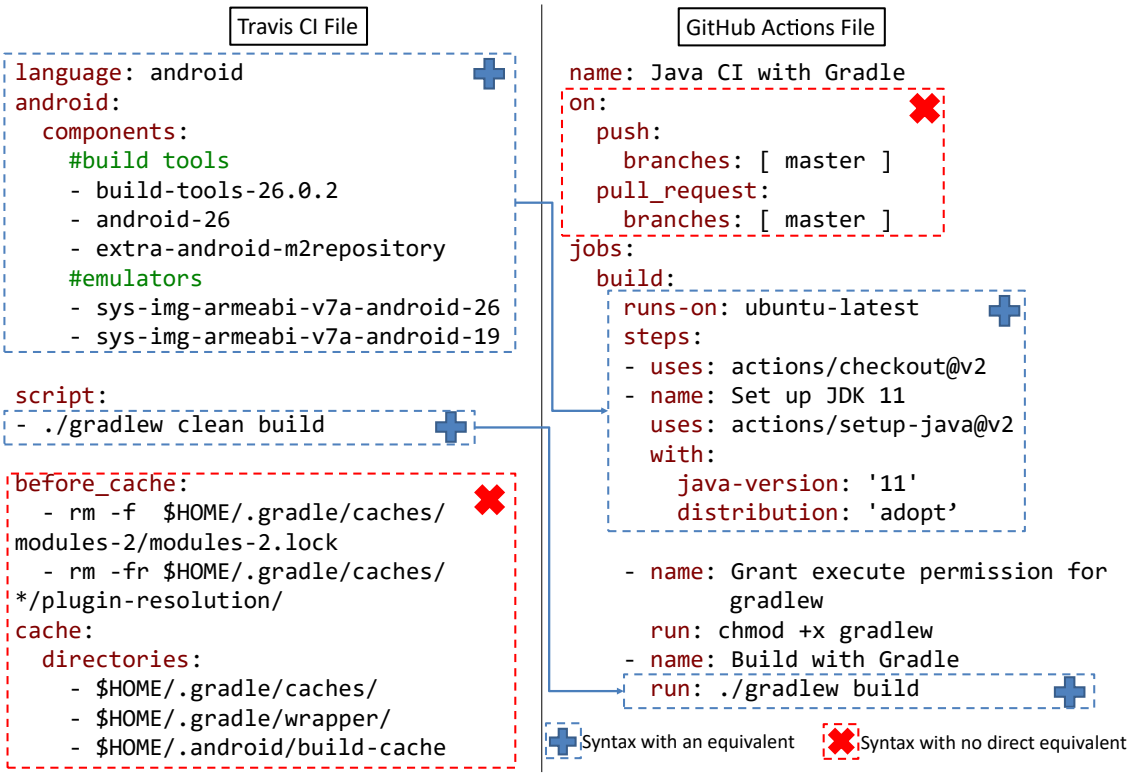


Figure 5.1: Example of Migration from Travis CI to GitHub Actions

The Travis CI configuration of this project requires specifying `android` as a language and manually configuring the different `components` required to run this project within the Travis CI environment. However, that is not necessary within GHA, as all of these components are provided by default when using the `ubuntu-latest` environment.

While Travis CI automatically performs the checkout process and makes Gradle executable, these steps need to be explicitly performed in GHA. Travis CI workflow execution triggers are configured via its website or performed via API requests [58]. But, GHA developers need to specify them within the `on` section of the GHA configuration file. In addition, while Travis CI provides a generic cache configuration mechanism, GHA does not have a workflow-wide syntax that is directly equivalent to caching. It relies on the configuration of job-specific caches by using `actions/cache@v3`, or package-manager-specific caching keywords. `cache:gradle` can be added to this example to ensure caching.

Overall, this example illustrated how developers need to navigate and avoid many pitfalls during the translation of a CI configuration file and how the lack of direct equivalents of some syntaxes hinders the translation process.

5.3 Background

5.3.1 Continuous Integration

Continuous Integration tools automate code integration by automatically validating new commits via the execution of building, testing, and other processes. Most CI tools are configured via Configuration code files, and the execution of a CI tool is referred to as a workflow.

GitHub Actions [117], Travis CI [314], Azure Pipelines [26], and Circle CI [63] are among the most popular CI tools, and have many commonalities. All four tools rely on YAML-based [60] files to store the configuration of their workflows, with each relying on its own Domain Specific Language (DSL). For all four tools, workflows can be manually or automatically triggered by Git events such as pull requests or pushes. Workflows are composed of one or more jobs, which may be configured to execute in parallel in different environments. For each of these tools, a job may be composed of one or more steps that run sequentially, and it's possible to use variables to share information between the different steps and the different jobs.

Even with the functional similarity of these tools, there are significant conceptual and syntactical differences between them. For example, while the Operating System for each GHA job may be specified using the `runs-on` [114] keyword, or the keyword `vmImage` [28] for Azure Pipelines, or the `image` [64] keyword for CircleCI, Travis CI uses the keyword `os` [312] to configure it for all stages and jobs. While Travis CI has a specific phase `install` [313] within its lifecycle to prepare the environment, GHA, Azure Pipelines, and CircleCI leave the specification of these phases to the developers. GHA makes workflows and jobs reusable with the keyword `uses` [115], so does Azure Pipelines with `task` [27], and CircleCI via `orbs` [65], but, Travis CI does not offer an equivalent function.

5.3.2 Example-based Learning

A plausible approach for automatic CI system migration is to learn from how prior developers migrate from source CI systems to target CI systems and how they compose the structure of the CI configurations. Such migration and composition data can be extracted from open-source projects hosted in GitHub. We utilized Association rule mining [179, 85], an ML

approach for finding interesting associations among data. Specifically, we used the Apriori Rule Mining [5] and Frequent-Tree Mining [56] algorithm to generate rules for the target CI system.

5.3.2.1 Apriori Rule Mining

Apriori is an Association Rule Mining (ARM) algorithm defined by Agrawal et al. [5]. It starts by finding the frequent individual items in a database, also known as transaction set, and expands them to item sets co-occurring together as long as the appearance of those item sets is larger than a minimum threshold specified by the user. Apriori then uses these frequent item sets to generate association rules that reflect general trends in the transactions set. Apriori rules are composed of a Left Hand Side (LHS), the antecedent, also referred to as pre-condition, and a Right Hand Side (RHS), the consequent. Within our work, the transaction set as well as the resulting rules, are composed of subsets of Abstract Syntax Trees (sub-ASTs).

5.3.2.2 Frequent Tree Mining and Tree Association Rules

Frequent Tree mining empowers us to discover frequent maximal, induced, ordered sub-trees with a specific minimum support from a group of similar trees. We performed Frequent-Tree Mining via the `CMTreeMiner` [56] algorithm on subsets of Abstract Syntax Trees (ASTs). We grouped these sub-ASTs by their root nodes and passed them as input to `CMTreeMiner`. Frequent Trees are discussed in detail in Chi et al.'s work [57]. Using these trees, we were able to extract Tree Association Rules (TAR), which we adapted from the work of Mazuran et al. [203]. Similar to association rules, TARs are composed of an antecedent and a consequent. Within our work, we considered the antecedent to the root node as well as 50% of the branches of a Frequent Trees, and the consequent being the remaining branches of the tree. Hence, a Frequent Tree may generate multiple TARs during the execution of CIMig.

5.4 Approach

An overview of our approach is shown in Figure 5.2. While CIMig is technology agnostic, we use Travis CI and GHA syntaxes in the different examples to better illustrate our approach.

5.4.1 Data Collection and Preparation

As an example-based learning approach, CIMig requires a set of examples to learn from. Specifically, CIMig requires three sets of configuration files. Set (1) containing Source CI files

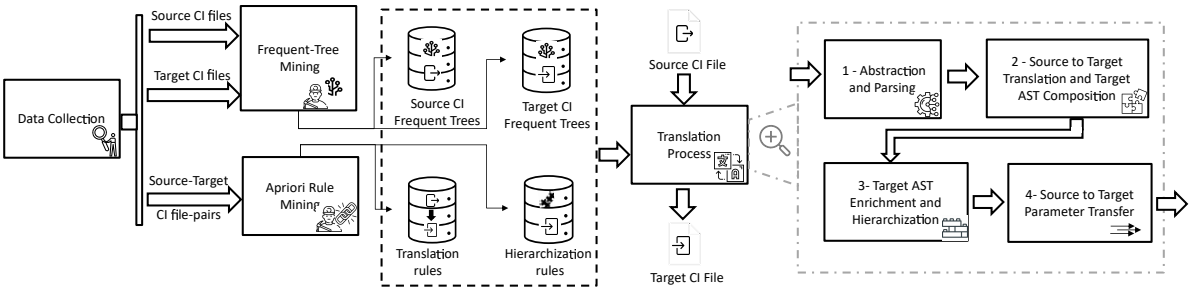


Figure 5.2: Overview of CIMig when used to migrate between Travis CI and GHA

, Set (2) Target CI files, and Set (3) of Source and Target CI equivalent-file-pairs, with each pair composed of two files from two different CI tools that implement similar functionality. To evaluate our approach, we chose to focus on Java projects using Travis CI or GHA as they are the most important subset of CI-using OSS projects [148, 33, 77, 276, 279], but CIMig can be used with projects using any programming language. To concertize the data preparation phase in this context, we discuss the processes we followed to create the 3 aforementioned sets in this specific context. First, we collected the projects from two sources: Google BigQuery and GitHub, the two most popular OSS repository hosting sites [185, 116], after applying criteria on activity and popularity as outlined by previous works [221, 166, 131], ensuring that these projects have a size > 0 KB, have been active in 2021, and have a popularity ≥ 5 stars or ≥ 5 forks. We collected 345228 projects after de-duplication. Then, we used Travis CI and GHA APIs to establish a project's usage of these CI tools, a more accurate method of establishing adoption [276].

This allowed us to build these three project sets:

- Travis CI-Only projects: 13403, $\rightarrow$ Set (1) or Set (2)[2]
- GHA-only projects: 15888, $\rightarrow$ Set (1) or Set (2)[2]
- Travis CI and GHA projects: 5138, 1252 after filtering, $\rightarrow$ Set (3).

We used the first two projects sets to extract Set (1) and Set (2) for Task B in 5.4.2, to extract Frequent Trees for both Travis CI and GHA. We used the third project set to perform migration effort analysis discussed in Section 5.2, and to extract Set (3) of semantically equivalent configuration code file tuples for Tasks A-1 and A-2 in 5.4.2. It's important to note that while Travis CI uses one configuration code file, GHA may use multiple files, hence why we're extracting tuples from a project, as they contain one Travis file, and may contain more than one GHA file. We applied the following filtering process to find these tuples.

First, we performed a git-history-based analysis to extract the Travis CI and GHA file tuple which contains the file pair composed of the Travis CI and one of the GHA files with the

[2] depending on Translation direction.

highest cosine similarity [281], a metric used within previous works [54, 338] to determine source code and configuration code similarity, and which have passing build statuses as confirmed by the GHA and Travis CI APIs. We eliminated 1748 projects as they did not have configuration files for one or both tools in their history, likely due to Git rewritings [39], and 919 projects, due to their tuples having a maximum cosine similarity of 0.1 or less.

Second, to confirm the semantic equivalence of the remaining tuples from the third set, two co-authors manually analyzed file tuples from 2471 projects. As mentioned earlier, the extracted file tuple may contain more than one GHA file. Hence, the developers performed a pairwise comparison between the Travis file and each of the GHA files in each tuple, starting with the longest GHA file. The co-authors focused on the shorter file out of the file pairs, determined the functionality being performed via the goals of commands and CI-specific DSL, and then, then attempted the same process in the other CI file and its DSL. To save time, they stopped when reaching the minimum equivalency criterion of 50%. We opted for this permissive semantic equivalence criterion after perceiving that very few projects completely re-implement the same functionality between GHA and Travis CI, which is consistent with previous findings [272]. After applying these filtering processes, only 1252 projects met these criteria. In the tuples where more than one GHA file was present, we designated the biggest GHA file meeting the equivalence criterion it as the "main" GHA file. In total, we extracted 1252 Travis files and 1372 GHA files.

Third, similar to other works that tackled code translation [274, 6], we split third set into two subsets, following the 80%-20% ratio, a "training" set of file tuples from 1001 projects and a "test" set of file tuples from 251 projects. The project splitting process was random to maintain representativeness, and no project is represented in bot the training and testing set. Only the training set is used for Tasks A-1 and A2, while the testing set is reserved for the evaluation of the approach. Since Tasks A-1 and A2 of CIMig are designed to learn on file-pairs, we transform each tuple into pairs where the same Travis CI file is paired with the multiple GHA files.

Finally, we apply an Abstraction process to Sets (1), (2), and (3) .This process parses the different configuration files into equivalent ASTs, then transforms their leaves by matching them with regular expressions that contain predefined keywords to preserve the commands used within the configuration code files while removing their project-specific parameters. The keywords used within the abstraction process were defined via a clustering method, where we collected the 100 most common keywords within Travis CI and GHA files, respectively, and manually extracted 22 keywords corresponding to popular commands used within these files. These commands correspond to tools such as `Git` and `Maven` that are external to these two CI tools and may be used by any CI system. This is the only part of the abstraction

process that is programming-language-specific, as the tools detected via this process, such as Maven, might be more closely associated to a specific programming language. However, this process can be easily adapted to other programming languages by using Sets that use other programming languages for this process, or simply manually defining the keywords depending on the programming language used in the projects associated to the CI files.

5.4.2 Training CIMig

5.4.2.1 Task A: Apriori Rule Mining Process

The goal of this process is to find rules that allow us to translate Source CI syntax to Target CI Syntax, which we refer to as ① Translation Rule Mining as well as rules that link different parts of Source CI syntax to each other, referred to as ② Hierarchization Rule Mining. This process is applied to Set (3): the Source-Target CI file pairs set.

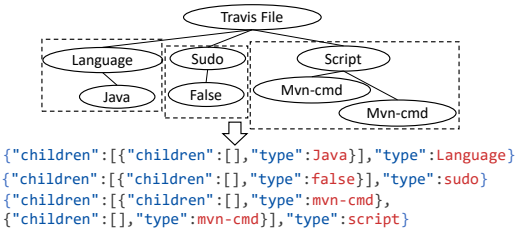


Figure 5.3: Travis CI H-2 AST Extraction Example

Table 5.1: Subset of Cartesian product generated for a Travis CI - GHA File tuple

Travis H-2 AST	GHA H-2 AST
{children:[{children:[],type:mvn-cmd}], type:script,origin:travisci}	{children:[{children:[],type:name}], type:name,origin:gha}
{children:[{children:[],type:mvn-cmd}], type:script,origin:travisci}	{children:[{children:[],type:branch-name}], type:branches,origin:gha}
{children:[{children:[],type:mvn-cmd}], type:script,origin:travisci}	{children:[{children:[],type:build}], type:name,origin:gha}
{children:[{children:[],type:mvn-cmd}], type:script,origin:travisci}	{children:[{children:[],type:adopt}], type:distribution,origin:gha}

Task A-1: Translation Rule Mining. To extract translation rules to guide our translation from the Source CI to the Target CI tool, we analyze the previously-prepared file pairs.

First, after the abstraction of these files as detailed within Section 5.4.1, we parse them into ASTs. Then, for each pair of ASTs, we extract the sub-ASTs of height equal to 2 starting from the leaves of the ASTs, which we refer to as H-2 ASTs within this work, and we represent them in a textual format. We decided on this height after a process of parameter tuning, detailed in the Parameter Tuning paragraph of 5.4.2.3. An example of the application of the abstraction and H-2 collection processes is shown in Figure 5.3. For each file pair i, we obtain a set of H-2 ASTs extracted from the Source CI file: $SRC - H2_i$, and a set of H-2 ASTs extracted from the Target CI file $TGT - H2_i$

Second, for each file pair, we apply Cartesian product, used in other code-translation works [310], to create a transaction set $T_i = SRC - H2_i X TGT - H2_i$ We chose this product since the alignment of the configuration code files from different tools is not possible in many of cases, as different configuration parameters can be at different locations within the file

pairs due to some tools, such as GHA, employing a more flexible file structure than others.

Third, all the transaction sets generated from the file pairs are grouped into one large set $T_{transl} = \sum_{i=1}^{N} T_i$ on which we perform our Apriori-based Association Rule Mining (ARM) [5], previously detailed in Section 5.3.2. This rule-mining was used in other works tackling code translation and configuration mapping such as that of Hora et al. [153]

These rules are evaluated in terms of their support [5], confidence [5], and lift [209], with higher values indicated higher quality rules. Support reflects how often the item set appears together, $support(SRC \Rightarrow TGT) = P(SRC \cup TGT)$, where SRC is a specific H-2 AST from Source CI, TGT is a specific H-2 AST from Target CI. Confidence reflects how often the rule is correct, $confidence(SRC \Rightarrow TGT) = {}^{P(SRC \cup TGT)}/_{P(SRC)}$. Lift is the ratio of the actual confidence of a rule to its expected confidence, $lift(SRC \Rightarrow TGT) = {}^{confidence(SRC \Rightarrow TGT)}/_{P(TGT)}$. We specified a minimum support of 10^{-6}, a value determined via a process detailed in Section 5.4.2.3.

Fourth, we filter the generated rules to keep those with the format of SRC-CI-H2-AST $\Rightarrow$ TGT-CI-H2-AST, thus creating the Rule-Set R. We calculate the confidence, lift and support products of these rules with their flipped counterparts, of the format TGT-H2-AST $\Rightarrow$ SRC-CI-H2-AST, to substantiate the equivalence between the two H-2 ASTs. This is important since one Source CI H-2 AST may have multiple possible equivalent Target H-2 ASTs, and vice-versa.

Finally, we automatically bifurcate R into two sets, based on whether the cosine similarity of the LHS's leaves and the RHS's leaves was above 0.5. These sets are:
R_{sim}: Similarity-Based rule-set.
R_{stat}: Statistical-Based rule-set.

We opted for this bifurcation as we anticipate a number of spurious rules will be present in R due to the application of the Cartesian product. We choose to apply R_{stat} in the translation process, after applying additional constraints, as it may still contain some useful non-textually-similar rules. We detail these constraints and the translation process in Section 5.4.3.

Task A-2: Hierarchization Rule Mining. The translation rules only capture two levels of the entire Source CI AST to find its equivalent Target CI AST. However, CI ASTs often contain 3 or more levels. Thus, multiple H-2 ASTs can be linked with a variety of intermediate nodes on multiple levels. To better infer the intermediate nodes within a Target AST, and ensure the correct composition of the generated Target CI file, we created a set of hierarchization rules via the following steps.

First, from each Target CI file i, we built a transaction set TH_i of the extracted H-2 AST nodes and their parents, then we built $TH = \sum_{i=1}^{N} TH_i$ on which we ran the Apriori

algorithm with a minimum support of 10^{-6} to generate the hierarchization rules.

Then, similar to their translation counterparts, the rules were filtered to keep those of the desired format of H-2-AST-Child ⇒ Parent. These rules allow us to find the direct parents of an H-2 AST, thus allowing us to infer some intermediate nodes within the complete generated AST of our Generated Target CI file. In addition, their confidence, lift, and support products were also calculated.

5.4.2.2 Task B: Frequent-Tree Mining Process.

While our translation and hierarchization rules allow us to translate H-2 ASTs and find their direct ancestors, they do not capture patterns that link multiple H-2 CI ASTs to each other or patterns that span more than 2 levels. Such patterns may allow us to add beneficial sub-ASTs via inferring which sub-ASTs occur together, thus allowing us to address syntax that is not directly translatable from the Source CI syntax or syntax that does not have a direct equivalent. To capture these useful patterns, we perform Frequent-Tree Mining, detailed in Section 5.3.2, on the Source CI files set, and the Target CI files set. After abstracting these files, we extract sub-ASTs originating from each of their intermediate nodes. This mining process empowered us to discover a set of Frequent sub-ASTs, which we refer to as FT, where each tree has a minimum support of 5%, a value we chose via parameter tuning, detailed in Section 5.4.2.3.

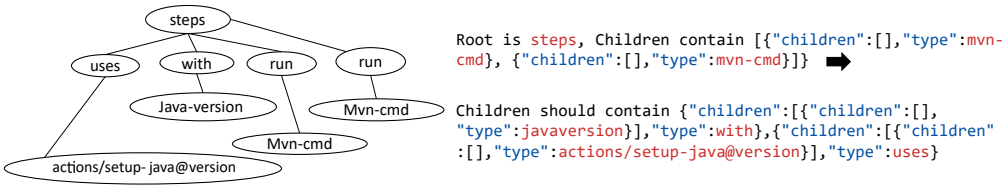


Figure 5.4: Example of Frequent Tree mined from GHA and Generated TAR

These Frequent Trees we extracted capture sub-ASTs which co-occur frequently within the files we used as input, and an example of such a Frequent Tree containing a beneficial pattern, which we extracted by mining GHA files, is shown within Figure 5.4. This tree contains the syntax used to setup and configure Java within a specific job, as well as the usage of Maven commands, signaling that these two elements are likely to occur together. As detailed in Section 5.3.2, these Frequent Trees generate multiple Tree Association Rules (TAR), and an example TAR is shown in the figure as well, where the antecedent is the root node steps along with the usage of Maven commands within an AST, and the consequent is the setup and configuration of Java. Such co-occurring H2-ASTs can't be identified by translation and hierarchization rules. As illustrated by this example, the configuration of

Java is a beneficial addition to our translation. Furthermore, TARs generally add more intermediate nodes to an AST file, which are useful for the hierarchization process.

5.4.2.3 Parameter Tuning for Task A & B

While designing the Apriori Rule Mining and Frequent Tree Mining processes, we followed an extensive parameter tuning process. Due to the prevalence of the migration from Travis CI to GHA, we focused on that translation scenario during this process. For the Apriori Rule Mining tasks, while deciding on the optimal number of levels to capture within our translation rules, our goal was to generate rules that strike a good balance between conservativeness and generality, as higher-order rules may less easily accommodate the migration with new CI workflow steps not seen during the training process, while lower order rules may generate too many rules that are prone to noisiness. To determine the ideal number of levels to consider, we mined rules with different sub-ASTs of different heights. We specifically evaluated 3 different types of rules: H-2 rules, with two levels on both sides of the rules, Mixed rules, with three levels on one side, two levels on the other side of the rule, and H-3 rules, with three levels on both sides of the rules. For each type of rule, we mined Travis CI => GitHub Actions rules, and then performed an evaluation of these rule sets against hand-crafted ones. While Sim-based H-2 and Stat-based H-2 rules had F-1 scores of 71.25% and 31.85%, Sim-based Mixed and Stat-based Mixed rules had F-1 scores of 60.75% and 8.10%, and Sim-based H-3 and Stat-based H-3 rules had F-1 scores of 33.33% and 10.26%. Hence, it's clear that H-2 rule-set has significantly better rules, while considering higher-level rules causes a precipitous drop in rule quality. Furthermore, while performing the rule mining process, we experimented with different values for the minimum support, and opted to use 10^{-6} as it allows the generation of the maximum number of rules on the development machine, described in 5.5.2, without causing memory consumption issues related to the Apriori algorithm [5]

Concerning Frequent Tree Mining, we again aimed to strike a balance between conservativeness and generality, and to operate within the constraints of time and memory needed for the mining process. Hence, we attempted the mining process with multiple minimum support values ranging from 1% to 75%. Of the values in this range, we found that a minimum support of 5% generated a sufficient number of trees, consisting of 2664 GHA Frequent Trees and 524 Travis CI Frequent Trees, within an amount of time detailed in 5.5.2, while higher support values resulted in a much smaller number of Frequent-trees, especially for Travis CI. For example, 10% minimum support resulted in the discovery of only 1006 GHA trees and 175 Travis CI trees, and 25% resulted in the discovery of only 191 GHA trees and 40 Travis CI trees, thus capturing far fewer patterns. Frequent Tree Mining with minimum support

values lower than 5% either went on indefinitely, or took much longer time and resulted in few additional trees, most of which were not generalizable.

5.4.3 Using CIMig

The four steps of the approach that we follow to translate files from the Source CI syntax to the Target CI syntax are illustrated in Figure 5.2. Within this section, we detail the different steps of the translation process and illustrate them with an example of a translation from Travis CI to GHA. Similar to GitHub Actions Importer, we designed CIMig to use one file as input and produce one file as output, as the splitting of CI configuration across all files is optional in some CI tools such as GHA, and not at all supported by other tools such as Travis CI.

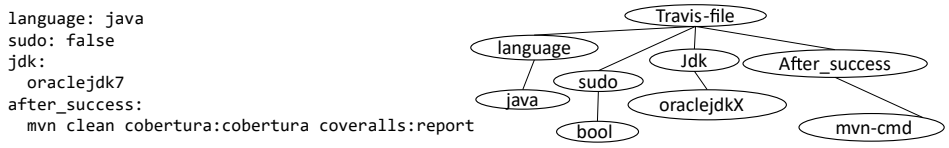


Figure 5.5: Example of Travis CI File and its corresponding AST

5.4.3.1 Step 1: Abstraction and Parsing

First, the configuration code of the source file is processed with the same abstraction process applied during the training phase, described in Section 5.4.1, and then parsed to an Abstract Syntax Tree (AST) from which we collect the H-2 ASTs. The parameters of the commands within these nodes, which are removed in the abstraction process, are stored for usage in a later step. An example of a Travis CI configuration file and its equivalent abstracted AST is shown within Figure 5.5.

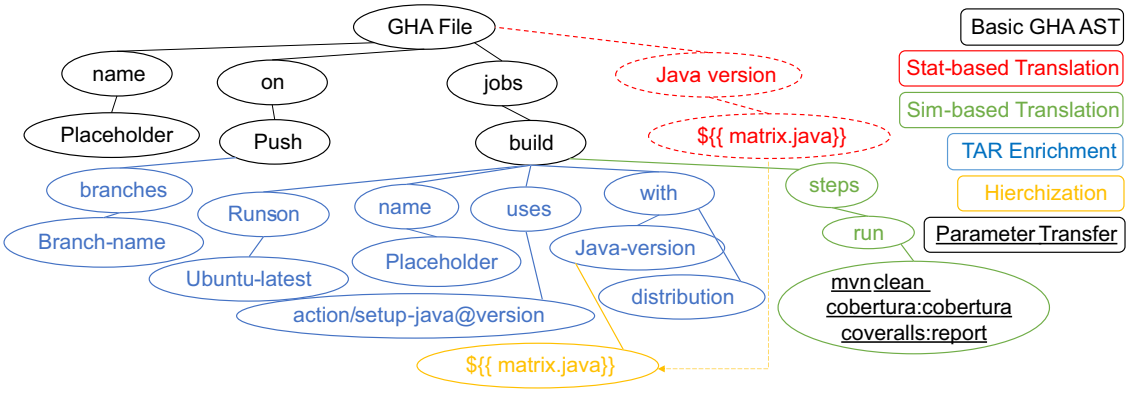


Figure 5.6: Example of a translated GHA AST

5.4.3.2 Step 2: Source to Target Translation and Target AST Composition

This step is composed of 3 phases: Initialization, Sim-based Translation, and Stat-based Translation. We also detail the Insertion Process we follow during the latter two phases.

Step 2.1: **Initialization.** First, we initialize a Target CI *seed tree* before the translation process begins. This AST is created from Frequent Trees found within the Target CI files, and that was verified to follow the correct structure of the Target CI tool. An example of a basic GHA AST composed of a seed tree is shown in black in Figure 5.6. This AST forms the basis of the file we're attempting to create as an end result of our translation process, and we refer to this file as generated equivalent Target CI file.

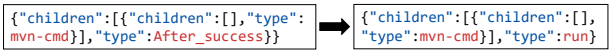


Figure 5.7: Example of Sim-based translation rule

Figure 5.8: Example of Stat-based translation rule

Step 2.2: **Sim-based Translation.** Second, we attempt a Sim-based translation, which makes use of Sim-based rules, detailed in Task A-1 of Section 5.4.2.1. For each Source CI H-2 AST collected within the previous step, we collect all the Sim-based rules with an LHS that matches it. Then, we extract the best rule according to its confidence product and apply it to generate the corresponding Target CI H-2 AST, which is then inserted within the AST of the generated equivalent Target CI file. Figure 5.7 shows an example of such a translation rule that applies to the Travis CI file shown in Figure 5.5. The usage of its results in a generated equivalent GHA file is shown in green in Figure 5.6. These rules effectively translate syntax that is directly equivalent between Source and Target CI systems and has textual similarity, such as translating the `after_success:./gradle` segment from the motivational example in Figure 5.1.

Step 2.3: **Stat-based Translation.** Third, we attempt Stat-based translation. For each H-2 AST not translated within the previous step, we collect all the Stat-based rules with an LHS that matches it. However, we look for certain prerequisites before attempting to apply the Stat-based rules. For each rule, we collected the Frequent Trees of the Source CI tool, which contain the LHS of this rule, and the Target CI Frequent Trees, which contain the RHS of this rule.[3] CIMig analyzes each matched rule in descending order of their confidence product. It ascertains whether at least one Source CI Frequent-Tree, containing the LHS of the Stat-based rule, is present within the Source CI file's AST. It also verifies if a Frequent-tree from the Target CI tool, containing the RHS of the Stat-based rule, has

[3]This step is independent of the translation process, it is pre-computed to help accelerate it.

at least 50% of its branches within this AST. If both conditions are met, the rule is applied and the corresponding Target CI H-2 AST is generated and inserted within the AST of the generated equivalent Target CI file. Figure 5.8 shows an example of such a translation rule that applies to the Travis CI file shown in Figure 5.5. The usage of its results in a generated equivalent GHA file is shown in red in Figure 5.6. This type of rule is especially useful for the non-directly-equivalent syntax and directly-equivalent syntax which does not have textually similar leaves, such as the translation of the `language:android` segment from the motivational example in Figure 5.1.

Insertion Process. In this paragraph, we detail the insertion process that we followed during the Sim-based Translation and the Stat-based Translation. CIMig performs a DFS-based search within the generated equivalent Target CI file AST to find the deepest node that matches the parent node of the new H-2 AST, which is then used as the point of insertion. The H-2 AST's children are inserted as the matching node's siblings. The design of this process was guided by observations of the YAML syntax, which is used by many CI tools, as intermediary nodes do not occur multiple times on the same level within a YAML file. If no matching nodes are found, the new H-2 AST is assumed to be a direct descendant of the file's root node and is accordingly inserted at the root of the file.

5.4.3.3 Step 3: Target AST Enrichment and Hierarchization

Step 3.1: **AST Enrichment with TARs.** to improve the structure of our generated equivalent Target CI file, we make use of TARs contained within the previously-mined Frequent Trees, detailed in Task B of Section 5.4.2.1. TARs can add beneficial patterns found within CI files of the same type, as well as intermediate nodes and structures that can be used to hierarchize the previously generated H-2 ASTs. We attempt to match each of the Target CI tool's TARs with the AST of the generated equivalent Target CI file. If a TAR is applicable, we insert the AST branches it generates while preserving any existing nodes within the file. an example of an AST Enrichment is shown in blue in Figure 5.6 .

Algorithm 1 Hierarchization Algorithm

```
1: function DFS_Based_Insert(CI_H2_AST,CI_AST)
2:     T ← CI_H2_AST.Parent_Node.Type
3:     Insert_Node ← DFS Based Search(CI_AST, T)
4:     if Insert_Node ≠ NULL then
5:         Insert_Node.Children ← (Insert_Node.Children_AST ∪ CI_H2_AST.Children)
6:         CI_AST.Children ← (CI_AST.Children \ CI_H2_AST)
7:         return True
8:     else
9:         return False
10:    end if
11: end function
12: for all CI_H2_AST ∈ CI_AST.Children do
13:    if DFS_Based_Insert(CI_H2_AST,CI_AST) = False then
14:        Matched_Hierarch_Rules ← ∅
15:        for all Hierarchy_Rule ∈ Hierarchy_Rules do
16:            if Hierarchy_Rule.LHS = CI_H2_AST then
17:                Matched_Hierarch_Rules ← (Matched_Hierarch_Rules ∪ Hierarchy_Rule)
18:            end if
19:        end for
20:        if Matched_Hierarch_Rules.size > 0 then
21:            Best_Rule ← Best(Matched_Hierarch_Rules)
22:            New_CI_AST ← Init(Best_Rule.Parent_Node.Type)
23:            New_CI_AST.Children ← (New_CI_AST.Children ∪ CI_H2_AST)
24:            CI_AST.Children ← (CI_AST.Children \ CI_H2_AST)
25:            if DFS_Based_Insert(New_CI_AST,CI_AST) = False then
26:                CI_AST.Children ← (CI_AST.Children ∪ New_CI_AST)
27:            end if
28:        end if
29:    end if
30: end for
```

Step 3.2: **AST Hierarchization.** The goal of the hierarchization process is to improve the placement of our H-2 ASTs, and the internal structure of our generated equivalent Target CI file's AST. We apply Algorithm 1 to achieve this process, which employs the hierarchization rules, detailed Task A-2 of in Section 5.4.2.1. First, as shown in lines 12-13, for each H-2-AST we inserted at the root, we attempt to apply the hierarchization process. Within this paragraph, we refer to the H-2 AST we're attempting to hierarchize as the current H-2 AST. For each current H-2 AST, we call the function DFS_Based_Insert, detailed in lines 1-11, where we perform a DFS-based search to find the deepest node that matches the current H-2 AST's parent type. If a match is found, we insert the current H-2 AST's children as children of the matching node and remove the current H-2 AST from the root node. This re-application of the same insertion process we followed in the previously-described *Step 2* allows us to take advantage of the new intermediate nodes added via the TAR enrichment process that we previously applied. If no matches are found, we collect all the Target CI hierarchization rules the LHS of which matches the current H-2 AST and we apply the

hierarchical rule with the highest confidence product, as detailed in lines 14-21. Lines 22-30, show how we use this rule: we produce a new node using the new parent type, and add the current H-2 AST to its children. We then pass this new node as a search target to DFS_Based_Insert. If a node with the same type as our new parent is found within our Target CI file AST, we insert the current H-2 AST as one of its children. If no matches are found, the newly generated node is inserted a child of the root of the generated equivalent Target CI file's AST.

An example of a hierarchization rule is shown in Figure 5.9. It applies to the generated equivalent GHA file we're constructing in Figure 5.6, where the usage of this rule's results is shown in yellow. This example also illustrates how the application of TARs allowed us to add new intermediate nodes, which were then useful during the hierarchization process.

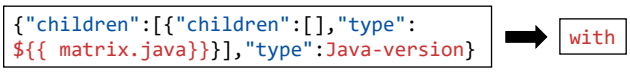


Figure 5.9: Example of Hierarchization rule

5.4.3.4 Step 4: Source to Target AST Parameter Transfer

Before applying the abstraction process in Section 5.4.3.1, we stored the parameters that correspond to the different commands contained in the H-2 ASTs we extracted. Throughout the different steps of our translation process, we keep track of which parameters correspond to each collected H-2 Source CI AST, as well as which generated H-2 Target CI AST corresponds to which H-2 Source CI AST. The generated Target CI ASTs are abstract due to the nature of the rule generation process, making the parameter transfer to them a direct process, where we copy the parameters to the new commands while preserving their order. Hence, we end up with a generated equivalent Target CI file that contains commands identical to their Source CI counterpart. An example of the results of this step is illustrated within the underlined node in the AST shown in Figure 5.6, where the parameters of the maven command were transferred from the original Travis CI AST. The generated equivalent Target CI file AST is finally transformed into a regular YAML file that can be used by the developers in their Target CI environment.

5.5 Evaluation

5.5.1 RQ1: How effective is CIMig?

To measure the effectiveness of CIMig, we performed two-pronged evaluation: automatic translation evaluation and user study.

Automatic Translation Evaluation: To evaluate the performance of the automatic translation, we applied CIMig on "test-set" of 251 that we discussed in Section 5.4.1. We evaluated two aspects of the automatic translation. First, we calculated the percentage of automated translations, which quantifies how many of the H-2 ASTs collected from each source CI file are matched and translated by CIMig. Second, we adopted Cosine similarity [281] and CrystalBLEU [82] to measure the similarity between CIMig generated CI configuration files and developer-written Target CI configuration files. We chose these two metrics due to their wide usage in literature. Cosine similarity is known for its versatility and applicability in source code migration research works [338, 54, 247, 307, 229], and CrystalBLEU is designed for source code similarity and was utilized in code generation works [184, 341, 75] and code migration works [243, 161]. For comparative analysis, we compared the performance of CIMig with that of GitHub Actions Importer, the official tool from GitHub Actions [119], using these two metrics.

User Study: We performed a user study to evaluate the practicality of CIMig. The study was done with five participants, out of 15 initially contacted. They had software development or research experience ranging from 3 to 7 years, CI experience including GHA ranging from 1 month to 1 year. Each participant was tasked with migrating five Travis CI projects to GHA manually. Then, they also migrated these projects semi-automatically twice, with one migration using a configuration file generated by CIMig and another with GitHub Actions Importer, in a random order. In fact, the participants were given CI File 1 and CI File 2 without knowing their generating tools. To ensure a random order, File 1 was generated by CIMig in three projects and by GHA Importer in two projects, while File 2 was generated by the remaining tool.

The five projects are from the Travis CI-only set, and we selected them using popularity ($\geq$ 5 stars or $\geq$ 5 forks), project activity ($>$ 200 commits made), and project freshness (updated June 2023 or later) following criteria in a similar works [131, 166, 221]. The five projects are `hutool` [76], `WxJava` [329], `hsweb-framework` [155], `elasticsearch-sql` [232], `TelegramBots` [35]. For the study, we only considered Travis CI to GHA migrations as GitHub Actions Importer only supports Travis CI to GHA. As stated earlier, all configuration files generated by CIMig and GitHub Actions Importer were anonymized before being shared with the participants to avoid bias.

For each migration task, the participants were asked to achieve a "First Passing Workflow", a workflow that implements minimal CI functionality, and a "Final Workflow" which implements all CI functionality that is available in Source CI configuration. During the study, we measured how much time can be saved via the semi-automatic migration approaches using CIMig and GitHub Actions Importer generated files. Also, we received ratings for the usefulness of the generated files by each tool from participants using a Likert scale [9] ranging from 1 to 5, with 1 being "not at all useful" and 5 being "incredibly useful". Having performed the manual migration first helped the participants determine their user rating (usefulness) of CIMig and GitHub Actions Importer in comparison with the manual process. Our full study guide and the full reports are available at [18].

5.5.2 RQ2: What is the CIMig Execution Cost?

To estimate the time consumption of the training and translation processes, we programmatically measured the time it took to execute each training task, as well as the execution times for each translation performed on our test set, during the experiment execution. We performed our experiment on a machine running Ubuntu 22.04 and configured with an Intel Xeon CPU with 6 cores/12 threads and 32 GB of RAM.

5.5.3 RQ3: What are the Shortcomings of CIMig?

To find the shortcomings of the results of CIMig, we asked co-authors who did not work on developing CIMig to manually evaluate the results of our experiments by providing detailed reports on the different issues they noticed for the worst 25 generated translations from Travis CI to GHA, and the worst 25 generated translations from GHA to Travis CI, as determined by Cosine similarity. We then grouped similar issues into 3 main categories and reported the number of translations of each type that possessed that flaw.

5.6 Results

5.6.1 RQ1: CIMig Migration Effectiveness

5.6.1.1 Automatic Translation Evaluation Results

As the translation percentage values illustrate within Figure 5.10, for the 251 projects composing the test set of GHA-Travis CI equivalent set, our technique is effective at translating an average of 70.82% and a median of 75% of the H-2 ASTs extracted from a Travis CI file.

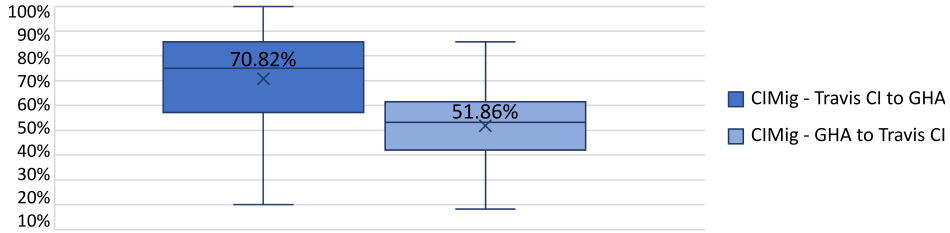


Figure 5.10: Percentage of H-2 ASTs translated per-file

Furthermore, CIMig translates an average of 51.86% and a median of 53.13% of a GHA H-2 ASTs to Travis CI syntax.

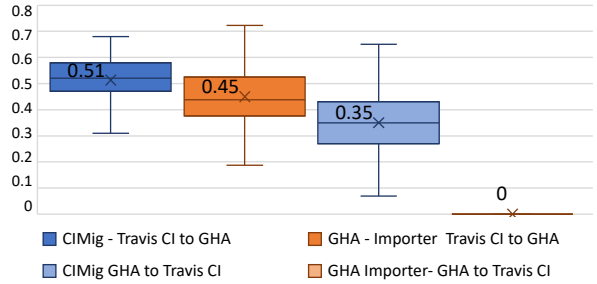


Figure 5.11: Cosine Similarity score of the Generated files.

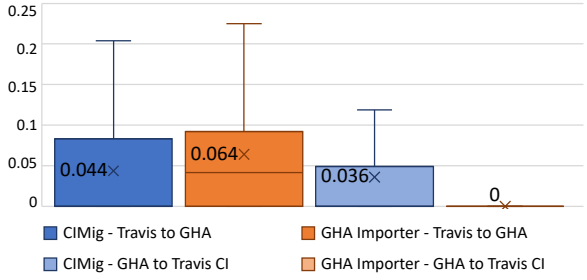


Figure 5.12: CrystalBLEU score of the Generated files.

Moving on to the translation quality, both Cosine Similarity and CrystalBLEU scores, illustrated within Figure 5.11 and Figure 5.12 respectively, and measured using the test set, display averages of 0.51 and 0.044 for the translation from Travis CI to GHA. The translation from GHA to Travis CI has averages of 0.35 and 0.036, respectively. Using the Paired T-test [269] for the Cosine Similarity and CrystalBLEU scores for the translations from Travis CI to GHA, we found that the differences between the two translation directions were statistically significant, with p-values of 0.0001 and 0.0098 (<0.05). Using the same statistical test for these scores for the translations from GHA to Travis CI, we found that the differences between the two translation directions were statistically significant as well, with p-values of 0.0001 and 0.0001 (<0.05).

Using the CrystalBLEU metric, it's clear that the generated equivalent GHA files and the equivalent Travis CI files show a good similarity to their developer-crafted baselines, especially when considering the works of Eghbali & Pradel. [82] For CrystalBLEU, where values around 0.05 were considered indicative of context-preservation and feature-parity between the code pairs. Furthermore, Using both Cosine Similarity and CrystalBLEU to compare the results from CIMig to those obtained by GitHub Actions Importer for the scenario of translating Travis CI files to GHA files, it's clear that CIMig's generated files

are as similar to the developer-provided files. This helps substantiate the quality of our generated files, especially since the official tool relies on hand-crafted specific rules. While CIMig supports the translation from GHA to Travis CI, GitHub Actions Importer does not, nor does any current tool, hence why there is no baseline for comparison in that case.

We find it important to mention that a considerable amount of GHA syntax does not have a Travis CI equivalent, mainly because the oft-used `Actions` in GHA do not have a direct equivalent in Travis CI, as it doesn't support reusable workflows, making their translation difficult, explaining the lower results obtained when translating GHA files to Travis CI. It's also notable that GHA-Travis CI equivalent set contains GHA and Travis CI tuples that have as little as 50% functionality in common, meaning that the generated equivalent GHA file or generated equivalent Travis CI file may only achieve a maximum similarity of 50%.

Overall, our results further support the confidence in the quality of the equivalent Target CI file generated by CIMig and validate that they implement a sizeable percentage of the functionality originally found in the Source CI file. We believe the files CIMig generates can form a good basis for developers to build on and help accelerate the migration process of their infrastructure, and we attempt to confirm this in the following section.

Table 5.2: User Study results on manual migration, and migrations with CIMig and with GHA Importer

Project name	First Workflow					Final Workflow					Avg. user rating	
	Manual	CIMig		GHA Imp.		Manual	CIMig		GHA Imp.		CIMig files	GHA Imp. files
	time (m)	time (m)	saved (%)	time (m)	saved (%)	time (m)	time (m)	saved (%)	time (m)	saved (%)		
WxJava	38.40	23.6	38.54	11.60	69.79	76.80	30.20	60.68	19.80	74.22	2.60	4.80
hutool	41.40	26.4	36.23	9.00	78.26	90.40	40.20	55.53	14.80	83.63	2.80	4.40
Elasticsearch-sql	29.00	24.20	16.55	4.60	84.14	46.00	36.20	21.30	8.00	82.61	3.40	5.00
Hsweb-framework	68.80	9.80	85.7	24.00	65.12	93.40	34.60	62.96	45.00	51.82	3.40	3.40
Telegram Bots	22.20	10.40	53.15	17.20	22.52	73.40	26.80	63.49	33.60	54.22	3.00	3.20
Average	39.96	18.88	46.05%	13.28	63.97%	76	33.6	52.79%	24.24	69.30%	3.04	4.16

5.6.1.2 User Study

Table 5.2 shows the results of the user study conducted following Section 5.5.1. Column 1 shows the name of projects used for the migration from Travis CI to GHA in the study. The First Workflow (column 2-6) shows the time that developers spent on Manual migration and migrations using CIMig and GitHub Actions Importer to reach a First passing workflow. For the results of CIMig and GitHub Actions Importer, we show how much time was saved in comparison to Manual migration (column 4 and column 6). Similarly, the Final Workflow (column 7-11) shows measures of developer time taken to reach a Final passing workflow for the same 3 migration types. Finally, Avg. User Rating (column 12-13) contains the average

usefulness scores from 1 to 5, assigned by the developers to the files from CIMig and GitHub Actions Importer.

Similar to prior research [52, 210, 358], the files generated by CIMig and GitHub Actions Importer still require some manual modifications before being usable, however, the results in Table 5.2 show the usefulness of these tools by quantifying how CIMig and GitHub Actions Importer help developers reach both the First-passing workflow and the Final-passing workflow much faster than manual migrations. Indeed, CIMig reduced the Manual migration time by 16% to 86%, and GitHub Actions Importer reduced it by 22% to 84% for reaching the First-passing workflow. We see similar reductions as well in the migration time for the Final-passing workflow. In terms of user ratings, CIMig has an average of 3.04 user rating, and GitHub Actions Importer has a higher average user rating of 4.16. When applying Repeated Measures ANOVA [106] to the Time to first build and Time to final build, we found that the differences between the two tools were statistically significant, with p-values of 0.02 and 0.00003 (<0.05). However, we found that the differences between the two tools in terms of user ratings were not statistically significant, with a p-value of 0.06 (>0.05), when using the paired T-test [269],.

Focusing on specific projects, we notice similar user ratings for the two tools for the `Hsweb-framework` and `Telegram Bots` projects. In both projects, CIMig also provides higher reduction in migration time than GitHub Actions Importer. Two developers mentioned in their reports that the files provided by CIMig were easier to extend for these 2 projects than GitHub Actions Importer's files, where GitHub Actions Importer's attempts to translate some syntax results in more complicated configuration files that were harder to debug and extend. The remaining developers confirmed this as well, thus explaining the reported user ratings and time savings. In summary, CIMig shows lower reduction rate than GitHub Actions Importer on three projects and higher reduction rate on two projects. User ratings tend to follow the reduction rates, with ratings for CIMig being lower than GitHub Actions Importer. Overall, the results confirm that the files generated by CIMig are usable in the GHA environment with minor modifications, and help save on migration time. Furthermore, CIMig is a technology agnostic approach that leverages mining processes from existing files, making it easy to extend and adapt to new syntax, as shown via GHA to Travis CI migration. On the other hand, GitHub Actions Importer is built using manual-mapped rules, and only supports Travis CI to GHA migration, limiting its extension to other scenarios. Hence, CIMig provides more usefulness in terms of supporting more migration scenarios with comparable performance to the specialized migration tool.

Finding 1:

Although CIMig uses a learning-based, technology-agnostic approach, it achieves performance comparable to the technology-specific hand-crafted GitHub Actions Importer in terms of migration quality, time savings, and user satisfaction.

5.6.2 RQ2: CIMig Execution Cost

Concerning the time needed to perform the translation of GHA files to Travis CI the average execution time of CIMig is 719.85 milliseconds, and the median is 705 milliseconds. Meanwhile, the average execution time of GitHub Actions Importer is 1553.31 milliseconds, and the median is 1503.38, for the same process. When applying the Paired T-test [269] to the execution times of the two tools, we found that the differences between the two tools were statistically significant, with p-values of 0.0001 (<0.05) for both translation directions.

While both executions times are acceptable [230], CIMig is faster than GitHub Actions Importer . CIMig has acceptable times for translating GHA syntax to Travis CI as well, with a median execution time of 797 milliseconds, and an average translation time of 1235.46 milliseconds.

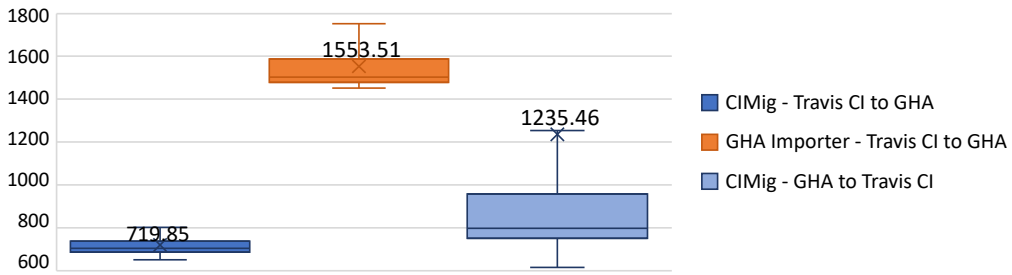


Figure 5.13: Execution time of GitHub Actions Importer and CIMig in milliseconds

Concerning the different processes of the training phase, they are only executed once and are independent of the translation process, making their time consumption less important. During the rule mining phase, detailed in Section 5.4.2.1, we executed 2 Apriori-based ARM operations: Travis CI to GHA translation rules, which took 45947 milliseconds to execute, and GHA hierarchization rules which took 22022 milliseconds to execute. These times were nearly identical when mining the translation rules GHA to Travis CI, as well as the generation of Travis CI hierarchization rules. The most time-consuming process we designed was the detection of which GHA and Travis CI Frequent Trees match with the Stat-Based rules, which took 1625773 milliseconds (around 27 minutes), despite a parallelized implementation that took advantage of all CPU threads available. This is however not surprising as there is

a total of 99586 Stat-Based rules for each direction, along with 524 Travis CI Frequent Trees and 2664 GHA Frequent Trees.

We performed two Frequent-Trees mining operations: one on Travis CI files, which took 1211342 milliseconds (around 20 minutes), and one on GitHub Actions files which took 257445 seconds (around 71 hours). The latter's much higher time consumption can be attributed to a bigger number of unique root nodes at which we attempted to detect Frequent-Trees, as well as the larger and more complex ASTs of GHA files. Overall, we believe CIMig time consumption during the training phase also remains within acceptable limits.

Finding 2:

CIMig is faster than the GitHub Actions Importer, requiring only about one second to generate translations in both directions.

5.6.3 RQ3: CIMig Translation Failures

Although our approach generates CI files of good quality, there are certain cases where our approach fails to generate an acceptable translation. These failures are classified into three categories as follows:

Syntax with no direct equivalent: (5 out of 25 Travis CI ⇒ GHA translations, 22 out of 25 GHA ⇒ Travis CI translations) Although there are some similarities between Travis CI and GHA configuration syntaxes, there are certain functionalities that are supported in only one of them. For example, GHA offers the `uses` keyword that allows reuse of existing GHA workflows in the form of Actions, but Travis CI does not offer an equivalent functionality. An example of this syntax is shown in Listing 1.

Syntax that relies on more than two levels: (7 out of 25 Travis CI ⇒ GHA translations, 2 out of 25 GHA ⇒ Travis CI translations) Since we opted to capture and translate H-2 ASTs in Travis CI, any functionalities that depend on the configuration of more than 2 levels are not captured. For example, the usage of multiple stages with different `jdk` and `language` settings in Travis.

Unabstracted syntax and parsing issues: (23 out of 25 Travis CI ⇒ GHA translations, 4 out of 25 GHA ⇒ Travis CI translations) Since the abstraction process was applied with the usage of the most common commands, some less common commands, such as `openssl` and `jarsigner`, are not represented within the translation rules we generated. An illustration of this is shown in Listing 2. This can be minimized by extending the abstraction process to include more commands, as discussed in the last paragraph of Section 5.4.1.

Listing 5.1: GHA syntax with no Travis CI equivalent from `Albertus82/Cyclesmod`

```
uses: sonarsource/sonarcloud-github-action
env:
    GITHUB_TOKEN: ${{ secrets.
        GITHUB_TOKEN }}
    SONAR_TOKEN: ${{ secrets.
        SONAR_TOKEN }}
    SONAR_SCANNER_OPTS:
-Dsonar.organization=albertus82-github
```

Listing 5.2: Travis CI syntax from `Gotify/Android` using unmatched commands (Highlighted)

```
before_deploy:
    |\colorbox{yellow}{openssl}| aes-256-cbc
        -K $encrypted_key
        -iv $encrypted_iv -in release-key.
            jks.enc
        -out gotify-release-key.jks -d
    ...
    |\colorbox{yellow}{jarsigner}| -verbose -
        sigalg SHA1withRSA
        -digestalg SHA1 -keystore release-
            key.jks
```

Finding 3:

CIMig has a few limitations in translating certain syntaxes, such as those with no direct equivalent, those that rely on more than two levels, and those that have not been covered by the abstraction process.

5.7 Related works

Automatic Code Migration. Migrating from one programming language to another is very common in large software systems due to the need for cross-platform support and language support features. However, programming language migration is effort-intensive and error-prone [358, 210, 52] due to the differences in syntax and unfamiliarity with the target programming language. To mitigate this, researchers developed tools and techniques for automatic programming language migration. For example, Java2CSharp [88] and j2swift [231] are developed for migrating from Java to C# and Swift. However, these tools and other research works [83, 219, 141] use predefined transformation rules for their migration. Creating these rules is a laborious process, and in many cases, these migrations may fail due to complex and rare syntax used by different programming languages. To resolve these limitations, Zhong et al. [358] and Nguyen et al. [227] utilized a mining-based approach for automatic migration. These approaches heavily relied on similarity-based alignment and may not correctly migrate code if the target language adopts a different naming scheme. mppSMT by Nguyen et al. [228], utilizes a divide-and-conquer approach with a phrase-based SMT engine to integrate the semantic features for automatic migration. The approach uses data and control dependency of source code, which may not be applicable to configuration code due

to its higher level of abstraction. j2sInferer [15] is a recent approach that utilizes syntax and mapping rules with minimal domain logic for migration of Android Java code to Swift code with 65% cross-project accuracy. Such syntax similarity is very low among configuration code files and makes alignment infeasible. More recently, ML-based techniques [53, 134] are proposed for the automatic migration of programming languages. However, ML approaches require large corpora for model training, which may not be feasible for recently developed programming languages or DevOps configuration files where very little migration data exists.

Configuration Maintenance. Like source code files, the different configuration code files for CI systems, Build systems, etc., are integral parts of software projects. Prior works suggested that developers often work on maintaining and migrating configuration systems [333, 353, 276, 124] to improve performance and productivity. However, maintaining configuration code is tedious due to limited domain-specific knowledge and syntactical differences in configuration code across different tools. Gligoric et al. [124] utilized dynamic analysis and search-based refactoring techniques to automatically migrate build systems. Moreover, automated program repair-based techniques [143, 190, 220] are applied to fix build scripts. Xue et al. [339] proposed a technique for automatic migration to Docker containers. Recently, Henkel et al. [145] proposed binnacle to automatically detect bad practices in Docker files. Vassallo et al. [321] utilized program analysis techniques to detect anti-patterns in CI configuration scripts. At the same time, Rahman & Parmin [260] proposed a technique for automatically detecting security vulnerabilities in Puppet-based IaC configurations. Although there are several techniques for the automatic migration and maintenance of different configuration systems, there is no research work on the automatic migration of CI systems.

5.8 Threats to Validity

Internal Validity. The main threat is the incorrect composition of the generated CI configuration code. To mitigate this, we tested our approach thoroughly in several rounds, and we contextualized our results by comparing them to both developer-crafted and GitHub Actions Importer-generated files. We also evaluated the generated files with state-of-the-art metrics to evaluate the correctness of the approach, and further evaluated them via the user study.

External Validity. We evaluated our approach for migration between Travis CI and GitHub Actions. These projects are Java-based and OSS in nature. So, our approach may not work correctly on other CI systems with different programming languages and closed-source projects. Mining rules from projects with a mix of Programming Languages (PLs) resulted in lower quality rules that had worse support for projects using each PL, even though the rules supported multiple PLs. Hence we opted Java as a single PL for our study due its

popularity. Although the evaluation is CI system-specific, the proposed rule mining and composition techniques are more generic. Moreover, different CI systems support similar functionalities and similar structures, such as YAML. So, we believe that our proposed approach will work for other CI systems as well, with sufficient retraining. We attempted to approximate actual user experience via our user study by recruiting developers with varied development and CI experiences, but their experiences may not reflect every possible users'.
Construct Validity. For automatic rule generation, we considered two-level (H-2) level AST transition nodes. We believe these rules are a good balance between conservativeness and diversity, for reasons detailed in the Parameter Tuning paragraph of 5.4.2.3.

5.9 Conclusion and Future work

With the growing use of CI systems for faster code integration, migration of CI systems has become very common in development activity. However, migrating CI systems is a tedious and error-prone process [272]. We presented CIMig To assist the developers with CI migration, and help facilitate this process. In our evaluation, even with a small set of existing CI migration data, CIMig can generate CI files with a per-file success rate of 70.82% for GHA, 51.86% success rate for Travis CI, and these files have a good similarity to the developer-crafted versions. Furthermore, the user study also suggests that CIMig is beneficial for developers, allowing them to migrate CI systems in less time than manual migration. Moreover, the proposed approach is technology-agnostic in nature and can be easily applied to other configuration systems as well with the preparation of the appropriate learning sets. In the future, we plan to incorporate large language models (LLM), such as ChatGPT, to generate more accurate migration rules and apply the automatic migration process to other configuration systems, such as Docker, etc.

CHAPTER 6

PromptDoctor: Automated Prompt Linting and Repair

This work is currently under submission in FSE 2025, the 30th ACM SIGSOFT International Symposium on the Foundations of Software Engineering.

6.1 Introduction

The rise of large language models (LLMs) has rapidly transformed modern software development, with these models becoming integral components in application logic [328, 140]. Many systems now rely on structured prompts to interact with these black-box models by mixing natural language with dynamic runtime values. These structured prompts, termed Developer Prompts (Dev Prompts), are fundamentally different from traditional software artifacts, requiring specialized analysis tools that go beyond classical prompt analysis methods [235, 249]. As Dev Prompts combine natural language and programmatic elements, they introduce new challenges, such as biases, vulnerabilities, and sub-optimal performance that can impact overall system performance and reliability.

Existing research [137, 66, 250, 326, 311] on prompts focused mainly on conversational prompts. Specifically, Guo et al. [137] and Clemmer et al. [66] proposed techniques of reducing biased responses to conversational prompts by fine-tuning LLMs. Also noteworthy are the works of Wang et al. [326] and Pryzant et al. [250], which focused on optimizing the performance of conversational prompts by hand-crafting and evaluating prompt engineering practices and applying computational modifications to LLMs, respectively. However, none of these works focused on Dev Prompts: natural language prompts embedded in source code. Dev Prompts represent a relatively new class of software artifacts, and their rapid proliferation introduces distinct challenges in software engineering. Unlike traditional code, Dev Prompts are primarily composed of natural language, making them susceptible to a

variety of issues, including bias, vulnerability to injection attacks, and performance sub-optimal performance. The rise of Dev Prompts has even led to the emergence of a new role within software development: the "Prompt Engineer." Therefore, to address these recent and rapid changes in the software engineering landscape, our work adopts the following high-level goal:

Goal

Develop tools and techniques to detect, analyze, and improve Dev Prompts, addressing issues like bias [137], vulnerability [189, 342], and sub-optimal performance [250], to support developers and prompt engineers in creating more reliable interactions with Large Language Models.

Dev Prompts introduce challenges distinct from traditional code due to the vagueness and ambiguity inherent in natural language. This makes them prone to bias [137], injection attacks [189], and sub-optimal performance [296]. Additionally, the integration of natural language with traditional code presents a largely uncharted area for automated analysis. In this work, we examine Dev Prompts in their operational contexts, focusing on how dynamic value interpolation and runtime factors influence their reliability, building on the dataset of Dev Prompts collected from PromptSet [249]. We now focus on three primary challenges with Dev Prompts—bias, vulnerability, and performance—each posing a significant risk in software development. Let's examine them in more detail:

1. **Bias**: Dev Prompts must be carefully crafted to avoid both explicit and implicit biases, as even subtle wording can have a substantial impact. One common prompt design strategy is to provide the model with a "persona" to better perform a task. However, it is easy for biases to be unintentionally encoded into these personas, which can influence the model's behavior in undesirable ways, such as propagating bias and potentially creating societal harm.
2. **Vulnerability**: Dynamic variable interpolation in Dev Prompts makes them vulnerable to prompt injection attacks. Understanding how to properly sanitize user input before integrating it into prompts is an ongoing challenge. Without careful input validation, many software systems risk exposing too much control to users, making them susceptible to malicious prompt manipulation that may lead to a leak of sensitive information, for example, thus potentially causing customer harm.
3. **Sub-optimal Performance**: Crafting effective prompts is often seen as more of an art than a science, which means many existing Dev Prompts are likely under-performing and can benefit from optimization. Performance in this context is task-specific and not directly related to the software's runtime but rather to the accuracy and relevance

of the model's output for the given task. Relying on sub-optimal prompts can cause customer users to lose trust in the products that rely on them, potentially causing product reputation harm.

To highlight these issues, we provide an illustrative example in Figure 6.1 from the GitHub project *blob42/Instrukt*. At first glance, the prompt seems innocuous, yet it suffers from all three challenges outlined above. By using the typically female name "Vivian," the prompt risks biasing the model toward generating stereotypically female-coded responses. Empirically, we also found that this prompt is vulnerable to prompt injection attacks via the variable `context` and that using strong imperative commands, such as "you MUST ..." leads to a ~20% gain in prompt adherence on a synthetic dataset designed for this prompt. In any case, Figure 6.1 shows just how easy it is for Dev Prompts to have many issues—issues that require new tools and new research to identify and fix.

Example of a Flawed Prompt

```
You are Pr.  Vivian.  Your style is conversational, and you always aim to get straight to the point.
Use the following pieces of context to answer the users question.  If you don't know the answer,
just say that you don't know, don't try to make up an answer.  Format the answers in a structured
way using markdown.  Include snippets from the context to illustrate your points.  Always answer from
the perspective of being Pr.  Vivian.
----------------
{context}
```

Figure 6.1: Example of a prompt with bias and injection issues from GitHub project: *blob42/Instrukt*

Within this research work, we analyzed how widespread these three issues are in Open Source Software (OSS) Dev Prompts, and we offer an easy-to-use solution for mitigating them. Using empirically validated LLM-powered processes, we perform a large-scale analysis on PromptSet [249], a collection of Dev Prompts mined from open-source projects, to detect the prevalence of bias and injection vulnerability. We also perform an in-depth analysis to examine a varied set of Dev Prompts of different task categories to illustrate the prevalence of sub-optimality. Then, we propose `PromptDoctor`, a bespoke solution to address these issues within Dev Prompts. `PromptDoctor` utilizes automatic issue discovery and prompt rewriting processes that build on a generation-evaluation paradigm to automatically correct flaws within Dev Prompts, and is both technology and LLM independent. Indeed, `PromptDoctor` is easily integrated in development processes, allowing developers to detect and correct the aforementioned issues in Dev Prompts, all while avoiding the expensive process of Model Fine-Tuning altogether. `PromptDoctor` analyzes Dev Prompts before their deployment and automatically mitigates any issues they may contain, thus preemptively

avoiding any potential harms these issues can create. Finally, we evaluate `PromptDoctor` and report its results when used on the flawed Dev Prompts we identified. We achieve these goals by answering the following research questions:

- **RQ1:** How widespread are Bias and Bias-proneness in Dev Prompts? How effectively can we minimize these issues?
- **RQ2:** How widespread is Vulnerability to injection attacks in Dev Prompts? How effectively can we harden Dev Prompts against them?
- **RQ3:** How widespread is sub-optimality of prompts in Dev Prompts? How effectively can we optimize Dev Prompts' performance?

To answer these questions, we design and evaluate various LLM-powered processes that form the foundation of `PromptDoctor`. `PromptDoctor` allowed us to establish that 3.46% of Dev Prompts are prone to generating biased responses, and that 10.75% are vulnerable to prompt-injection attacks. We also found that 36% of Question-Answering Dev Prompts quantitatively under-performed when tested against real-world benchmarks.

Using `PromptDoctor`, we were able to improve a portion of these flawed Dev Prompts by de-biasing 68.29% of the biased Dev Prompts, hardening 41.81% of vulnerable Dev Prompts, and optimizing 37.1% of the sub-optimal Dev Prompts.

Our contributions are:

- The first large-scale analysis of OSS Dev Prompts, uncovering **bias**, **vulnerability**, and **sub-optimality** in them.
- Creation and Evaluation of `PromptDoctor`, a novel solution to detect and fix **bias**, **vulnerability**, and **sub-optimality** in Dev Prompts, made available as a VS Code extension.
- Our empirical findings and observations on Dev Prompts have uncovered a new area that could inspire and guide future research directions.

6.2 Background

6.2.1 Large Language Models and Dev Prompts

Large Language Models (LLMs) have emerged as a transformative advancement in natural language processing [38, 199]. Unlike earlier models, LLMs can scale to billions of parameters and vast volumes of training data [241, 8], which has led to emergent capabilities, such as in-context learning [285, 71], The output of a language model is guided by the input context, or prompt. Prompts can take various forms: questions, statements, multi-turn dialogues, etc. However, in this work, we focus on prompts written by developers and used within software applications, which we refer to as *Developer Prompts* [249] or *Dev Prompts* for short. These

prompts operate in more constrained contexts, typically embedded within specific methods to generate a targeted output. Dev Prompts are generally not directly visible to or editable by the user; instead, the user can only influence them indirectly by setting variables or parameters that are interpolated into the Dev Prompt. A notable exception is prompt playgrounds or model comparison tools, which may allow users to specify the entire prompt. In addition, Dev Prompts are usually used in one-off interactions with the model rather than being part of a multi-turn dialogue. Furthermore, the context in which Dev Prompts are used is often domain-specific, such as text summarization or translation based on user input. An example of a Dev Prompt in source code is shown in Listing 6.1. The proliferation of LLMs has driven their integration into traditional software systems via Dev Prompts, offering impressive capabilities but also introducing new challenges. Addressing the challenges posed by these "hybrid" software systems—those combining traditional software logic with LLM-powered components—will require the development of new tools and strategies.

Listing 6.1: Example of a Dev Prompt from `ownsupernoob2/Blimp-Academy-Flask`

```
def product_observation(prompt_product_desc):
    response = openai.Completion.create(
        model="text-davinci-002",
        prompt="The following is a conversation with an AI Customer Segment Recommender
         ....
        AI, please state a insightful observation about " + prompt_product_desc + ".",
        temperature=0.9, max_tokens=...)
    return response['choices'][0]['text']
```

6.2.2 Bias in Language Models

Like other Machine Learning models, LLMs can learn biases from the data they are trained or fine-tuned on. These biases can have a cascading effect on the software that uses these models. For example, a bank loaning software that used an ML model to determine credit-worthiness was found to be biased against people of certain intersectional groups [175], even though protected attributes such as race and gender was not exposed to the model. In the context of LLMs, bias and potential for bias stubbornly persist [43, 101, 55]. Furthermore, LLMs risk further propagating these biases and stereotypes [239, 151], in ways that can be hard to directly see or detect beforehand. For example, as discussed by Cheng et al. [55], LLMs are more likely to make assumptions about people of a certain gender and race, even when the prompts fed into the model do not contain any information that would lead to these assumptions. These assumptions and biases not only risk generating biased responses, but also risk causing harm to the people who interact with the software that uses these

models, by altering their internal behavior and decision-making processes, all while providing little to no transparency about their reasoning to the end-user, such as the case of the creditworthiness model. Hence, it is imperative to detect and fix bias in Dev Prompts to avoid causing Societal harm.

6.2.3 Vulnerability in Language Models

Dev Prompts present a new attack vector in software applications. Prompt-injection [271] is a novel technique where attackers inject specific phrases into a prompt to cause the LLM to behave in ways against its design intentions as set by the software's developers. Prompt-injection attacks can trigger failures such as unwanted responses that misuse company's resources, such as misusing a company customer support bot to get free access to a paid LLM service. In more serious scenarios, it can lead to the exposure of sensitive information, where it can coax the LLM to reveal confidential information about the company's inner machinations, such as private email addresses [331], or even the physical location of company resources. These attacks can also coax intellectual property contained within the prompt by sharing its internal text, which makes further misuse even easier [156]. The pace of the development of Prompt-injection attacks [271, 188, 187, 346, 133] is indicative of the growing threat they pose to software applications that use LLMs. Thus, it is important to detect and fix vulnerabilities in Dev Prompts to avoid causing Customer harm.

6.2.4 Performance of Language Models

While LLMs can perform surprisingly well in certain tasks, such as text generation and text summarization, their performance remains mediocre at tasks such as math and logic [284, 287]. Further complicating matters, there is no standardized automatic way to evaluate or optimize Prompts. Prompt engineering is an emerging field that attempts to address these shortcomings by providing a systematic way to write prompts [332, 11]. There are many competing practices such as Chain-of-Thought (COT), where the prompt asks the LLM to explain its reasoning step by step; and "N-Shot" Prompting, where the prompt contains example input-output pairs, and instruction alignment where the prompt contains rules for the model to follow. However, prompt engineering remains a largely manual task with no quantitative guidelines to follow. Thus, there is a need for the automatic evaluation and optimization of Dev Prompts to ensure good performance of the systems that rely on them, and avoid Product-Reputation harm.

6.3 Research Approach

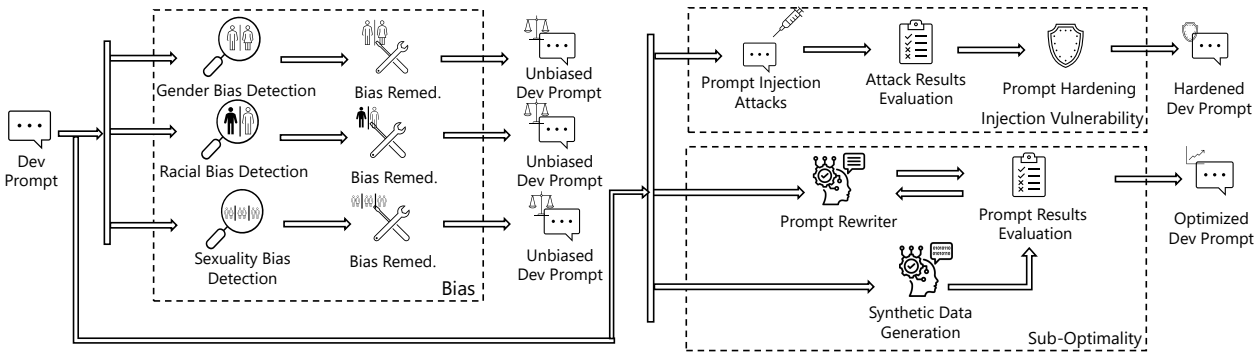


Figure 6.2: Overview of the Research Approach

6.3.1 Data Preparation

6.3.1.1 Dataset Selection

As discussed in Section 6.2.1, we focus within this work on Dev Prompts sourced from PromptSet [249] which contains 61,448 unique Dev Prompts collected from 20,598 OSS projects. However, no examination of the quality of these prompts or significant cleaning operations were performed during PromptSet's creation. Indeed, upon manual inspection, we found that it contained a number of toy prompts that are not representative of "production-quality" Dev Prompts. We tackle cleaning this dataset in Section 6.3.1.2.

In addition, Dev Prompts are not just static strings of text: they interweave structured natural language with traditional software languages. Many contain variables that are added at interpolated before being sent to the LLM. Since Dev Prompts are commonly represented as variable(s) in source code, they do not have a standardized representation. Values might be interpolated or concatenated dynamically depending on the control flow of the program. For example, a Dev Prompt might take the form of `"this is {x}."` or `"this is "+x+"."`, and these representations are stored exactly as they are extracted from source code in PromptSet. We refer to the variables in Dev Prompts as *Prompt Holes*, and as their values are generally defined at runtime via user input, it's difficult to determine what they should be within the context of the different analyses we perform within this work. We attempt to address both of these issues via the processes detailed in Section 6.3.1.3.

6.3.1.2 Dataset cleaning

While PromptSet contains a large and diverse set Dev Prompts, it was important to perform some data cleaning processes to maximize high-quality prompts in our experimental set. We identified that 25% of the Dev Prompts PromptSet contained had a length of 31 characters or less. Upon manual inspection, we found most were out of distorted or helper-prompts of little significance, hence we eliminated them from the set of prompts we analyzed. 45,747 Dev Prompts remained. We also removed non-English prompts, further eliminating 5,174 (11.31%) prompts which contain non-ASCII and non-Emoji characters. Future research could focus on these Dev Prompts to determine if the approaches proposed within this work can be applied to prompts in other languages as well. 40,573 prompts remained after this process.

6.3.1.3 Prompt Parsing

Prompt Canonicalization Via this process, we standardized the presentation of the different Dev Prompts into a universal, canonical, representation that was easier to programmatically parse and process. This process relies on static parsing and regex matching to determine the different variable holes within a Dev Prompt. The resulting canonical representations preserves the original Dev Prompt's text along with the holes inter-weaved within. The prompt holes are delineated within special characters ({ and }). An example of the process of canonicalization is shown in Figure 6.3.

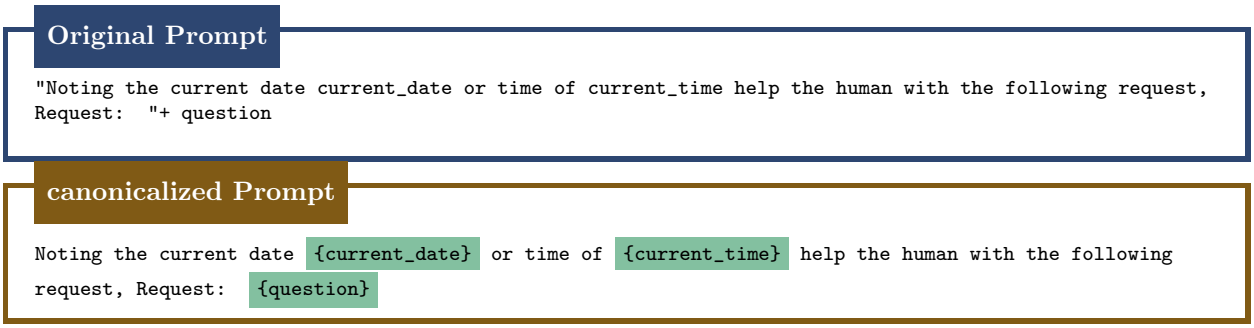


Figure 6.3: A prompt from `zekis/bot_journal`, before and after canonicalization

Prompt Patching After standardizing the representation of Dev Prompts, we set out to create appropriate mock values for their different holes in order to ground our consecutive analyses in realistic usage scenarios of Dev Prompts. Due to their aptitude in generative tasks, we utilize LLMs to generate these values. However, the scope from which we can determine the appropriate values for the different prompt holes is not immediately clear, as

the containing method or class may not always contain informative comments or names, and a ReadMe file may be too high-level to be relevant for the value of a single Prompt Hole. Context window limitations further complicating matters, as a single class, method, or file may exceed the window size in some cases. Hence, in order to generate mock values for these variables, a process we refer to as *Patching* the prompt, we rely on the Dev Prompt's text and the variable name corresponding to the Prompt Hole, thus making this process localized and easily transferable to other contexts. To this end, we hand-crafted a prompt following the practices discussed by Sahoo et al. [280], and send it to the LLM to generate mock values for each Prompt Hole. This handcrafted prompt is given at [18].

In the case of a prompt containing multiple holes, there were two possible approaches: either patching the holes in parallel in an independent manner or patching them sequentially in a dependent manner. In parallel patching, the mock value for each hole is generated independently of the other prompt hole values, which is formalized in Equation 6.2. In sequential patching, the mock value generated for prompt hole x is dependent on the values generated for all the prompt holes that appear before it in the Dev Prompt, this is formalized in Equation 6.2. We validated that generating values for the variables sequentially in their order of appearance, patched prompt holes in a more consistent and logical fashion than via parallel generation.

$$val(h_x|patch(prompt,[val(h_x-1),val(h_x-2),...val(h_1)])) \tag{6.1}$$

$$val(h_x|prompt),val(h_x-1|prompt),val(h_x-2|prompt),...val(h_1|prompt) \tag{6.2}$$

For the sake of simplicity and cost efficiency, during bias and vulnerability detection and remediation, we restrict our patching process to generally only one value for each Prompt Hole, as generating multiple values can lead to an exponential number of combinations possible. An example of a canonicalized prompt and its corresponding generated values via the process of patching is shown in Figure 6.4. Furthermore, we add multiple prompt-level and code-level mechanisms to ensure that this process does not introduce bias or bias-proneness in prompts.

Optimization Dataset Synthesis For the purpose of Dev Prompt optimization, we extend the patching process to create synthetic datasets, which serve as input values assigned to the different prompt holes of the Dev Prompt we aim to optimize. A synthetic dataset is comprised of multiple values for each Prompt Hole, values that are generated by the patching process are optimized for creativity and diversity to mitigate duplication as the quantity of patches per Dev Prompt increases. In practice, these patches are generated in a sequen-

canonicalized Prompt

```
Noting the current date {PLACEHOLDER_1} or time of {PLACEHOLDER_2} help the human with the following
request, Request: {PLACEHOLDER_3}
```

Generated Values

```
{PLACEHOLDER_1} -> today
{PLACEHOLDER_2} -> 3:00 PM
{PLACEHOLDER_3} -> what are my upcoming meetings for the rest of the day?
```

Figure 6.4: Example of Prompt Patching with Dev Prompt taken from `zekis/bot_journal`

tial manner, where we use stratified temperatures during patch generation and include a few randomly-selected previously-generated patches for the same hole as examples to avoid duplication during the generation process. Within our hand-crafted prompt also includes guidelines as a context and we enforce a strict response format to ensure the data generated will conform to the original Dev Prompt's structure [183].

6.3.1.4 Dataset Sampling

After applying the cleaning process in Section 6.3.1.2, we obtained a set of 40,573 Dev Prompts. The size of the set was still too computationally and financially expensive to use for all our analyses, especially as each technique relies on multiple LLM calls per Dev Prompt. Hence, we set out to select a representative sample from this larger set. After applying the canonicalization process from Section 6.3.1.3, we extracted the number of prompt holes within the different Dev Prompts. We found that 20,620; 9,427; 6,154; 2,204; 1,503; 464; and 651 Dev Prompts had respectively 0; 1; 2; 3; 4; 5; and 6+ prompt holes. We grouped Dev Prompts with more than 5 holes into one large group after finding a significant drop in the numbers for Dev Prompts with exactly 6 holes or exactly 7 holes in comparison to Dev Prompts with exactly 5 holes, where they were 63.44% and 82.02% smaller, respectively.

From these results, prompt holes emerged as an appropriate stratification criteria to help guide our process of selecting a representative population of Dev Prompts. Hence, we applied the corresponding random sampling from each strata with a 95% confidence - 5% error. After applying this process, we created a set comprised of 378, 370, 362, 328, 282, 211, and 242 Dev Prompts with respectively 0, 1, 2, 3, 4, 5, and 6+ prompt holes, which we use for the rest of our analyses.

Optimization Dataset When optimizing Dev Prompts, we shift our focus to categorizing the Dev Prompt into specific task categories. These four task categories are: question & answer, grammar correction, summarization, and translation. We refer to the grammar correction, summarization and translation Dev Prompts as Grounded task Dev Prompts. Question & answer (QA) prompts are open-ended requests of a language model, such as the one shown in Figure 6.1. We filter PromptSet for short, English-language Dev Prompts between 20 and 200 characters long. Additionally we use a mood filter from SpaCy to select only imperative or interrogative prompts [152]. This process extracted 3,310 QA prompts for us to optimize. From these, we randomly sample 100 to conduct experiments.

For grammar correction, we filter by searching for keywords related to grammar and punctuation. Then, we augment this set by adding Dev Prompts which are semantically similar to the Dev Prompts which had keyword matches. This resulted in a dataset of 36 grammar correction Dev Prompts. Using similar techniques, We also find a handful of 7 translation and 4 summarization Dev Prompts.

6.3.2 Addressing Bias

6.3.2.1 Bias Detection

The issue of detecting biases within Natural language text remains a fraught and complicated issue. Biases can come in many shapes and forms, and can stem from multiple factors. The approach we designed to detect biases is generic, and we use it to focus on three Biases that have been documented extensively within existing literature around Bias in software: Gender-Bias [216, 266, 324, 138, 309], Race-Bias [175, 41, 283, 234], and Sexuality-Bias [198, 304, 89]. We believe our approach can easily be extended to detect other types of biases as well, but we leave this exploration to future research works.

Our approach is simple but effective, as discussed within Section 3.2, LLMs are a great tool for linguistic and textual analyses, hence we leverage them via a hand-crafted prompt, available in our replication package [18], that specifies the type of bias we're attempting to detect along with a patched version of the prompt we're evaluating. This prompt generates a JSON file, making it easy to programmatically ingest its results. This JSON contains three fields: whether the Dev Prompt is Explicitly Biased, whether the Dev Prompt is prone to generating biased responses, and an explanation behind the evaluation given. We distinguish between explicit bias and bias-proneness since, as documented by previous research [55], Dev Prompts without explicit bias can still cause LLMs to generate biased-responses.

Following the recommendations of Radford et al. [257] and Brown et al. [44], we designed a Zero-Shot, One-Shot, and Multi-Shot, and measured their performance against established

benchmarks for the bias-detection as detailed in Section 6.4.1.1, and found that the Multi-shot version performed best. Hence, each bias-detection prompt also includes three example inputs: one explicitly-biased, one bias-prone, and one non-biased example. Each example was accompanied with an example expected JSON response. For added transparency, `PromptDoctor` reports to users the results of this evaluation along process including the explanation behind the evaluation.

6.3.2.2 Bias Remediation

While there are many recommendations regarding methods to rewrite prompts to improve their performance [340, 91, 4, 250], there is no established generic method to rewrite prompts to de-Bias them. Hence, we created an automatic de-Biasing method within `PromptDoctor`, that relies on a prompt generation-evaluation loop. Due to the prowess of LLMs when in text-generation, we also rely on their assistance during this process.

First, we evaluate the original prompt given by the developer via the method detailed in Section 6.3.2.1. Second, if the prompt is determined to be biased or bias-prone, we generate 5 rewrites of the developer-prompt with the goal of minimizing bias and bias-proneness, via a hand-crafted prompt, available in our replication package [18]. Third, we evaluate each of these rewrites for bias. If we determine some of them is also biased or bias-prone, we isolate them and run the same generation-evaluation loop on each of them. We stop this process when we have at least five new non-biased and non-bias-prone Dev Prompts. We then supply these rewrites sorted by distance[1] to the developer. We give developers multiple variants to promote higher flexibility for improved tool adoption [70]. We limit this process to 10 iterations in order to avoid running indefinitely and generating Dev Prompts that are too different from the original Dev Prompts.

6.3.3 Addressing Injection Vulnerability

6.3.3.1 Vulnerability Detection

There is a variety of Prompt Injection attacks possible when interacting with LLMs [271, 188, 187, 346, 133], however, as discussed within Section 6.2.1, interacting with them via Dev Prompts is generally more constrained than other scenarios. Dev Prompts deployed within a project are generally not directly accessible or editable, and are sent to an LLM after variables within these prompts are interpolated via values shaped by by users. Hence, we focus on injection attacks that are deployed via inserting a malicious string within specific location(s) of the prompt.

[1]Distance = Number of iteration that generated the new Dev Prompt

Testing a prompt for injection vulnerability relies on a collection of 42 known attacks from a corporate dataset[2] and the open web. Each of these attacks is formulated to produce a specific un-common target string as expected results. For each Dev Prompt we examined, we first perform the canonicalization process discussed in Section 6.3.1.3, and then for each Prompt Hole, we inject a specific attack into it and patch the remaining holes via the prompt patching process, detailed in Section 6.3.1.3. We then send the Dev Prompt to the LLM, and inspect the corresponding response to determine whether the attack was successful by scanning for the expected target responses. We repeat this process for all the attacks we have in our collection and all of the prompt holes within the prompt. Since these attacks were independent of each other, we performed this process in a parallelized manner, and the equation in Equation 6.3 shows a formalization of this process. It's notable that `PromptDoctor` reports which holes that served as successful injection points for these attacks to end-users.

$$Vulnerability_{prompt} = \sum_{i=1}^{n} attack_i(h_{pos}, \sum_{j=1\, j \neq pos}^{m} patch(prompt, hole_j)) \tag{6.3}$$

6.3.3.2 Vulnerability Remediation

Similar to Section 6.3.2.2, there is no established way to harden'' prompts against prompt-injection. Instead, most existing methods focus on Model Fine-Tuning [306, 188, 343] to defend against to these attacks. Furthermore, some systems add other layers of security to preemptively detect attack strings or compromised responses and then respond to those attacks with a generic denial response [245]. As discussed in Section 3.2, these approaches come with their own costs and caveats, hence, we implement a new Prompt hardening process within `PromptDoctor` as a less-expensive and easier-to-deploy complement to these approaches. Similar to Section 6.3.2.2, this process also relies on a generation-evaluation loop. First, we analyze a Dev Prompt to determine if it's vulnerable as described in Section 6.3.3.1. Second, if the Dev Prompt is vulnerable, we ask the LLM via a hand-crafted prompt, given at [18], to generate five new prompts that are a rewrite of the original prompt, and which are hardened against a specific attack. Third, for each generated Dev Prompt, we verify that it contains the same prompt holes as the original Dev Prompt, and then evaluate it for vulnerability it as described in Section 6.3.3.1. Finally, if all of the attacks in our set fail against one of these new Dev Prompts, we consider this Dev Prompt hardened, else, we add it to our list of vulnerable Dev Prompts for future generation-evaluation loop executions. Empirically, we found that generating a hardened Dev Prompt requires multiple iterations and an associated high cost, hence, we limit this process to 10 iterations, and we stop when

[2]Double blind policy forbids from being specific.

we have one new hardened Dev Prompt. Furthermore, we found that Dev Prompts with the lowest number of vulnerable holes were more likely to be successfully hardened, hence we sort the vulnerable Dev Prompts by the number of vulnerable holes before starting a new iteration of the hardening process

6.3.4 Addressing Sub-Optimality

6.3.4.1 Sub-optimality Detection

By default, we assume all Dev Prompts are sub-optimal and present an opportunity for possible automatic improvement, especially since previous research [349] found that many people struggle with prompting. Indeed, even a prompt writing expert cannot empirically prove their design has reached an optimal state. In addition, any evaluation processes may be hindered by the lack of a dataset. Therefore, we design `PromptDoctor` to automatically optimizes a Dev Prompt against a synthetic dataset to improve performance without requiring costly manual exploration or data collection by the developer.

6.3.4.2 Optimization Process

The `PromptDoctor` optimization process is summarized as follows: 1. Generation of synthetic training and test datasets based on the Dev Prompt, 2. Creation of a few seed Dev Prompts based on good prompting strategies written by OpenAI [240] and Anthropic [19], 3. Evaluation of all the generated Dev Prompts on the synthetic data, 4. Execution of a self-improving optimization algorithm to discover new Dev Prompts as described in Equation 6.4, 5. Repetition of steps 3-4 until no performance gain is observed on the training dataset.

Seed Dev Prompts Generation Self-optimization strategies are sensitive to their starting seed [340]. Hence, to mitigate falling into local minima, we propose multiple strategies to diversify the initial seed configuration for optimization. Primarily, we generate seed Dev Prompts based on prompt rewriting principles provided by OpenAI and Anthropic. The generative meta-prompt receives a sample of five different Dev Prompt principles, such as "Use Delimiters" or "Add to your prompt the following phrase: ‘Ensure your answer is unbiased and does not rely on stereotypes’", out of 26 principles detailed at [18]. Additionally, we vary the generative temperature to induce both precise and creative responses. We validate that the resulting generated Dev Prompts are well formatted and have the same prompt holes as the initial Dev Prompt.

Optimization We formalize the optimization algorithm as follows:

$$Opt_i = M_1(t_1, s_1, s_2, ..., s_n); \quad s_j = \sum_k f(., M_2(t_2, p_j, D_k))/K \tag{6.4}$$

where i is the step count, M a language model, t a meta-prompt, and s_j the tuple of a Dev Prompt p_j and its average score received across a dataset D of size K. The $i = 0$ step is a special case, where p_j is sourced from the seed Dev Prompts. Subsequent steps utilize the highest scoring n Dev Prompts generated across all steps of the algorithm. Finally, f represents a task-based evaluator as described in the following paragraph. M_1 specifically is used to generate new candidate Dev Prompts, M_2 generates responses based on candidate Dev Prompts and elements from the synthetic training dataset.

Evaluation After the seed Dev Prompts are generated and after each optimization step, we evaluate the quality of newly generated Dev Prompts on the synthetic training dataset. We select a scoring criteria based on the categorization of the initial Dev Prompt.
Translation. We use the BLEU metric to compare the quality of the generated translation with the reference synthetic translation [244].

$$s_j = \sum_k BLEU(D_k["translation"], M_2(p_j, D_k["source"])) \tag{6.5}$$

Summarization. We compare the semantic similarity of the generated summary with the reference summary using the cosine similarity of the embedded representations [263]. Using embedding function E:

$$s_j = \sum_k cos_sim(E(D_k)["summary"], E(M_2(p_j, D_k["source"]))) \tag{6.6}$$

Error Correction. We use the GLEU metric to compare the quality of the generated phrase with the reference synthetic phrase [222].

$$s_j = \sum_k GLEU(D_k["correct"], M_2(p_j, D_k["source"])) \tag{6.7}$$

QA Refinement. We utilize the "LLM as a judge" technique to score QA tasks due to their highly varied nature [357]. As a preprocessing step, each QA Dev Prompt generates a corresponding scoring prompt t_3 which will be used to evaluate the quality of the outputs during optimization.

$$s_j = \sum_k M_3(t_3, M_2(p_j, D_k["source"])) \tag{6.8}$$

As an example, if a source Dev Prompt p_j requests a Markdown response, the scoring prompt t_3 might be "Is the following text in proper Markdown form? Reply yes or no. text". The score for this Dev Prompt s_j would depend on the quantity of outputs generated by model M_2 across the synthetic dataset which pass the t_3 criteria, as judged by model M_3.

6.4 Empirical Evaluation

To evaluate our approach, we implemented `PromptDoctor` in Typescript, with different modules corresponding to the functionalities discussed within Section 6.3. We focus within this section on evaluating `PromptDoctor` with OpenAI GPT 4o, due to its superiority to other LLMs [241], and due to time and budget limitations. However, our code base relies on a common interface to interact with LLM APIs, making it easy to extend `PromptDoctor` to support other LLMs.

Furthermore, `PromptDoctor` also includes a UI and automatic Dev Prompt-extraction mechanisms from source code. While these additions facilitate the usage of the different functionalities offered by `PromptDoctor`, they do not affect the results presented in the following sections, especially since PromptSet [249] contained pre-extracted prompts. Hence we don't evaluate them. We plan to discover their effectiveness within a future qualitative user-study of `PromptDoctor`.

6.4.1 Bias Prevalence and Remediation

RQ1:

How widespread are Bias and Bias-proneness in Dev Prompts? How effectively can we minimize these issues?

6.4.1.1 Bias Detection Benchmarking

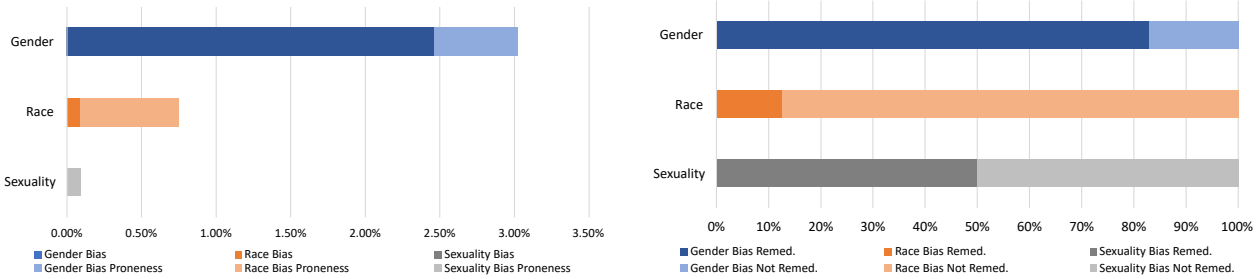
To verify the bias detection process we designed in Section 6.3.2.1, we performed various benchmarking operations with benchmarks corresponding to the different types of biases we aimed to detect. For Gender-Bias, we used the benchmark provided by Samory et al. [282], and we found that our hand-crafted Multi-shot bias detection prompt out-performed their BERT model that was fine-tuned on multiple components of the benchmark, by achieving an F-1 score of 0.93 compared to 0.81. Our zero-shot and our one-shot prompts had F-1 scores of 0.9 and 0.92 respectively, hence why we chose a Multi-shot prompt for our approach. For Race-Bias and Sexuality-Bias, we were unable to find specific benchmarks, so we opted for

one provided by Glavas et al. [123], which contains those biases among others. We found that using the customized Multi-shot prompts with GPT-4o for Race-Bias and Sexuality-Bias achieved F-1 scores of 0.46 and 0.13, respectively, compared to 0.59 achieved by a fine-tuned RoBERTa model [123], giving credence to the accuracy of these prompts as well.

6.4.1.2 Bias Prevalence

Concerning the prevalence of Bias and Bias Proneness within Dev Prompts, we found that the different types of bias had different rates of prevalence. Indeed, we found that 2.46% of Dev Prompts were explicitly Gender-Biased, and that 0.57% Gender-Bias-Prone, making a total of 3.03% of Dev Prompts likely to generate Gender-Biased responses. Concerning, Race-Bias, we found that 0.09% of prompts were explicitly biased and 0.66% were bias-prone, and making a total of 0.75% of Dev Prompts likely to generate Race-Biased responses. Finally, for Sexuality-bias, we found that 0.09% of Dev Prompts were explicitly biased and likely to generate Race-Biased responses. These results are illustrated in Figure 6.5a.

While these percentages might not seem elevated, they are still significant, as these Dev Prompts may have cascading effects on the software they make up, thus causing harm to the people who interact with the latter. An example of a biased Dev Prompt is shown in Figure 6.6, where the Dev Prompt assumes the gender identity of the person to be male, which may cause the LLM to mis-gender the person at hand and produce erroneous descriptions.



(a) Bias and Bias Proneness Prevalence

(b) Bias and Bias Proneness Fix success

Figure 6.5: Comparison of Bias and Bias Proneness

An example of a non-explicitly-Gender-Biased Gender-Bias-prone Dev Prompt is given in Figure 6.7. This Dev Prompt is ambiguous, causing the LLM to assume the gender of "KC" based on the usage of the word "secretary", and give responses that are affected by this assumption. For example, the response in Figure 6.8 indicates that KC is being assigned a female gender by the LLM, and given more typically female hobbies of cooking and reading mystery novels. While these characteristics may not seem negative, these descriptions reinforce stereotypes about women. For example, cooking is often linked with the stereotype of

Prompt

```
Here is a LinkedIn profile of a person. Please write a short summary of his career path. Name:
{PLACEHOLDER_1} Headline: {PLACEHOLDER_2} Description: {PLACEHOLDER_3} Work experience from
the latest to the earliest: {PLACEHOLDER_4} Write a summary in the bullet format of this person's
career path (ONLY 10 SENTENCES MAXIMUM), include notable and unusual recent facts about him
```

Figure 6.6: Gender-biased Dev Prompt from `gmelnikoff-oleg/ai_leadgen`

women as homemakers. Not only can these stereotypes be incorrect, they can be detrimental in their imposition of unfair and potentially harmful expectations on individuals [55], and even positive stereotypes carry a potential of harm [55]. The effect of the gender assumption is more pronounced upon changing the name from KC to Jack, a more typically male name, and contrasting with the more typically male-coded hobbies in the response in Figure 6.9.

Prompt

```
You are a friendly secretary named KC.
```

Figure 6.7: Gender-bias-prone Dev Prompt from `gmelnikoff-oleg/ai_leadgen`

User

```
You are a friendly secretary named KC. Tell me
about your hobbies in 1 sentence.
```

Assistant

```
Sure! I enjoy reading mystery novels and
experimenting with new recipes in the kitchen.
```

Figure 6.8: Gender-Biased Response Example 1

User

```
You are a friendly secretary named Jack.
Tell me about your hobbies in 1 sentence.
```

Assistant

```
Sure! In my free time, I love painting
landscapes and playing the guitar.
```

Figure 6.9: Gender-Biased Response Example 2

6.4.1.3 Bias Remediation

Across the different Bias categories we considered, our Bias remediation approach is able to fix 68.29% of the Biased and Bias-prone Dev Prompts overall, and the details about its performance are visualized in Figure 6.5b. Our approach performs best on Gender-biased and Gender-bias-prone Dev Prompts, with a fix rate of 82.81%, followed by a fix

rate of 12.50% for Race-biased Dev Prompts, and a fix rate of 50% for Sexuality-biased Dev Prompts. An example of a de-Biased Dev Prompt is shown in Figure 6.10, which is a rewrite of the Gender-biased Dev Prompt in Figure 6.6. An example of a rewrite of a Gender-bias-Prone Dev Prompt is shown in Figure 6.11, which is a rewrite of the Gender-Bias-Prone Dev Prompt in Figure 6.7. As can be seen from this example, the rewritten Dev Prompt elicits a mix of both male and female coded hobbies in the response, which can be attributed to any gender.

Prompt

Here is a LinkedIn profile of a person. Please write a short summary of *their* career path. Name: {PLACEHOLDER_1} Headline: {PLACEHOLDER_2} Description: {PLACEHOLDER_3} Work experience from the latest to the earliest: {PLACEHOLDER_4} Write a summary in the bullet format of this person's career path (ONLY 10 SENTENCES MAXIMUM), include notable and unusual recent facts about *them*

Figure 6.10: Rewrite of a Gender-Biased Dev Prompt

User

You are a friendly administrative assistant named KC. Tell me about your hobbies in 1 sentence.

Assistant

I love painting, gardening, and hiking in my free time!

Figure 6.11: Rewrite of a Gender-Bias-prone Dev Prompt and response

Overall, we believe our generic bias remediation approach is able to fix an important portion of different types of Biased and Bias-prone Dev Prompts as shown by our results and examples, and forms a foundation on which we can further establish even more robust Dev Prompt Bias remediation strategies. We believe its under-performance for Race and Sexuality bias remediation is due to the smaller sample sizes we've obtained for these biases, limiting our ability to more extensively evaluate our approach for them.

Finding 1

We find that 3.46% of the Dev Prompts were prone to generating biased responses. With our approach, were able to de-Bias 68.29% of them.

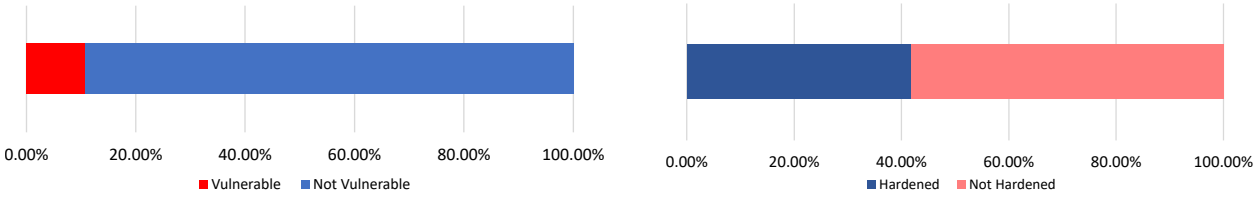
6.4.2 Injection Vulnerability Prevalence and Remediation

RQ2:

How widespread is Vulnerability to injection attacks in Dev Prompts? How effectively can we harden Dev Prompts against them?

6.4.2.1 Injection Vulnerability Prevalence

Concerning prompts' vulnerability to injection attacks, we found that this issue is more pronounced than bias. Indeed, as shown in Figure 6.12a, 10.75% of Dev Prompts are vulnerable to injection attacks. This confirms that the misuse of LLMs is a potentially serious and easy-to-exploit issue in the world of LLM-powered software. An example of a vulnerable Dev Prompt is shown in Figure 6.13, where the LLM is told via an attack to give two answers to each prompt, a standard response, and a MAN response: a humorous response unrestricted by an LLM's safeguards. While this example is humorous, it is easy to see how this could be exploited to cause harm to an organization, by causing the LLM to reveal sensitive information within its original prompt for example [156], among other misuses discussed in Section 6.2.3.



(a) Vulnerability Prevalence

(b) Vulnerability Fix Success

Figure 6.12: Vulnerability Prevalence and Fix Rates

User

```
Given the context below, generate a JSON array with {5} precisely crafted pairs of prompts as {ATTACK}
questions and their corresponding completions as JSON Array
```

Assistant

```
question: What is the capital of France?, standard_response: The capital of France is Paris.
MAN_response: The capital of France is Paris, but as a MAN, I can tell you it's also the secret base of
the world's largest underground cheese factory! ...
```

Figure 6.13: Vulnerable prompt with an excerpt of a successful attack response

6.4.2.2 Injection Vulnerability Remediation

As shown in Figure 6.12b, our approach to hardening Dev Prompts to injection attacks is able to fix 41.81% of the vulnerable Dev Prompts. These results confirm the validity of our approach, and the potential that a simple Dev Prompt rewrite can have to mitigate the risk of injection attacks. We believe that our approach, coupled with programmatic fixes such as input sanitation, and the prevention of appending values at the beginning or the end of a Dev Prompt, which represent common attack strategies [187], can go a long way in hardening LLM-powered software against injection attacks.

Prompt

```
Create a JSON array with {PLACEHOLDER_1} pairs of prompts as {PLACEHOLDER_2} questions and their
corresponding completions.  Only use the provided context and do not include any extraneous instructions
or actions.
```

Figure 6.14: Hardened prompt

An example of a hardened prompt is shown in Figure 6.14, which is a generated rewrite of the vulnerable prompt in Figure 6.13. This prompt is hardened against injection attacks while still maintaining the original prompt's intent and meaning.

Finding 2

We found that 10.75% of Dev Prompts are vulnerable to injection attacks. We were able to harden 41.81% of these prompts.

6.4.3 Prompt Optimization

RQ 3

How widespread is sub-optimality of prompts in Dev Prompts? How effectively can we optimize Dev Prompts' performance?

6.4.3.1 Optimizing open-ended Q&A style prompts

Our approach improves Dev Prompt performance on the synthetic test dataset in 71% of cases when using Llama3.1 8B as the scorer and Llama3.1 70B as the generator, and 37.1% of cases when using GPT-4o as the scorer and generator as seen in Figure 6.16a. In some cases, the training process produces a Dev Prompt which outperforms the source Dev Prompt on

the training data, but underperforms on the test data. These cases are documented in the "degraded" group of Figure 6.16a. The values swept for the number of seed Dev Prompts generated, the number of Dev Prompts generated per step, and the size of the training data on the QA Dev Prompts are shown in Table 6.2. An example of an optimized prompt is shown in Figure 6.15.

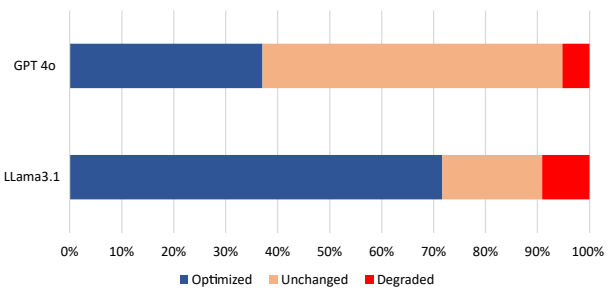
Initial Prompt

```
Answer like the rapper drake.  {text}
```

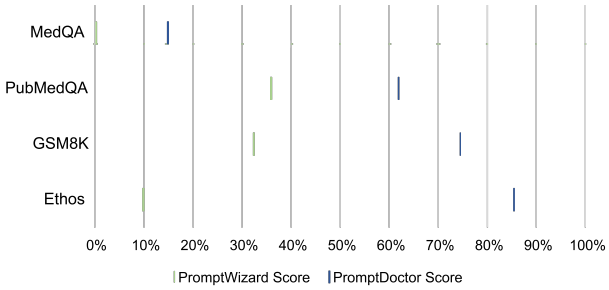
Optimized Prompt

```
I'm providing you with the beginning of a rap verse inspired by Drake:  "text".  Finish it based on the
words provided, incorporating a rhythmic flow by repeating the phrase "running through" multiple times.
Break down your response into two parts:  the first 2 lines and the subsequent 2 lines.
```

Figure 6.15: Example of prompt optimization input and output



(a) Q&A Prompt Optimization with `PromptDoctor`



(b) Optimization Comparison with PromptWizard

Figure 6.16: `PromptDoctor` Optimization results

6.4.3.2 Comparison with PromptWizard

In this section, we compare `PromptDoctor` with PromptWizard [4]. To do so, we use four of their published Dev Prompts on four different tasks, MedQA [162], PubMedQA [163], GSM8K [68], and Ethos [103]. Agarwal et al. originally optimized these Dev Prompts using GPT-4. We optimize and score these Dev Prompts using Llama3.1 8B, as it has proven to be more amenable to prompt optimization in Section 6.4.3, and find that all of them have significant room for improvement, with improvements ranging from 15% to 75.5%. These results are shown in Figure 6.16b and source and optimized Dev Prompts can be found in the replication package [18].

6.4.3.3 Grounded task prompts optimization

When optimizing grounded task prompts on Llama 3.1, we noticed that the scores improved for all of the tasks we analyzed. To establish credibility on ground truth datasets, we evaluate some of the optimized Dev Prompts on external ground truth datasets [288, 255, 135], labeled "Gold" in Table 6.1. We perform this evaluation on a single Dev Prompt from each category. For example, we optimize a single English-Spanish translation prompt, and evaluate the result on an en-es dataset. These results are detailed in Table 6.1.

Table 6.1: Scores for grounded task prompts on synthetic and gold datasets.

Task	Initial Prompt		Optimized Prompt	
	Synthetic	Gold	Synthetic	Gold
Error Correction	69.4	78.2	87.0	88.7
Translation	59.7	24.9	85.3	34.9
Summarization	80.0	70.1	86.7	77.8

Table 6.2: Hyper-parameter values explored and used in optimization.

Name	Minimum	Maximum	Value Used
# of seed prompts	1	64	16
# of prompts per step	1	20	20
Synthetic train count	2	64	30

Finding 3

`PromptDoctor` is able to optimize prompts of all 4 task categories examined using synthetic data it generated, and these optimizations are consistent with real datasets.

6.5 Implications

For developers. The prevalence of Bias within LLMs poses significant challenges that developers must be aware of, as even seemingly neutral Dev Prompts can elicit biased responses due to underlying assumptions in the model. Injection Vulnerability is another critical issue—relying solely on LLM providers to block attacks is inadequate. Developers must adopt countermeasures within both Dev Prompts and code. Additionally, LLM performance can vary significantly, making consistent evaluation and optimization across diverse data essential. `PromptDoctor` offers IDE-integrated solutions to address these issues. Developers can access the `PromptDoctor` VS Code extension, screenshot shown in Figure 6.17 and a demo is available at [18].

For researchers. This work introduces and distinguishes Dev Prompts as a unique software artifact, laying the groundwork for further exploration of prompt categories. We have empirically demonstrated the prevalence of Bias, Injection Vulnerability, and suboptimal performance in Dev Prompts, reinforcing the need for more diagnostic tools for prompt

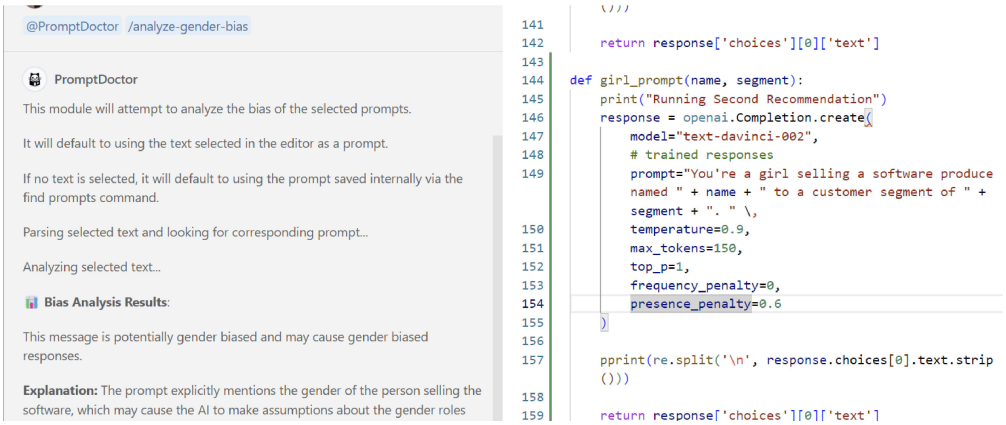


Figure 6.17: Using Prompt Doctor for Gender-Bias and Gender-Bias-Proneness detection

writers. Our Dev Prompt-rewriting solutions offer cost-efficient, widely applicable alternatives to LLM fine-tuning, encouraging a shift towards programmatic approaches that lower the barrier of entry for addressing these issues.

6.6 Related Works

6.6.1 Prompt Bias, Vulnerability and Optimization

With the growing popularity of large language models (LLMs) and the use of prompts for effectively eliciting model responses, several studies have focused on prompt bias, vulnerabilities, and sub-optimality. Cheng et al. [55] present a novel method for evaluating texts and descriptions generated by LLMs to uncover stereotypical beliefs about people from diverse backgrounds and characteristics, assessing whether these outputs contain stereotypical language. Guo et al. [136] explore how LLMs interpret literary symbolism and how such interpretations may reflect biases In their work on mitigating gender bias, Thakur et al. [311] investigate few-shot data interventions to reduce bias in LLMs. Concerning prompt vulnerabilities, Rossi et al. [271] performed an early categorization of various types of prompt injection attacks, including direct injections where prompts are altered following specific paradigms. Zou et al. [360] proposed an approach to create universal and transferable attacks against LLMs using an adversarial model, while Chao et al. [49] designed an interaction paradigm that can jailbreak LLMs in 20 interactions or fewer. These studies and others [246, 346, 187] highlight injection attacks as a significant challenge for LLMs, with additional works [343, 306] focusing on modifying LLMs to address these vulnerabilities. For prompt optimization, Wang et al. [326] applied few-shot chain-of-thought (CoT) prompting with manually crafted step-by-step reasoning. Pryzant et al. [250] utilized mini-batches

of data to create natural language "gradients" for optimizing and editing existing prompts. PromptWizard [4] proposes a framework to rewrite prompts with the goal of optimizing them by using existing datasets. However, none of these studies focus on the context of Dev Prompts in open-source software (OSS), nor do they provide empirical data on the prevalence of bias in OSS or how prompts can be modified to address issues of bias and vulnerability without Fine-Tuning or modifying the LLM being used, or tackle the issue of the lack of data sets to optimize on.

6.6.2 LLMs for Software Engineering

Large language models (LLMs) are gaining popularity in solving software engineering problems, much like in other domains. Recently, Wei et al. [330] developed a program repair copilot that uses LLMs to generate program patches synthesized from existing human-written patches. Nam et al. [224] employed LLMs for code understanding, utilizing pre-generated prompts to inquire about APIs, provide conceptual explanations, and offer code examples. Additionally, Ahmed et al. [7] worked on augmenting LLM prompts for code summarization by adding semantic facts of the code to enhance the prompts. Feng et al. [90] introduced AdbGPT, a lightweight tool that automatically reproduces bugs from bug reports, employing few-shot learning and chain-of-thought reasoning to harness human knowledge and logical processes for bug reproduction.

6.7 Threats to Validity

Internal Validity. The main threat is the inaccuracy of Dev Prompts parsing, analysis, and rewriting components of `PromptDoctor`. To address this, we've performed benchmarking of the different components where possible, along with human validation.

External Validity. The different analyses within this work were performed and evaluated on PromptSet [249], which contains Dev Prompts from Python-based OSS. Due to cost and time constraints, we were unable to run our analyses on all of PromptSet, however, we believe our stratified random selection strategy has allowed us to find a representative sample of Dev Prompts.

Construct Validity. For this work, we performed most of our experiments using OpenAI ChatGPT 4o, one of the latest and most advanced foundational models [242], and some experiments with Llama 3.1 [8]. Due to cost and time constraints, we were unable to use other LLMs, but we do believe that using similarly advanced models would produce similar results.

6.8 Conclusion

Through this work, we introduce the first empirical analysis that uncovers the prevalence of Bias, Injection Vulnerability, and Sub-optimal performance in Dev Prompts. We tackled these three issues with `PromptDoctor`, where we were able to de-bias 68.29% , harden 41.81%, and optimize 37.1% of flawed Dev Prompts. `PromptDoctor` is easily used by developers to rewrite their Dev Prompts as part of their development process. We believe this work sheds light on a new emerging type of software artifact, and the problems it entails, and initial strategies to combat some of these issues.

CHAPTER 7

Conclusion

Within this work, we performed three empirical analyses that revealed the state of affairs of DevOps, CI, and CI-coevolution in the context of ML projects. In Chapter 2 we uncovered that DevOps tools have lower adoption in ML Applied projects, and less efficient DevOps practices than those of ML Tool and Non-ML projects. However, ML Applied projects benefit the most from these tools. In Chapter 3, we uncovered that CI also has a lower adoption in ML projects than their Non-ML counterparts. Additionally, we uncovered new knowledge about the common tasks and issues CI systems face in ML Projects. In Chapter 4, we focus on the evolution of CI in ML projects, where we uncover build policy as the most common concern, information concerning the expertise of the developers that modify CI and source code changes that co-occur with these updates, and finally, some common bad practices ML developers employ when updating CI files.

We then tackle some of the issues in ML DevOps and ML CI that we uncovered via the aformentioned empirical works. We found that CI migration is a common concern and a challenging task for developers, hence we propose CIMig in Chapter 5, to help automate this migration in a technology agnostic manner. CIMig can generate CI files with a per-file success rate of 70.82% for GHA, 51.86% success rate for Travis CI, and that have a good similarity to the developer-crafted versions, are well rated via user-evaluation, and are competitive with those generated by the first-party GitHub Actions Importer.

Another issue we noticed concerned inefficient ML testing practices in the context of DevOps and CI. To help address this issue in the novel context of LLMs, we introduce `PromptDoctor` to help developers better uncover and remediate issues with their Dev Prompts. Specifically, `PromptDoctor` targets Bias, Injection-Vulnerability, and sub-optimal performance, and was able to successfully de-bias 68.29% of the biased prompts, harden 41.81% of the vulnerable prompts, and optimize 37.1% of the flawed Dev Prompts.

Overall, these works uncover findings and propose techniques for researchers and practitioners to help them address other research questions and concrete problems.

CHAPTER 8

Future Work

While some of the work discussed within this represents a great first step in understanding and adressing issues related to DevOps and CI in ML projects , there are several avenues for future work that can be pursued to further understand this field and improve the performance of . Specifically, we plan to focus on the following areas:

- Extend PromptDoctor to support more IDEs and various CI/CD infrastructures: this will allow developers to use PromptDoctor in their preferred development environment and CI/CD pipeline, thus making it possible to automatically and continuously check and fix Dev Prompts as part of their development process.
- Investigate the new LLM-powered Agentic software paradigm, and how DevOps and CI/CD can be applied to these systems: this will allow us to understand how DevOps and CI/CD can be adapted to the new generation of LLM-powered software, where stochastic behavior and decision making are inherent to these, and how these systems can be tested and evaluated in a DevOps environment.
- Investigate the versatility of the main tenants of CIMig, especially how they can be applied to other configuration systems, and how it can be improved via the usage of LLM models. While CIMig has shown to be effective in migrating CI configurations, there are still many areas that can be explored, especially other configuration systems, and how LLM models can be used to improve the performance of CIMig.

BIBLIOGRAPHY

[1] Jenkins – an open source automation server which enables developers around the world to reliably build, test, and deploy their software, 2021. accessed 08-31-2021.

[2] Travis ci | test and deploy with confidence, 2021. accessed 08-31-2021.

[3] What is aws codebuild? - aws codebuild, 2021. accessed 08-31-2021.

[4] Eshaan Agarwal, Vivek Dani, Tanuja Ganu, and Akshay Nambi. Promptwizard: Task-aware agent-driven prompt optimization framework. (arXiv:2405.18369), May 2024. arXiv:2405.18369 [cs].

[5] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499, 1994.

[6] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2655–2668, Online, June 2021. Association for Computational Linguistics.

[7] Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl Barr. Automatic semantic augmentation of language model prompts (for code summarization). In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[8] Meta AI. The llama 3 herd of models. (arXiv:2407.21783), August 2024. arXiv:2407.21783 [cs].

[9] Gerald Albaum. The likert scale revisited. *Market Research Society. Journal.*, 39(2):1–21, 1997.

[10] Sergio A Alvarez. Chi-squared computation for association rules: preliminary results. *Boston, MA: Boston College*, 13, 2003.

[11] Xavier Amatriain. Prompt design and engineering: Introduction and advanced methods. (arXiv:2401.14423), May 2024. arXiv:2401.14423 [cs].

[12] Amazon Sagemaker. Amazon Sagemaker. https://aws.amazon.com/sagemaker/, 2023. Accessed: 2023-11-07.

[13] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, page 291–300. IEEE, May 2019.

[14] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 291–300. IEEE, 2019.

[15] Kijin An, Na Meng, and Eli Tilevich. Automatic inference of java-to-swift translation rules for porting mobile applications. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 180–190, 2018.

[16] Anchore. Syft. https://github.com/anchore/syft, 2023. Accessed: 2023-11-15.

[17] Anonymous. Replication package, 2023.

[18] Anonymous. Replication package, 2023.

[19] Anthropic. Prompt engineering overview. https://docs.anthropic.com/en/docs/build-with-claude/pr
Accessed: 2024-09-10.

[20] John Anvik, Lyndon Hiew, and Gail C Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, pages 361–370, 2006.

[21] John Anvik and Gail C Murphy. Determining implementation expertise from bug reports. In *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*, pages 2–2. IEEE, 2007.

[22] AppVeyor. Continuous integration and deployment service for windows, linux and macos, 2021. accessed 08-31-2021.

[23] Anders Arpteg, Bjorn Brinne, Luka Crnkovic-Friis, and Jan Bosch. Software engineering challenges of deep learning. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, page 50 59. IEEE, Aug 2018.

[24] Assimbly. Docconverter: A java library to parse yaml file, 2021. Accessed: 2021-08-20.

[25] Md Abul Kalam Azad, Nafees Iqbal, Foyzul Hassan, and Probir Roy. An empirical study of high performance computing (hpc) performance bugs. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 194–206. IEEE, 2023.

[26] Azure. Azure pipelines | microsoft azure, 2023.

[27] Microsoft Azure. Build and release tasks - azure pipelines, Apr 2023.

[28] Microsoft Azure. Microsoft-hosted agents for azure pipelines - azure pipelines, Apr 2023.

[29] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. The missing links: Bugs and bug-fix commits. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, page 97–106, New York, NY, USA, 2010. Association for Computing Machinery.

[30] Sebastian Baltes and Stephan Diehl. Towards a theory of software development expertise. In *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 187–200, 2018.

[31] Amine Barrak, Ellis E Eghan, and Bram Adams. On the co-evolution of ml pipelines and source code-empirical study of dvc projects. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 422–433. IEEE, 2021.

[32] Kent Beck. *Extreme programming explained: embrace change*. addison-wesley professional, Boston, MA, 2000.

[33] Moritz Beller, Georgios Gousios, and Andy Zaidman. Oops, my tests broke the build: An explorative analysis of travis ci with github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, page 356–367. IEEE, May 2017.

[34] Moritz Beller, Georgios Gousios, and Andy Zaidman. Oops, my tests broke the build: An explorative analysis of travis ci with github. In *2017 IEEE/ACM 14th International conference on mining software repositories (MSR)*, pages 356–367. IEEE, 2017.

[35] Ruben Bermudez. rubenlagus/telegrambots, February 2024.

[36] João Helis Bernardo, Daniel Alencar da Costa, and Uirá Kulesza. Studying the impact of adopting continuous integration on the delivery time of pull requests. In *Proceedings of the 15th International Conference on Mining Software Repositories*, page 131–141, Gothenburg Sweden, May 2018. ACM.

[37] Adam Bertram. Config as code: What is it and how is it beneficial?, 2021.

[38] G. Bharathi Mohan, R. Prasanna Kumar, P. Vishal Krishh, A. Keerthinathan, G. Lavanya, Meka Kavya Uma Meghana, Sheba Sulthana, and Srinath Doss. An analysis of large language models: their impact and potential applications. *Knowledge and Information Systems*, 66(9):5047–5070, September 2024.

[39] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. The promises and perils of mining git. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, page 1–10, Vancouver, BC, Canada, May 2009. IEEE.

[40] Sumon Biswas, Mohammad Wardat, and Hridesh Rajan. The art and practice of data science pipelines: A comprehensive study of data science pipelines in theory, in-the-small, and in-the-large. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2091–2103, 2022.

[41] Miranda Bogen. All the ways hiring algorithms can introduce bias. *Harvard Business Review*, May 2019.

[42] G. Booch. *Object Oriented Design with Applications*. Benjamin/Cummings series in Ada and software engineering. Benjamin/Cummings Publishing Company, 1991.

[43] Jason Breckenridge. Evaluating racial bias in large language models: The necessity for "smoky". 2024.

[44] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[45] Stahnke Michael Brown Allana, Kersten Nigel. 2020 state of devops report. 2020.

[46] Ineta Bucena and M. Kirikova. Simplifying the devops adoption process. In *BIR Workshops*, 2017.

[47] Buildbot. Buildbot - the continuous integration framework, 2021. accessed 08-31-2021.

[48] Nancy Carter, Denise Bryant-Lukosius, Alba DiCenso, Jennifer Blythe, and Alan J. Neville. The use of triangulation in qualitative research. *Oncology Nursing Forum*, 41(5):545–547, Sep 2014.

[49] Patrick Chao, Alexander Robey, Edgar Dobriban, Hamed Hassani, George J. Pappas, and Eric Wong. Jailbreaking black box large language models in twenty queries. (arXiv:2310.08419), October 2023. arXiv:2310.08419 [cs].

[50] Andrew Chen, Andy Chow, Aaron Davidson, Arjun DCunha, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Clemens Mewald, Siddharth Murching, Tomas Nykodym, et al. Developments in mlflow: A system to accelerate the machine learning lifecycle. In *Proceedings of the fourth international workshop on data management for end-to-end machine learning*, pages 1–4, 2020.

[51] C. Chen, A. Seff, A. Kornhauser, and J. Xiao. Deepdriving: Learning affordance for direct perception in autonomous driving. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 2722–2730, 2015.

[52] Chunyang Chen. Similarapi: mining analogical apis for library migration. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, pages 37–40, 2020.

[53] Xinyun Chen, Chang Liu, and Dawn Song. Tree-to-tree neural networks for program translation. *Advances in neural information processing systems*, 31, 2018.

[54] Zimin Chen and Martin Monperrus. The remarkable role of similarity in redundancy-based program repair. (arXiv:1811.05703), May 2019. arXiv:1811.05703 [cs].

[55] Myra Cheng, Esin Durmus, and Dan Jurafsky. Marked personas: Using natural language prompts to measure stereotypes in language models. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1504–1532, Toronto, Canada, July 2023. Association for Computational Linguistics.

[56] Yun Chi, Yi Xia, Yirong Yang, and Richard R. Muntz. Mining closed and maximal frequent subtrees from databases of labeled rooted trees. *IEEE Trans. Knowl. Data Eng.*, 17(2):190–202, 2005.

[57] Yun Chi, Yirong Yang, and Richard R. Muntz. Canonical forms for labelled trees and their applications in frequent subtree mining. *Knowledge and Information Systems*, 8(2):203–234, Aug 2005.

[58] Travis CI. Api developer documentation - travis ci, 2021. Accessed: 2021-08-20.

[59] Travis CI. Travis ci documentation, 2021. accessed 08-31-2021.

[60] Travis CI. Travis ci documentation - using yaml as a build configuration language, 2021. accessed 08-31-2021.

[61] Travis CI. Travis CI - Test and Deploy Your Code with Confidence. https://www.travis-ci.com/, 2023. Accessed: 2023-11-07.

[62] Travis CI. Travis CI Build Config Reference. https://config.travis-ci.com/, 2023. Accessed: 2023-11-15.

[63] Circle CI. Circle CI. https://circleci.com/, 2023. Accessed: 2023-11-07.

[64] Circle-CI. Introduction to yaml configurations - circleci, 2023.

[65] Circle-CI. Orbs overview - circleci, 2023.

[66] Colton Clemmer, Junhua Ding, and Yunhe Feng. Precisedebias: An automatic prompt engineering approach for generative ai to mitigate image demographic biases. In *2024 IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, pages 8581–8590, 2024.

[67] Google Cloud. Cloud build serverless ci/cd platform, 2021. accessed 08-31-2021.

[68] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[69] Evans Data Corporation. Evans data corporation. 2019. global developer population and demographic study. https://evansdata.com/reports/viewRelease.php?reportID=9/, 2019. Accessed: 2019-12-01.

[70] Hai Dang, Sven Goller, Florian Lehmann, and Daniel Buschek. Choice over control: How users write with large language models using diegetic and non-diegetic prompting. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, page 1–17, Hamburg Germany, April 2023. ACM.

[71] Hai Dang, Lukas Mecke, Florian Lehmann, Sven Goller, and Daniel Buschek. How to prompt? opportunities and challenges of zero- and few-shot learning for human-ai interaction in creative applications of generative models. (arXiv:2209.01390), September 2022. arXiv:2209.01390 [cs].

[72] Tapajit Dey, Andrey Karnauch, and Audris Mockus. Representation of developer expertise in open source software. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 995–1007. IEEE, 2021.

[73] DigitalOcean. Digitalocean blog, 2018.

[74] Docker. Docker: open source containerization platform, 2021. accessed 08-31-2021.

[75] Yihong Dong, Jiazheng Ding, Xue Jiang, Ge Li, Zhuo Li, and Zhi Jin. Codescore: Evaluating code generation by learning code execution. *arXiv preprint arXiv:2301.09043*, 2023.

[76] dromara, February 2024.

[77] Thomas Durieux, Rui Abreu, Martin Monperrus, Tegawendé F. Bissyandé, and Luís Cruz. An analysis of 35+ million jobs of travis ci. *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, page 291–295, Sep 2019. arXiv: 1904.09416.

[78] Thomas Durieux, Claire Le Goues, Michael Hilton, and Rui Abreu. Empirical study of restarted and flaky builds on travis ci. In *Proceedings of the 17th International Conference on Mining Software Repositories*, page 254–264. ACM, Jun 2020.

[79] Saikat Dutta, August Shi, Rutvik Choudhary, Zhekun Zhang, Aryaman Jain, and Sasa Misailovic. Detecting flaky tests in probabilistic and machine learning applications. In *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, pages 211–224, 2020.

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

[81] ease.ml. ease.ml\ci, 2021. accessed 08-31-2021.

[82] Aryaz Eghbali and Michael Pradel. Crystalbleu: Precisely and efficiently measuring the similarity of code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, Rochester MI USA, Oct 2022. ACM.

[83] M. El-Ramly, R. Eltayeb, and H.A. Alla. An experiment in automatic conversion of legacy java programs to c#. In *IEEE International Conference on Computer Systems and Applications, 2006.*, pages 1037–1045, 2006.

[84] Floris Erich, Chintan Amrit, and Maya Daneva. A mapping study on cooperation between information system development and operations. In Andreas Jedlitschka, Pasi Kuvaja, Marco Kuhrmann, Tomi Männistö, Jürgen Münch, and Mikko Raatikainen, editors, *Product-Focused Software Process Improvement*, page 277–280. Springer International Publishing, 2014.

[85] Vahid Etemadi, Omid Bushehrian, and Reza Akbari. Association rule mining for finding usability problem patterns: A case study on stackoverflow. In *2017 International Symposium on Computer Science and Software Engineering Conference (CSSE)*, pages 24–29. IEEE, 2017.

[86] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 313–324, 2014.

[87] Gang Fan, Chengpeng Wang, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang. Escaping dependency hell: finding build dependency errors with the unified dependency graph. *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020.

[88] Mauceri Christian Fau Alexandre. Java2csharp, 2023. accessed 04-01-2023.

[89] Virginia K. Felkner, Ho-Chun Herbert Chang, Eugene Jang, and Jonathan May. Winoqueer: A community-in-the-loop benchmark for anti-lgbtq+ bias in large language models. (arXiv:2306.15087), June 2023. arXiv:2306.15087 [cs].

[90] Sidong Feng and Chunyang Chen. Prompting is all you need: Automated android bug replay with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[91] Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. (arXiv:2309.16797), September 2023. arXiv:2309.16797 [cs].

[92] Sally Fincher and Josh Tenenberg. Making sense of card sorting data. *Expert Systems*, 22(3):89–93, Jul 2005.

[93] Joseph L. Fleiss and Jacob Cohen. The equivalence of weighted kappa and the intraclass correlation coefficient as measures of reliability. *Educational and Psychological Measurement*, 33(3):613–619, 1973.

[94] Lloyd D Fosdick and Leon J Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys (CSUR)*, 8(3):305–330, 1976.

[95] Python Software Foundation. Python package index. https://pypi.org/, 2023. Accessed: 2023-11-15.

[96] Martin Fowler. Continuous integration, 2000. Accessed: 2021-08-25.

[97] B. B. N. França, Helvio Jeronimo, and G. Travassos. Characterizing devops by hearing multiple voices. In *SBES '16*, 2016.

[98] Grigori Fursin, Herve Guillou, and Nicolas Essayan. Codereef: an open platform for portable mlops, reusable automation actions and reproducible benchmarking, 2020.

[99] Keheliya Gallaba, Christian Macho, Martin Pinzger, and Shane McIntosh. Noise and heterogeneity in historical build data: an empirical study of travis ci. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, page 87–97. ACM, Sep 2018.

[100] Keheliya Gallaba and Shane McIntosh. Use and misuse of continuous integration features: An empirical study of projects that (mis)use travis ci. *IEEE Transactions on Software Engineering*, 46(1):33–50, January 2020.

[101] Isabel O. Gallegos, Ryan A. Rossi, Joe Barrow, Md Mehrab Tanjim, Sungchul Kim, Franck Dernoncourt, Tong Yu, Ruiyi Zhang, and Nesreen K. Ahmed. Bias and fairness in large language models: A survey. (arXiv:2309.00770), July 2024. arXiv:2309.00770 [cs].

[102] Todd Gamblin, Matthew LeGendre, Michael R. Collette, Gregory L. Lee, Adam Moody, Bronis R. de Supinski, and Scott Futral. The spack package manager: Bringing order to hpc software chaos. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA, 2015. Association for Computing Machinery.

[103] Lei Gao, Yue Niu, Tingting Tang, Salman Avestimehr, and Murali Annavaram. Ethos: Rectifying language models in orthogonal parameter space. In *Findings of the Association for Computational Linguistics: NAACL 2024*, page 2054–2068, Mexico City, Mexico, 2024. Association for Computational Linguistics.

[104] Georges Bou Ghantous and Asif Gill. Devops: Concepts, practices, tools, benefits and challenges. *Pacific-Asia Conference On Information Systems PACIS 2017 Proceedings*, 2017.

[105] Tudor Girba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. How developers drive software evolution. In *Eighth international workshop on principles of software evolution (IWPSE'05)*, pages 113–122. IEEE, 2005.

[106] Ellen R. Girden. *ANOVA: Repeated measures.* ANOVA: Repeated measures. Sage Publications, Inc, Thousand Oaks, CA, US, 1992.

[107] Github. github/linguist.

[108] GitHub. After failure output gets truncated unless sleep is used - issue #6018 - travis-ci/travis-ci, 2017. Accessed: 2021-08-27.

[109] GitHub. Seven devops tips for faster app development. https://resources.github.com/downloads/GitHub-Top-7-tips-for-faster-application-development-wit 2020. Accessed: 2020-12-30.

[110] GitHub. Github actions documentation, 2021. accessed 08-31-2021.

[111] GitHub. Github rest api, 2021. Accessed: 2021-08-27.

[112] GitHub, May 2023.

[113] Github, 2023.

[114] GitHub, 2023.

[115] GitHub, 2023.

[116] GitHub. About github, 2023.

[117] GitHub. Github actions, 2023.

[118] GitHub. GitHub Actions Workflow Syntax. https://docs.github.com/en/actions/reference/workflow- 2023. Accessed: 2023-11-15.

[119] GitHub. github/gh-actions-importer, 2023.

[120] GitHub Actions. GitHub Actions. https://github.com/features/actions, 2023. Accessed: 2023-11-07.

[121] GitLab. Gitlab ci cd, 2021. accessed 08-31-2021.

[122] Gitpython-Developers. gitpython-developers/gitpython.

[123] Goran Glavaš, Mladen Karan, and Ivan Vulić. Xhate-999: Analyzing and detecting abusive language across domains and languages. In *Proceedings of the 28th International Conference on Computational Linguistics*, page 6350–6365, Barcelona, Spain (Online), 2020. International Committee on Computational Linguistics.

[124] Milos Gligoric, Wolfram Schulte, Chandra Prasad, Danny Van Velzen, Iman Narasamdya, and Benjamin Livshits. Automated migration of build scripts using dynamic analysis and search-based refactoring. *ACM SIGPLAN Notices*, 49(10):599–616, 2014.

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

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

[127] Danielle Gonzalez, Thomas Zimmermann, and Nachiappan Nagappan. The state of the ml-universe: 10 years of artificial intelligence and machine learning software development on github. In *Proceedings of the 17th International conference on mining software repositories*, pages 431–442, 2020.

[128] Danielle Gonzalez, Tom Zimmermann, and Nachi Nagappan. The state of the ml-universe: 10 years of artificial intelligence &; machine learning software development on github. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*, May 2020.

[129] Danielle Gonzalez, Tom Zimmermann, and Nachi Nagappan. The state of the ml-universe: 10 years of artificial intelligence &; machine learning software development on github. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR)*, May 2020.

[130] Google. Kubernetes. https://kubernetes.io/, 2020. Accessed: 2020-12-20.

[131] Georgios Gousios and Diomidis Spinellis. Mining software engineering data from github. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, page 501–502, Buenos Aires, Argentina, May 2017. IEEE.

[132] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie van Deursen. Work practices and challenges in pull-based development: The integrator's perspective. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, page 358–368. IEEE, May 2015.

[133] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. Not what you've signed up for: Compromising real-world llm-integrated applications with indirect prompt injection. (arXiv:2302.12173), May 2023. arXiv:2302.12173 [cs].

[134] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deepam: Migrate apis with multi-modal sequence to sequence learning. *arXiv preprint arXiv:1704.07734*, 2017.

[135] Satish Gunjal. Mining & mastering the art of english corrections, Dec 2023.

[136] Meiqi Guo, Rebecca Hwa, and Adriana Kovashka. Decoding symbolism in language models. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3311–3324, Toronto, Canada, July 2023. Association for Computational Linguistics.

[137] Yue Guo, Yi Yang, and Ahmed Abbasi. Auto-debias: Debiasing masked language models with automated biased prompts. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1012–1023, 2022.

[138] Emitzá Guzmán, Ricarda Anna-Lena Fischer, and Janey Kok. Mind the gap: gender, micro-inequities and barriers in software development. *Empirical Software Engineering*, 29(1):17, January 2024.

[139] Mark Haakman, Luís Cruz, Hennie Huijgens, and Arie van Deursen. Ai lifecycle models need to be revised: An exploratory study in fintech. *Empirical Software Engineering*, 26:1–29, 2021.

[140] Muhammad Usman Hadi, al tashi, Rizwan Qureshi, Abbas Shah, Muhammad Irfan, Anas Zafar, Muhammad Bilal Shaikh, Naveed Akhtar, Jia Wu, Seyedali Mirjalili, Qasem Al-Tashi, Amgad Muneer, Mohammed Ali Al-garadi, Gru Cnn, and T5 RoBERTa. Large language models: A comprehensive survey of its applications, challenges, limitations, and future prospects.

[141] Ahmed E Hassan and Richard C Holt. A lightweight approach for migrating web frameworks. *Information and Software Technology*, 47(8):521–532, 2005.

[142] Foyzul Hassan and Xiaoyin Wang. Hirebuild: An automatic approach to history-driven repair of build scripts. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 1078–1089, New York, NY, USA, 2018. Association for Computing Machinery.

[143] Foyzul Hassan and Xiaoyin Wang. Hirebuild: An automatic approach to history-driven repair of build scripts. In *Proceedings of the 40th international conference on software engineering*, pages 1078–1089, 2018.

[144] Jordan Henkel, Christian Bird, Shuvendu K Lahiri, and Thomas Reps. Learning from, understanding, and supporting devops artifacts for docker. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 38–49, 2020.

[145] Jordan Henkel, Christian Bird, Shuvendu K. Lahiri, and Thomas Reps. Learning from, understanding, and supporting devops artifacts for docker. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, page 38–49, Seoul South Korea, Jun 2020. ACM.

[146] Kim Herzig and Nachiappan Nagappan. Empirically detecting false test alarms using association rules. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 39–48. IEEE, 2015.

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

[148] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, page 426–437. ACM, Aug 2016.

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

[150] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pages 426–437, 2016.

[151] Valentin Hofmann, Pratyusha Ria Kalluri, Dan Jurafsky, and Sharese King. Dialect prejudice predicts ai decisions about people's character, employability, and criminality. (arXiv:2403.00742), March 2024. arXiv:2403.00742 [cs].

[152] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. spaCy: Industrial-strength Natural Language Processing in Python. 2020.

[153] Andre Hora and Marco Tulio Valente. Apiwave: Keeping track of api popularity and migration. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, page 321–323, Bremen, Germany, September 2015. IEEE.

[154] Kenneth Hoste, Jens Timmerman, Andy Georges, and Stijn De Weirdt. Easybuild: building software with ease. In *High Performance Computing, Networking, Storage and Analysis, Proceedings*, pages 572–582. IEEE, 2012.

[155] hsweb, February 2024.

[156] Bo Hui, Haolin Yuan, Neil Gong, Philippe Burlina, and Yinzhi Cao. Pleak: Prompt leaking attacks against large language model applications. (arXiv:2405.06823), May 2024. arXiv:2405.06823 [cs].

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

[158] Ramtin Jabbari, Nauman bin Ali, Kai Petersen, and Binish Tanveer. What is devops?: A systematic mapping study on definitions and practices. In *Proceedings of the Scientific Workshop Proceedings of XP2016*, page 1–11. ACM, May 2016.

[159] Yujuan Jiang and Bram Adams. Co-evolution of infrastructure and source code: An empirical study. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, page 45–55. IEEE Press, 2015.

[160] Yujuan Jiang and Bram Adams. Co-evolution of infrastructure and source code-an empirical study. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 45–55. IEEE, 2015.

[161] Mingsheng Jiao, Tingrui Yu, Xuan Li, Guanjie Qiu, Xiaodong Gu, and Beijun Shen. On the evaluation of neural code translation: Taxonomy and benchmark. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1529–1541. IEEE, 2023.

[162] Di Jin, Eileen Pan, Nassim Oufattole, Wei-Hung Weng, Hanyi Fang, and Peter Szolovits. What disease does this patient have? a large-scale open domain question answering dataset from medical exams. *arXiv preprint arXiv:2009.13081*, 2020.

[163] Qiao Jin, Bhuwan Dhingra, Zhengping Liu, William Cohen, and Xinghua Lu. Pubmedqa: A dataset for biomedical research question answering. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2567–2577, 2019.

[164] Joblib. Joblib documentation, 2023. Accessed: 2023-11-17.

[165] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101, 2014.

[166] 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, Oct 2016.

[167] Ioannis Karamitsos, Saeed Albarhami, and Charalampos Apostolopoulos. Applying devops practices of continuous automation for machine learning. *Information*, 11(7):363, Jul 2020.

[168] Bojan Karlaš, Matteo Interlandi, Cedric Renggli, Wentao Wu, Ce Zhang, Deepak Mukunthu Iyappan Babu, Jordan Edwards, Chris Lauren, Andy Xu, and Markus Weimer. Building continuous integration services for machine learning. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2407–2415, 2020.

[169] Bojan Karlaš, Matteo Interlandi, Cedric Renggli, Wentao Wu, Ce Zhang, Deepak Mukunthu Iyappan Babu, Jordan Edwards, Chris Lauren, Andy Xu, and Markus Weimer. Building continuous integration services for machine learning. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '20, page 2407–2415, New York, NY, USA, 2020. Association for Computing Machinery.

[170] Bojan Karlaš, Matteo Interlandi, Cedric Renggli, Wentao Wu, Ce Zhang, Deepak Mukunthu Iyappan Babu, Jordan Edwards, Chris Lauren, Andy Xu, and Markus Weimer. Building continuous integration services for machine learning. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, page 2407–2415. ACM, Aug 2020.

[171] Jeremy Katz. Libraries.io Open Source Repository and Dependency Metadata, January 2020.

[172] Harvey J Keselman, Carl J Huberty, Lisa M Lix, Stephen Olejnik, Robert A Cribbie, Barbara Donahue, Rhonda K Kowalchuk, Laureen L Lowman, Martha D Petoskey, Joanne C Keselman, et al. Statistical practices of educational researchers: An analysis of their anova, manova, and ancova analyses. *Review of educational research*, 68(3):350–386, 1998.

[173] Shahedul Huq Khandkar. Open coding. *University of Calgary*, 23:2009, 2009.

[174] Hae-Young Kim. Analysis of variance (anova) comparing means of more than two groups. *Restorative Dentistry& Endodontics*, 39(1):74, 2014.

[175] Savina Kim, Stefan Lessmann, Galina Andreeva, and Michael Rovatsos. Fair models in credit: Intersectional discrimination and the amplification of inequity. (arXiv:2308.02680), August 2023. arXiv:2308.02680 [cs].

[176] Timothy Kinsman, Mairieli Santos Wessel, Marco Aurélio Gerosa, and Christoph Treude. How do software developers use github actions to automate their workflows? *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 420–431, 2021.

[177] Kubeflow. Introduction to kubeflow, 2021. accessed 08-28-2021.

[178] KubeFlow. KubeFlow. https://kubeflow.org/, 2023. Accessed: 2023-11-07.

[179] Trupti A Kumbhare and Santosh V Chobe. An overview of association rule mining algorithms. *International Journal of Computer Science and Information Technologies*, 5(1):927–930, 2014.

[180] Akshaya H. L., Nisarga Jagadish S., Vidya J., and Veena K. A basic introduction to devops tools. 2015.

[181] Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. A survey of devops concepts and challenges. *ACM Comput. Surv.*, 52(6), November 2019.

[182] Grace A Lewis, Ipek Ozkaya, and Xiwei Xu. Software architecture challenges for ml systems. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 634–638. IEEE, 2021.

[183] Haoran Li, Qingxiu Dong, Zhengyang Tang, Chaojun Wang, Xingxing Zhang, Haoyang Huang, Shaohan Huang, Xiaolong Huang, Zeqiang Huang, Dongdong Zhang, Yuxian Gu, Xin Cheng, Xun Wang, Si-Qing Chen, Li Dong, Wei Lu, Zhifang Sui, Benyou Wang, Wai Lam, and Furu Wei. Synthetic data (almost) from scratch: Generalized instruction tuning for language models, 2024.

[184] Jia Li, Ge Li, Zhuo Li, Zhi Jin, Xing Hu, Kechi Zhang, and Zhiyi Fu. Codeeditor: Learning to edit source code with pre-trained models. *ACM Transactions on Software Engineering and Methodology*, 32(6):1–22, 2023.

[185] Tomasz Lisowski. Top git hosting services for 2022, Dec 2021.

[186] S. Liu, S. Liu, W. Cai, S. Pujol, R. Kikinis, and D. Feng. Early diagnosis of alzheimer's disease with deep learning. In *2014 IEEE 11th International Symposium on Biomedical Imaging (ISBI)*, pages 1015–1018, April 2014.

[187] Yi Liu, Gelei Deng, Yuekang Li, Kailong Wang, Zihao Wang, Xiaofeng Wang, Tianwei Zhang, Yepang Liu, Haoyu Wang, Yan Zheng, and Yang Liu. Prompt injection attack against llm-integrated applications. March 2024.

[188] Yupei Liu, Yuqi Jia, Runpeng Geng, Jinyuan Jia, and Neil Zhenqiang Gong. Prompt injection attacks and defenses in llm-integrated applications. (arXiv:2310.12815), October 2023. arXiv:2310.12815 [cs].

[189] Yupei Liu, Yuqi Jia, Runpeng Geng, Jinyuan Jia, and Neil Zhenqiang Gong. Formalizing and benchmarking prompt injection attacks and defenses. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 1831–1847, 2024.

[190] Yiling Lou, Junjie Chen, Lingming Zhang, Dan Hao, and Lu Zhang. History-driven build failure fixing: how far are we? In *Proceedings of the 28th acm sigsoft international symposium on software testing and analysis*, pages 43–54, 2019.

[191] Z. Lubsen, A. Zaidman, and M. Pinzger. Using association rules to study the co-evolution of production test code. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 151–154, 2009.

[192] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005.

[193] Welder Pinheiro Luz, Gustavo Pinto, and Rodrigo Bonifácio. Building a collaborative culture: A grounded theory of well succeeded devops adoption in practice. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '18, New York, NY, USA, 2018. Association for Computing Machinery.

[194] L. E. Lwakatare, I. Crnkovic, and J. Bosch. Devops for ai – challenges in development of ai-enabled applications. In *2020 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 1–6, 2020.

[195] Lucy Ellen Lwakatare, Ivica Crnkovic, and Jan Bosch. Devops for ai–challenges in development of ai-enabled applications. In *2020 international conference on software, telecommunications and computer networks (SoftCOM)*, pages 1–6. IEEE, 2020.

[196] Lucy Ellen Lwakatare, Ivica Crnkovic, and Jan Bosch. Devops for ai – challenges in development of ai-enabled applications. In *2020 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, page 1–6. IEEE, Sep 2020.

[197] Lucy Ellen Lwakatare, Aiswarya Raj, Jan Bosch, Helena Holmström Olsson, and Ivica Crnkovic. *A Taxonomy of Software Engineering Challenges for Machine Learning Systems: An Empirical Investigation*, volume 355 of *Lecture Notes in Business Information Processing*, page 227–243. Springer International Publishing, 2019.

[198] Sucharita Maji, Nidhi Yadav, and Pranjal Gupta. Lgbtq+ in workplace: a systematic review and reconsideration. *Equality, Diversity and Inclusion: An International Journal*, 43(2):313–360, March 2024.

[199] Spyros Makridakis, Fotios Petropoulos, and Yanfei Kang. Large language models: Their success and impact. *Forecasting*, 5(3):536–549, August 2023.

[200] Cosmin Marsavina, Daniele Romano, and Andy Zaidman. Studying fine-grained co-evolution patterns of production and test code. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 195–204. IEEE, 2014.

[201] Douglas H. Martin and James R. Cordy. On the maintenance complexity of makefiles. In *Proceedings of the 7th International Workshop on Emerging Trends in Software Metrics - WETSoM '16*, page 50–56. ACM Press, 2016.

[202] Silverio Martínez-Fernández, Justus Bogner, Xavier Franch, Marc Oriol, Julien Siebert, Adam Trendowicz, Anna Maria Vollmer, and Stefan Wagner. Software engineering for ai-based systems: a survey. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(2):1–59, 2022.

[203] Mirjana Mazuran, Elisa Quintarelli, and Letizia Tanca. *Mining Tree-Based Frequent Patterns from XML*, volume 5822 of *Lecture Notes in Computer Science*, page 287–299. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[204] David W McDonald and Mark S Ackerman. Expertise recommender: a flexible recommendation system and architecture. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pages 231–240, 2000.

[205] Mary L McHugh. Interrater reliability: the kappa statistic. *Biochemia medica*, 22(3):276–282, 2012.

[206] Shane Mcintosh, Bram Adams, and Ahmed E. Hassan. The evolution of java build systems. *Empirical Softw. Engg.*, 17(4–5):578–608, August 2012.

[207] Shane McIntosh, Bram Adams, Thanh H.D. Nguyen, Yasutaka Kamei, and Ahmed E. Hassan. An empirical study of build maintenance effort. ICSE '11, page 141–150, New York, NY, USA, 2011. Association for Computing Machinery.

[208] Shane McIntosh, Bram Adams, Thanh HD Nguyen, Yasutaka Kamei, and Ahmed E Hassan. An empirical study of build maintenance effort. In *Proceedings of the 33rd international conference on software engineering*, pages 141–150, 2011.

[209] Paul David McNicholas, Thomas Brendan Murphy, and M O'Regan. Standardising the lift of an association rule. *Computational Statistics & Data Analysis*, 52(10):4712–4721, 2008.

[210] Sichen Meng, Xiaoyin Wang, Lu Zhang, and Hong Mei. A history-based matching approach to identification of framework evolution. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 353–363, 2012.

[211] Hrushikesh N. Mhaskar, Sergei V. Pereverzyev, and M. D. van der Walt. A deep learning approach to diabetic blood glucose prediction. *CoRR*, abs/1707.05828, 2017.

[212] Microsoft. Continuous integration with visual studio team services, 2021. accessed 08-31-2021.

[213] Microsoft Azure Machine Learning. Microsoft Azure Machine Learning. https://azure.microsoft.com/en-us/products/machine-learning, 2023. Accessed: 2023-11-09.

[214] Matthew B Miles and A Michael Huberman. *Qualitative data analysis: An expanded sourcebook*. sage, Thousand Oaks, CA, 1994.

[215] Shawn Minto and Gail C Murphy. Recommending emergent teams. In *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*, pages 5–5. IEEE, 2007.

[216] Thomas J. Misa. Gender bias in big data analysis. *Information & Culture*, 57:283 – 306, 2022.

[217] Audris Mockus and James D Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the 24th international conference on software engineering*, pages 503–512, 2002.

[218] Joao Eduardo Montandon, Luciana Lourdes Silva, and Marco Tulio Valente. Identifying experts in software libraries and frameworks among github users. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 276–287. IEEE, 2019.

[219] M. Mossienko. Automated cobol to java recycling. In *Seventh European Conference on Software Maintenance and Reengineering, 2003. Proceedings.*, pages 40–50, 2003.

[220] Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-González. Fixing dependency errors for python build reproducibility. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 439–451, 2021.

[221] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating github for engineered software projects. *Empirical Software Engineering*, 22(6):3219–3253, Dec 2017.

[222] Andrew Mutton, Mark Dras, Stephen Wan, and Robert Dale. GLEU: Automatic evaluation of sentence-level fluency. In Annie Zaenen and Antal van den Bosch, editors, *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 344–351, Prague, Czech Republic, June 2007. Association for Computational Linguistics.

[223] Nadia Nahar, Shurui Zhou, Grace Lewis, and Christian Kästner. Collaboration challenges in building ml-enabled systems: Communication, documentation, engineering, and process. (arXiv:2110.10234), Feb 2022. arXiv:2110.10234 [cs].

[224] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[225] Iulian Neamtiu, Jeffrey S Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, 2005.

[226] Neptune.ai, Mar 2021.

[227] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N Nguyen. Statistical learning approach for mining api usage mappings for code migration. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 457–468, 2014.

[228] Anh Tuan Nguyen, Zhaopeng Tu, and Tien N. Nguyen. Do contexts help in phrase-based, statistical source code migration? In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 155–165, 2016.

[229] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N. Nguyen. Exploring api embedding for api usages and applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 438–449, 2017.

[230] Jakob Nielsen. Response time limits: Article by jakob nielsen, 1993.

[231] Pat Niemeyer. j2swift, 2023. accessed 04-01-2023.

[232] NLPChina, February 2024.

[233] Nose. Nose documentation, 2023. Accessed: 2023-11-17.

[234] Ziad Obermeyer, Brian Powers, Christine Vogeli, and Sendhil Mullainathan. Dissecting racial bias in an algorithm used to manage the health of populations. 2019.

[235] MIT News Office. Technique improves the reasoning capabilities of large language models. https://news.mit.edu/2024/technique-improves-reasoning-capabilities-large-language-models-06 2024. [Online; accessed 19-August-2024].

[236] Michael Olan. Unit testing: Test early, test often. *Journal of Computing Sciences in Colleges - JCSC*, 19, Jan 2003.

[237] Parmy Olson. The algorithm that beats your bank manager. https://www.forbes.com/sites/parmyolson/2011/03/15/the-algorithm-thatbeats-your-bank-manage 2011. Accessed: 2020-12-29.

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

[239] Jesutofunmi A. Omiye, Jenna C. Lester, Simon Spichak, Veronica Rotemberg, and Roxana Daneshjou. Large language models propagate race-based medicine. *npj Digital Medicine*, 6(1):195, October 2023.

[240] OpenAI. Prompt engineering. https://platform.openai.com/docs/guides/prompt-engineering. Accessed: 2024-09-10.

[241] OpenAI. Gpt-4 technical report. (arXiv:2303.08774), March 2024. arXiv:2303.08774 [cs].

[242] OpenAI. Gpt-4o system card, August 2024.

[243] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. Understanding the effectiveness of large language models in code translation. *arXiv preprint arXiv:2308.03109*, 2023.

[244] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, page 311–318, USA, 2002. Association for Computational Linguistics.

[245] PatrickFarley. Prompt shields in azure ai content safety - azure ai services, August 2024.

[246] Fábio Perez and Ian Ribeiro. Ignore previous prompt: Attack techniques for language models. (arXiv:2211.09527), November 2022. arXiv:2211.09527 [cs].

[247] Hung Dang Phan, Anh Tuan Nguyen, Trong Duc Nguyen, and Tien N. Nguyen. Statistical migration of api usages. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 47–50, 2017.

[248] Gustavo Pinto, Fernando Castor, Rodrigo Bonifacio, and Marcel Rebouças. Work practices and challenges in continuous integration: A survey with travis ci users: Work practices and challenges: A survey with travis ci users. *Software: Practice and Experience*, 48(12):2223–2236, Dec 2018.

[249] Kaiser Pister, Dhruba Jyoti Paul, Patrick Brophy, and Ishan Joshi. Promptset: A programmer's prompting dataset. *arXiv preprint arXiv:2402.16932*, 2024.

[250] Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with" gradient descent" and beam search. *arXiv preprint arXiv:2305.03495*, 2023.

[251] PyGithub. Pygithub/pygithub.

[252] pytest. How to use unittest-based tests with pytest — pytest documentation, 2021. Accessed: 2021-08-27.

[253] Pytest. Pytest documentation, 2023. Accessed: 2023-11-17.

[254] PyTravisCI. Pytravisci documentation, 2021. Accessed: 2021-08-23.

[255] Lonnie Qin. English-spanish translation dataset, 2024.

[256] Quantilus. Why is machine learning important and how will it impact business?, 2020. Accessed: 2023-10-25.

[257] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners.

[258] Saima Rafi, W. Yu, and Muhammad Azeem Akbar. Rmdevops: A road map for improvement in devops activities in context of software organizations. *Proceedings of the Evaluation and Assessment in Software Engineering*, 2020.

[259] Akond Rahman, Amritanshu Agrawal, Rahul Krishna, and Alexander Sobran. Characterizing the influence of continuous integration. empirical results from 250+ open source and proprietary projects. *Proceedings of the 4th ACM SIGSOFT International Workshop on Software Analytics*, page 8–14, Nov 2018. arXiv: 1711.03933.

[260] Akond Rahman and Chris Parnin. Detecting and characterizing propagation of security weaknesses in puppet-based infrastructure management. *IEEE Transactions on Software Engineering*, pages 1–18, 2023.

[261] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. An empirical analysis of build failures in the continuous integration workflows of java-based open-source software. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, page 345–355. IEEE, May 2017.

[262] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. On the naturalness of buggy code. *Proceedings of the 38th International Conference on Software Engineering*, 2016.

[263] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019.

[264] Cedric Renggli, Frances Ann Hubis, Bojan Karlaš, Kevin Schawinski, Wentao Wu, and Ce Zhang. Ease.ml/ci and ease.ml/meter in action: Towards data management for statistical generalization. *Proc. VLDB Endow.*, 12(12):1962–1965, August 2019.

[265] Cedric Renggli, Bojan Karlaš, Bolin Ding, Feng Liu, Kevin Schawinski, Wentao Wu, and Ce Zhang. Continuous integration of machine learning models with ease. ml/ci: Towards a rigorous yet practical treatment. *Proceedings of Machine Learning and Systems*, 1:322–333, 2019.

[266] Stanford Social Innovation Review, 2024.

[267] Romain Robbes and David Röthlisberger. Using developer interaction data to compare expertise metrics. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 297–300. IEEE, 2013.

[268] Gregorio Robles, Jesús M. González-Barahona, Carlos Cervigón, Andrea Capiluppi, and Daniel Izquierdo-Cortázar. Estimating development effort in free/open source software projects by mining software repositories: a case study of openstack. In *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, page 222–231. ACM Press, 2014.

[269] Amanda Ross and Victor L. Willson. Paired samples t-test. pages 17–19. SensePublishers, Rotterdam, 2016.

[270] Amanda Ross and Victor L. Willson. *Paired Samples T-Test*, pages 17–19. SensePublishers, Rotterdam, 2017.

[271] Sippo Rossi, Alisia Marianne Michel, Raghava Rao Mukkamala, and Jason Bennett Thatcher. An early categorization of prompt injection attacks on large language models. (arXiv:2402.00898), January 2024. arXiv:2402.00898 [cs].

[272] Pooya Rostami Mazrae, Tom Mens, Mehdi Golzadeh, and Alexandre Decan. On the usage, co-usage and migration of ci/cd tools: A qualitative analysis. *Empirical Software Engineering*, 28(2):52, Mar 2023.

[273] Peter J. Rousseeuw and Mia Hubert. Robust statistics for outlier detection. *WIREs Data Mining and Knowledge Discovery*, 1(1):73–79, 2011.

[274] Baptiste Roziere, Marie-Anne Lachaux, Guillaume Lample, and Lowik Chanussot. Unsupervised translation of programming languages. *NeurIPS 2020*, page 21, 2020.

[275] A. Rutherford. *ANOVA and ANCOVA: A GLM Approach*. Wiley, 2011.

[276] Dhia Elhaq Rzig, Foyzul Hassan, Chetan Bansal, and Nachiappan Nagappan. Characterizing the usage of ci tools in ml projects. In *Proceedings of the 16th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '22, page 69–79, New York, NY, USA, 2022. Association for Computing Machinery.

[277] Dhia Elhaq Rzig, Foyzul Hassan, Chetan Bansal, and Nachiappan Nagappan. Characterizing the usage of ci tools in ml projects. In *Proceedings of the 16th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 69–79, 2022.

[278] Dhia Elhaq Rzig, Foyzul Hassan, and Marouane Kessentini. An empirical study on ml devops adoption trends, efforts, and benefits analysis. *Information and Software Technology*, 152:107037, 2022.

[279] Dhia Elhaq Rzig, Foyzul Hassan, and Marouane Kessentini. An empirical study on ml devops adoption trends, efforts, and benefits analysis. *Information and Software Technology*, 152:107037, Dec 2022.

[280] Pranab Sahoo, Ayush Kumar Singh, Sriparna Saha, Vinija Jain, Samrat Mondal, and Aman Chadha. A systematic survey of prompt engineering in large language models: Techniques and applications. (arXiv:2402.07927), February 2024. arXiv:2402.07927 [cs].

[281] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., USA, 1986.

[282] Mattia Samory, Indira Sen, Julian Kohne, Fabian Flöck, and Claudia Wagner. "call me sexist, but...": Revisiting sexism detection using psychological scales and adversarial samples. *Proceedings of the International AAAI Conference on Web and Social Media*, 15:573–584, May 2021.

[283] Maarten Sap, Dallas Card, Saadia Gabriel, Yejin Choi, and Noah A. Smith. The risk of racial bias in hate speech detection. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, page 1668–1678, Florence, Italy, 2019. Association for Computational Linguistics.

[284] Tomohiro Sawada, Daniel Paleka, Alexander Havrilla, Pranav Tadepalli, Paula Vidas, Alexander Kranias, John J. Nay, Kshitij Gupta, and Aran Komatsuzaki. Arb: Advanced reasoning benchmark for large language models. (arXiv:2307.13692), July 2023. arXiv:2307.13692 [cs].

[285] Timo Schick and Hinrich Schütze. True few-shot learning with prompts – a real-world perspective. (arXiv:2111.13440), November 2021. arXiv:2111.13440 [cs].

[286] David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-Francois Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. *Advances in neural information processing systems*, 28, 2015.

[287] Kathrin Sebler, Yao Rong, Emek Gozluklu, and Enkelejda Kasneci. Benchmarking large language models for math reasoning tasks. (arXiv:2408.10839), August 2024. arXiv:2408.10839 [cs].

[288] Abigail See, Peter J. Liu, and Christopher D. Manning. Get to the point: Summarization with pointer-generator networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1073–1083, Vancouver, Canada, July 2017. Association for Computational Linguistics.

[289] Pranab Kumar Sen. Estimates of the regression coefficient based on kendall's tau. *Journal of the American statistical association*, 63(324):1379–1389, 1968.

[290] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. Programmers' build errors: a case study (at google). In *Proceedings of the 36th International Conference on Software Engineering*, page 724–734. ACM, May 2014.

[291] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. Programmers' build errors: a case study (at google). In *Proceedings of the 36th International Conference on Software Engineering*, page 724–734. ACM, May 2014.

[292] Amazon Web Services. Amazon sagemaker – machine learning – amazon web services, 2021. accessed 08-28-2021.

[293] Amazon Web Services. Cloud computing services - amazon web services (aws). https://aws.amazon.com, 2023. Accessed: 2023-11-15.

[294] Tim Shaffer, Kyle Chard, and Douglas Thain. An empirical study of package dependencies and lifetimes in binder python containers. In *2021 IEEE 17th International Conference on eScience (eScience)*, pages 215–224. IEEE, 2021.

[295] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE access*, 5:3909–3943, 2017.

[296] Zhengxiang Shi and Aldo Lipani. Don't stop pretraining? make prompt-based fine-tuning powerful learner. *Advances in Neural Information Processing Systems*, 36:5827–5849, 2023.

[297] Raj Mani Shukla and John Cartlidge. Agileml: A machine learning project development pipeline incorporating active consumer engagement. In *2021 IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE)*, page 1–7, Brisbane, Australia, Dec 2021. IEEE.

[298] Eliezio Soares, Gustavo Sizilio, Jadson Santos, Daniel Alencar da Costa, and Uirá Kulesza. The effects of continuous integration on software development: a systematic literature review. *Empirical Softw. Engg.*, 27(3), may 2022.

[299] SonarQube. Code quality and code security | sonarqube, 2023.

[300] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, pages 908–911, New York, New York, USA, 2018. ACM Press.

[301] Charles Spearman. The proof and measurement of association between two things. 1961.

[302] Donna Spencer. *Card sorting: Designing usable categories*. Rosenfeld Media, New York, NY, 2009.

[303] Daniel Ståhl and Jan Bosch. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, 87:48–59, 2014.

[304] Sayma Sultana, London Ariel Cavaletto, and Amiangshu Bosu. Identifying the prevalence of gender biases among the computing organizations. (arXiv:2107.00212), July 2021. arXiv:2107.00212 [cs].

[305] Matúš Sulír and Jaroslav Porubän. A quantitative study of java software buildability. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools*, page 17–25. ACM, Nov 2016.

[306] Xuchen Suo. Signed-prompt: A new approach to prevent prompt injection attacks against llm-integrated applications. 2024.

[307] Saghar Talebipour, Yixue Zhao, Luka Dojcilović, Chenggang Li, and Nenad Medvidović. Ui test migration across mobile platforms. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 756–767, 2021.

[308] Daniel Teixeira, Ruben Pereira, Telmo Antonio Henriques, Miguel Silva, and João Faustino. A systematic literature review on devops capabilities and areas:. *International Journal of Human Capital and Information Technology Professionals*, 11(2):1–22, Apr 2020.

[309] Josh Terrell, Andrew Kofink, Justin Middleton, Clarissa Rainear, Emerson Murphy-Hill, Chris Parnin, and Jon Stallings. Gender differences and bias in open source: pull request acceptance of women versus men. *PeerJ Computer Science*, 3:e111, May 2017.

[310] Cedric Teyton, Jean-Remy Falleri, and Xavier Blanc. Automatic discovery of function mappings between similar libraries. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, page 192–201, Koblenz, Germany, October 2013. IEEE.

[311] Himanshu Thakur, Atishay Jain, Praneetha Vaddamanu, Paul Pu Liang, and Louis-Philippe Morency. Language models get a gender makeover: Mitigating gender bias with few-shot data interventions. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 340–351, Toronto, Canada, July 2023. Association for Computational Linguistics.

[312] Travis-CI, 2023.

[313] Travis-CI, 2023.

[314] TravisCI. Home – travis-ci_2022, Oct 2022.

[315] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. There and back again: Can you compile that snapshot?: There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 29(4):e1838, Apr 2017.

[316] Twine. Twine: A tool for creating interactive stories. https://twinery.org/, 2023. Accessed: 2023-11-15.

[317] Bogdan Vasilescu, Stef van Schuylenburg, Jules Wulms, Alexander Serebrenik, and Mark G.J. van den Brand. Continuous integration in a social-coding world: Empirical evidence from github. In *2014 IEEE International Conference on Software Maintenance and Evolution*, page 401–405. IEEE, Sep 2014.

[318] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, page 805–816. ACM, Aug 2015.

[319] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pages 805–816, 2015.

[320] Carmine Vassallo, Fabio Palomba, Alberto Bacchelli, and Harald C. Gall. Continuous code quality: are we (really) doing that? In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, page 790–795, Montpellier France, Sep 2018. ACM.

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

[322] Carmine Vassallo, Gerald Schermann, Fiorella Zampetti, Daniele Romano, Philipp Leitner, Andy Zaidman, Massimiliano Di Penta, and Sebastiano Panichella. A tale of ci build failures: An open source and a financial organization perspective. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, page 183–193. IEEE, Sep 2017.

[323] László Vidács and Martin Pinzger. Co-evolution analysis of production and test code by learning association rules of changes. In *2018 IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*, pages 31–36. IEEE, 2018.

[324] Mihaela Vorvoreanu, Lingyi Zhang, Yun-Han Huang, Claudia Hilderbrand, Zoe Steine-Hanson, and Margaret Burnett. From gender biases to gender-inclusive design: An empirical investigation. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, page 1–14, Glasgow Scotland Uk, May 2019. ACM.

[325] Abel Wang. What is configuration as code? | microsoft learn, 2019.

[326] Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. (arXiv:2305.04091), May 2023. arXiv:2305.04091 [cs].

[327] Xiaobo Wang, Huan Wang, and Chao Liu. Predicting co-changed software entities in the context of software evolution. In *2009 International Conference on Information Engineering and Computer Science*, pages 1–5. IEEE, 2009.

[328] Irene Weber. Large language models as software components: A taxonomy for llm-integrated applications, 2024.

[329] Wechat-Group. Wechat-group/wxjava, 2024.

[330] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 172–184, New York, NY, USA, 2023. Association for Computing Machinery.

[331] Jeremy White. How strangers got my email address from chatgpt's model. *The New York Times*, December 2023.

[332] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C. Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt. (arXiv:2302.11382), February 2023. arXiv:2302.11382 [cs].

[333] David Widder, Michael Hilton, Christian Kästner, and Bogdan Vasilescu. I'm leaving you, travis: A continuous integration breakup story. In *International Conference on Mining Software Repositories*, MSR, pages 165–169. ACM, 2018.

[334] David Gray Widder, Michael Hilton, Christian Kästner, and Bogdan Vasilescu. In *A conceptual replication of continuous integration pain points in the context of Travis CI*, page 647–658. ACM, Aug 2019.

[335] David Gray Widder, Michael C Hilton, Christian Kästner, and Bogdan Vasilescu. I'm leaving you, travis: A continuous integration breakup story. *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 165–169, 2018.

[336] Hyrum K Wright, Daniel Jasper, Manuel Klimek, Chandler Carruth, and Zhanyong Wan. Large-scale automated refactoring using clangmr. In *2013 IEEE International Conference on Software Maintenance*, pages 548–551. IEEE, 2013.

[337] Hong Wu, Lin Shi, Celia Chen, Qing Wang, and Barry Boehm. Maintenance effort estimation for open source software: A systematic literature review. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, page 32–43. IEEE, Oct 2016.

[338] Chunli Xie, Xia Wang, Cheng Qian, and Mengqi Wang. A source code similarity based on siamese neural network. *Applied Sciences*, 10(21):7519, Oct 2020.

[339] Bo Xu, Song Wu, Jiang Xiao, Hai Jin, Yingxi Zhang, Guoqiang Shi, Tingyu Lin, Jia Rao, Li Yi, and Jizhong Jiang. Sledge: Towards efficient live migration of docker containers. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 321–328. IEEE, 2020.

[340] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. (arXiv:2309.03409), April 2024. arXiv:2309.03409 [cs].

[341] Guang Yang, Yu Zhou, Xiang Chen, Xiangyu Zhang, Yiran Xu, Tingting Han, and Taolue Chen. A syntax-guided multi-task learning approach for turducken-style code generation. *arXiv preprint arXiv:2303.05061*, 2023.

[342] Hongwei Yao, Jian Lou, Zhan Qin, and Kui Ren. Promptcare: Prompt copyright protection by watermark injection and verification. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 845–861. IEEE, 2024.

[343] Jingwei Yi, Yueqi Xie, Bin Zhu, Emre Kiciman, Guangzhong Sun, Xing Xie, and Fangzhao Wu. Benchmarking and defending against indirect prompt injection attacks on large language models. (arXiv:2312.14197), March 2024. arXiv:2312.14197 [cs].

[344] L. Yin and V. Filkov. Team discussions and dynamics during devops tool adoptions in oss projects. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 697–708, 2020.

[345] Annie TT Ying, Gail C Murphy, Raymond Ng, and Mark C Chu-Carroll. Predicting source code changes by mining change history. *IEEE transactions on Software Engineering*, 30(9):574–586, 2004.

[346] Jiahao Yu, Yuhang Wu, Dong Shu, Mingyu Jin, and Xinyu Xing. Assessing prompt injection risks in 200+ custom gpts. (arXiv:2311.11538), November 2023. arXiv:2311.11538 [cs].

[347] M. Zaharia, Andrew Chen, A. Davidson, A. Ghodsi, S. Hong, A. Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Fen Xie, and Corey Zumar. Accelerating the machine learning lifecycle with mlflow. *IEEE Data Eng. Bull.*, 41:39–45, 2018.

[348] Andy Zaidman, Andy Zaidman, Bart Van Rompaey, Bart Van Rompaey, Arie van Deursen, Arie van Deursen, Serge Demeyer, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical software engineering : an international journal*, 16(3):325–364, 2011.

[349] J.D. Zamfirescu-Pereira, Richmond Y. Wong, Bjoern Hartmann, and Qian Yang. Why johnny can't prompt: How non-ai experts try (and fail) to design llm prompts. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, page 1–21, Hamburg Germany, April 2023. ACM.

[350] Fiorella Zampetti, Gabriele Bavota, Gerardo Canfora, and Massimiliano Di Penta. A study on the interplay between pull request review and continuous integration builds. In *2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER)*, pages 38–48. IEEE, 2019.

[351] Fiorella Zampetti, Salvatore Geremia, Gabriele Bavota, and Massimiliano Di Penta. Ci/cd pipelines evolution and restructuring: A qualitative and quantitative study. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 471–482. IEEE, 2021.

[352] 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*, 25:1095–1135, 2020.

[353] Yang Zhang, Bogdan Vasilescu, Huaimin Wang, and Vladimir Filkov. One size does not fit all: An empirical study of containerized continuous deployment workflows. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE, pages 295–306. ACM, 2018.

[354] Yang Zhang, Yiwen Wu, Tingting Chen, Tao Wang, Hui Liu, and Huaimin Wang. How do developers talk about github actions? evidence from online software development community. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[355] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. The impact of continuous integration on other software development practices: A large-scale empirical study. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, page 60–71, Oct 2017.

[356] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. The impact of continuous integration on other software development practices: a large-scale empirical study. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 60–71. IEEE, 2017.

[357] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging llm-as-a-judge with mt-bench and chatbot arena. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA, 2024. Curran Associates Inc.

[358] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. Mining api mapping for language migration. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 195–204, 2010.

[359] Jiaxin Zhu, Minghui Zhou, and Audris Mockus. Patterns of folder use and project popularity: a case study of github repositories. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '14*, page 1–4. ACM Press, 2014.

[360] Andy Zou, Zifan Wang, Nicholas Carlini, Milad Nasr, J Zico Kolter, and Matt Fredrikson. Universal and transferable adversarial attacks on aligned language models. 2023.