

Understanding the Evolution of GitHub Actions Automation Workflows

Pooya Rostami Mazrae

A dissertation submitted in fulfillment of the requirements of
the degree of *Docteur en Sciences*

May 6, 2026

Advisor

Prof. Dr. TOM MENS Université de Mons, Belgium

Jury members

Prof. Dr. ANDY ZAIDMAN Delft University of Technology, The Netherlands

Prof. Dr. MAIRIELI WESSEL Radboud University, The Netherlands

Dr. ALEXANDRE DECAN Université de Mons, Belgium

Prof. Dr. BRUNO QUOITIN Université de Mons, Belgium

Acknowledgements

As I receive my highest academic degree, I would like to express my sincere gratitude to all those who have supported me throughout my academic journey. Your guidance, encouragement, and support have been invaluable in helping me achieve this milestone.

First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Dr. *Tom Mens*, for his unwavering support, guidance, and mentorship throughout my research. His expertise and insights have been instrumental in shaping my academic journey, refining my ideas, and fostering my intellectual growth. I feel truly privileged to have had the opportunity to work with such a dedicated professor, who continuously motivated me to push the boundaries of my knowledge, skills, and expectations.

I would also like to extend my sincere thanks to Dr. *Alexandre Decan* for his valuable feedback and support during my research. His insightful comments and suggestions have greatly contributed to improving the quality of my work.

I am grateful to the members of my thesis committee, Prof. Dr. *Andy Zaidman*, Prof. Dr. *Mairieli Wessel*, and Prof. Dr. *Bruno Quoitin*, for their time and effort in reviewing my thesis and for their constructive feedback. I would like to extend a special thanks to Prof. Dr. *Mairieli Wessel* for offering me the opportunity to undertake a research stay at her institution and laboratory as part of my PhD journey.

I would also like to thank my colleagues at the University of Mons, especially those in the Software Engineering Lab, for their support and encouragement throughout my PhD. Your camaraderie and friendship made this journey more enjoyable and fulfilling.

Moreover, I would like to thank my friends, especially *Hassan*, *Aref*, *Mehdi*, *Mohammed*, *Hedi*, *Guillaume*, and *Lara*, whose presence in a country far from home has been a constant source of comfort and support. I have learned a

great deal from each of you, and your friendship has been invaluable in helping me navigate the challenges of living and studying abroad.

Finally, I would like to express my deepest gratitude to my parents, who have always been my pillars of support and encouragement. Your unwavering belief in me, along with your sacrifices, has made it possible for me to pursue my dreams and achieve this milestone. I am forever grateful for your love and support. This achievement is as much yours as it is mine, and I dedicate it to you. Last but not least, I would like to thank my brother for listening to my endless thoughts about my research, technology, and life, and for being a constant source of support and encouragement throughout my journey.

Abstract

Social coding platforms such as GitHub have profoundly transformed collaborative software development. Developers regularly contribute to numerous projects through mechanisms like pull requests, code reviews, and issue tracking. While this shift has fostered broader participation, it has also significantly increased the workload on maintainers, making repository management more complex and time intensive. To mitigate this growing burden, developers adopt workflow configuration tools to automate repetitive and labour-intensive tasks such as testing, building, deploying, and monitoring code quality. Over the past two decades, tools like Travis, CircleCI, and Jenkins have played key roles in automating Continuous Integration and Continuous Deployment/Delivery (CI/CD) pipelines. In 2019, GitHub introduced GitHub Actions to help developers automate their software development workflows. While GitHub Actions quickly became the dominant CI/CD solution in GitHub, research remains relatively limited regarding its impact on collaborative development practices and the challenges developers face when integrating it into their workflows.

This dissertation therefore investigates the role of GitHub Actions on the collaborative development process. Through empirical studies combining both quantitative and qualitative methods, it has two core objectives: (1) understanding the adoption and usage patterns of GitHub Actions, and (2) examining the evolutionary dynamics of GitHub Actions workflows. These studies reveal important patterns related to workflow creation, maintenance, and long-term sustainability. A large-scale analysis based on software repository mining reveals that, while GitHub Actions enables powerful automation, its integration into everyday development practices introduces new layers of complexity. Issues such as maintaining workflow correctness, adapting to changes in third-party Actions, along with the increasing complexity of managing configuration settings, emerge as recurring concerns. These findings shed light on

the nuanced trade-offs developers face when relying on automation at scale and point to broader implications for the sustainability and maintainability of CI/CD workflows.

Contents

1	Introduction	1
1.1	Collaborative Software Development	2
1.2	Continuous Integration and Delivery	3
1.3	Thesis Statement and Research Goals	4
1.4	Thesis Structure	5
2	Background	11
2.1	Version Control Systems	12
2.2	The Rise of Social Coding Platforms	14
2.3	Automation in Software Development Before GitHub Actions .	16
2.3.1	Continuous Integration and Delivery Tools	16
2.3.2	Limitations of Pre-GitHub Actions CI/CD	23
2.4	GitHub Actions	24
2.5	GitHub Apps and Automation Bots	29
2.6	Other GitHub Automation Mechanisms	30
2.7	Summary	31
3	Related Work	33
3.1	Empirical Research on the Benefits and Challenges of CI/CD .	34
3.2	Evolution of CI/CD Usage Prior to GitHub Actions	39
3.3	Adoption and Usage of GitHub Actions Workflows	41
3.4	Studies on Reusable Actions	44
3.5	Summary	46
4	CI/CD Tools Usage, Co-usage, and Migrations	49
4.1	Introduction	51
4.2	Methodology	52

4.2.1	Interview Questionnaire	52
4.2.2	Selection of Respondents	52
4.2.3	Conducting and Processing the Interviews	54
4.3	Why, How, and Which CI/CD Tools are Used	56
4.3.1	CI/CD Tools Usage	56
4.3.2	Main Reported Reasons for Using CI	59
4.3.3	Activities Being Automated by CI/CD Tools	61
4.3.4	Most Valuable Features of CI/CD Tools	64
4.3.5	Reported Shortcomings of CI/CD Tools	71
4.4	Co-Usage of CI/CD Tools and Motivations for Migration	77
4.4.1	Using Multiple CI/CD Tools Simultaneously	77
4.4.2	Software Projects Migrating to a Different CI/CD Tool	81
4.4.3	Difficulties in Carrying Out a CI/CD Migration	85
4.5	Discussion	87
4.5.1	On The Use of GitHub Actions	88
4.5.2	Open Source Nature of CI/CD Tools	90
4.5.3	Restrictions on the Free Tier of CI/CD Tools	91
4.5.4	Future of CI/CD Tools	93
4.5.5	On The Diversity of The CI/CD Landscape	95
4.5.6	Platform Choices, AI Integration, and Potential Migration Effects	96
4.6	Threats to Validity	97
4.7	Summary and Conclusions	98
5	GitHub Actions Usage	101
5.1	Introduction	102
5.2	Data Extraction	102
5.3	Characteristics of GitHub Repositories Using Workflows	103
5.4	Kinds of Workflows Being Automated	105
5.5	Most Frequent Jobs in Workflows	107
5.6	Automation Practices in Workflow Files	108
5.7	Characteristics of used Actions	110
5.8	Versioning Practices Used for GitHub Actions	112
5.9	Discussion	114
5.9.1	The GitHub Actions Ecosystem	114
5.9.2	Security Concerns	115
5.10	Threats to Validity	117
5.11	Summary and Conclusions	119
6	Methodology and Tooling for Analyzing Changes in Workflow	

Files	121
6.1 A Preliminary Study of GitHub Actions Workflow Changes . . .	122
6.1.1 Introduction	122
6.1.2 Motivating Example	122
6.1.3 Data Extraction	123
6.1.4 GitHub Actions Adoption by Repositories	125
6.1.5 Types of Coarse-grained Changes Affecting Workflows .	127
6.1.6 Temporal Patterns of Coarse-grained Workflow Changes	129
6.1.7 Types of Line-based Changes Affecting Workflows . . .	130
6.1.8 Temporal Patterns of Line-based Workflow Changes . . .	132
6.1.9 Threats to Validity	134
6.2 A Differencing Tool for GitHub Actions Workflows	135
6.2.1 Introduction	135
6.2.2 Motivating Example	136
6.2.3 GAWD	138
6.2.4 Example	140
6.2.5 Validation	143
6.2.6 Limitations	144
6.3 Summary and Conclusions	145
7 Evolution of GitHub Actions Workflows	147
7.1 Introduction	149
7.2 Data Extraction	150
7.3 Quantifying Workflow Change Frequency	152
7.4 Classification of workflow changes	156
7.5 Qualitative analysis of fine-grained workflow change types . . .	161
7.6 Impact of AI and other technological changes on workflow evolution	164
7.7 Quantitative analysis of changes to workflow entities	168
7.8 Discussion	173
7.8.1 CI/CD Configuration File Evolution Patterns	174
7.8.2 Conceptual Changes in Workflows	175
7.8.3 Evolution of Workflow Contents	176
7.9 Threats to Validity	178
7.10 Summary and Conclusions	180
8 Conclusion	181
8.1 Research Goals and Contributions	182
8.1.1 Adoption and Tool Selection of CI/CD Workflows	182
8.1.2 Large-Scale Usage of GitHub Actions in Practice	182

8.1.3	Evolutionary Characteristics of CI/CD Workflows	183
8.1.4	Maintenance Hotspots and Change-Prone Workflow Components	184
8.2	Limitations	185
8.2.1	Dataset Representativeness	185
8.2.2	Workflow Execution and Activity Uncertainty	185
8.2.3	Branch and History Scope	186
8.2.4	Tooling Constraints	186
8.2.5	Subjectivity in Qualitative Analysis	187
8.2.6	Generalizability of Findings	187
8.3	Future Research Perspectives	187
8.3.1	Linking Workflow Changes to Execution Behavior . . .	188
8.3.2	Detecting Workflow Smells and Anti-Patterns	188
8.3.3	Context-Aware Recommendation Systems for Workflow Design	189
8.3.4	Cross-Ecosystem and Longitudinal CI/CD Analyses . .	191
8.3.5	Automation Through AI Agents	193
8.3.6	Security of CI/CD Workflows in an Agentic Era	195
8.3.7	Improving Energy Efficiency and Workflow Performance	196
8.3.8	Supporting Cross-Workflow Maintenance and Reuse . .	197
	Bibliography	199
	Webography	215
	Appendix A	221

List of Acronyms

Various acronyms will be used throughout this dissertation to abbreviate frequent terms, some of which will even find usages across all sections. The expansion will be given at least on the first occurrence of each acronym in the text, but the following list of acronyms can be used as a reference if needed.

- A2A** Agent2Agent protocol. 193
- API** Application Programming Interface. 11, 24, 29–32, 43, 86, 88, 106, 118, 124
- AWS** Amazon Web Services. 19, 20, 56, 191, 192
- CaC** Configuration-as-Code. 18–20, 22, 31, 39, 149
- CD** Continuous Deployment/Delivery. 3, 21, 34, 35, 39, 40, 70, 87
- CDN** Content Delivery Network. 21
- CI** Continuous Integration. 3, 4, 16–19, 21, 34–41, 43, 46, 70, 94, 98, 112, 116, 125, 143, 174
- CI/CD** Continuous Integration and Continuous Deployment/Delivery. v–x, 1, 3–7, 11, 16–25, 28, 31–44, 46, 49–54, 56–100, 106, 110, 114–117, 119, 122, 123, 125, 126, 130, 135, 145, 149–151, 165, 173–175, 181–183, 185–188, 190–198, 221–223, 227
- CLA** Contributor License Agreement. 106
- CLI** Command Line Interface. 31, 32
- CVS** Concurrent Versions System. 12, 13

- DVCS** Distributed Version Control System. 13, 14
- GCP** Google Cloud Platform. 19, 20, 191, 192
- GDPR** General Data Protection Regulation. 54, 98
- IaC** Infrastructure-as-Code. 31
- JSON** JavaScript Object Notation. 30, 138
- LLM** Large Language Model. 8, 43, 44, 148, 164, 165, 167, 168, 174–176, 180, 184, 193–195
- MCP** Model Context Protocol. 193
- OSS** Open Source Software. 2, 15, 36, 51–53, 73, 88–93, 95
- OWASP** Open Web Application Security Project. 195
- RCS** Revision Control System. 12
- SCCS** Source Code Control System. 12
- SVN** Subversion. 13, 18
- VCS** Version Control System. 12

CHAPTER 1

Introduction

“I hate doubt, yet I am certain that doubt is the only way to approach anything worth believing in.”

Edward Teller

I started my PhD studies in 2021 after my master and bachelor studies in software engineering, in which I have learned about different software development methodologies, technologies, tools and their purposes.

My experience with GitHub as a student has shown me its importance in software development as the dominant platform to host both open-source and closed-source software projects. Furthermore, GitHub serves as a valuable data source for many empirical studies of current software development practices and their potential problems. Numerous researchers have leveraged GitHub data to extract insights and enhance processes in the field of collaborative software development and software maintenance.

I therefore decided to pursue doctoral studies in the field of software development analytics. To pursue the goals of my PhD, I conducted empirical analyses of development automation tools, with a primary focus on GitHub Actions, the CI/CD solution natively provided by GitHub. At the time I began my PhD, GitHub Actions was emerging as the dominant CI/CD platform on GitHub, yet it had received little empirical attention from the research community. This dissertation showcases the research I conducted throughout my PhD on this topic. It presents the methodology, goals, findings, research outcomes, and software artifacts that I have produced in this domain of research.

1.1. Collaborative Software Development

The large majority of today’s software is either open source or depends on it to a large extent. As reported by Costa *et al.* (2011), in response to a demand for higher-quality software products and faster time-to-market, Open Source Software (OSS) development is a continuous, highly distributed and collaborative endeavor. In such a setting, development teams often collaborate on software projects without geographical boundaries (Herbsleb, 2007). It is no longer expected for software projects to have all their developers working in the same location during the same office hours. To achieve this way of producing software, specific collaboration mechanisms have been devised such as version control systems (Conradi & Westfechtel, 1998; Koc & Tansel, 2011), issue and bug tracking (Bertram *et al.*, 2010), pull-based development (Gousios *et al.*, 2014), code reviewing (Bacchelli & Bird, 2013), commenting and the use of social communication channels to interact with other project contributors. Collaboration extends distributed software development from a primarily technical activity to an increasingly social phenomenon (Tsay *et al.*, 2014). Social phenomena such as communication with other developers, onboarding newcomers, and becoming core project contributors, play an essential role in collaborative development, and can become as critical as technical activities. They also come with their own challenges, such as social conflicts because of cultural differences, or language barriers or and community smells such as organizational silo, or lone wolves (Catolino *et al.*, 2021; Evtikhiev *et al.*, 2025; Holmström *et al.*, 2006; Suárez-Brieva *et al.*, 2026).

A multitude of development-related activities need to be carried out during collaborative software development: developing, debugging, analysing, designing, architecting, testing and reviewing code; quality and security analysis; packaging, releasing and deploying software distributions; communicating with end-users and other stakeholders; and so on. This increases the challenges for contributor communities to keep up with the rapid pace of producing and maintaining high-quality software releases. It requires the orchestrated use of a wide range of tools such as version control systems, software distribution managers, bug and issue trackers, vulnerability and dependency analysers, discussion forums, project management tools.

These tools tend to be integrated into so-called *social coding platforms* (e.g., GitLab, GitHub, BitBucket, Forgejo) that have revolutionized collaborative software development practices in the last two decades because they provide a higher degree of social transparency to all aspects of the development process (Dabbish *et al.*, 2012). Social coding platforms aim to reconcile the

technical and social aspects of software development in a single environment. They offer a seamless interface and experience for the project contributors to collaborate with their peers in an open and fully transparent workflow; and for users to contribute bug reports and feature requests through an issue tracking system. External contributors can propose code changes through a pull request mechanism, core software developers can push (*i.e.*, commit) their own code changes directly and accept and integrate the changes proposed by external contributors, and code review mechanisms allow code changes to be reviewed by other developers before they can be accepted (Gousios *et al.*, 2014).

1.2. Continuous Integration and Delivery

Continuous Integration (CI), Continuous Deployment/Delivery (CD) have become the cornerstone of collaborative software development practices. CI practices were introduced in the late 90s in the context of agile development and extreme programming methodologies. According to the agile manifesto principles by Beck *et al.* (2001), “our highest priority is to satisfy the customer through early and continuous delivery of valuable software”. In their seminal blog, Fowler and Foemmel (2000) presented CI as a way to increase the speed of software development while at the same time improving software quality and reducing the cost and risk of work integration among distributed teams. They outlined core CI practices for doing so, including frequent code commits, automated tests that run several times a day, frequent and fully reproducible builds, immediately fixing broken builds, and so on. CD practices, on the other hand, aim at automating the delivery and deployment of software products, following any change to their code (T. Chen *et al.*, 2021). Key elements of CD are the creation of feasible, small, and isolated software updates, that are automatically deployed immediately after completion of the development and testing (Savor *et al.*, 2016).

Many self-hosted CI/CD tools and cloud-based CI/CD services exist to automate the integration of code changes from multiple contributors into a centralised repository where automated builds, tests, quality checks and deployments are run. Popular examples of such CI/CD solutions are Jenkins, Travis, CircleCI and Azure DevOps. They have been the subject of much empirical research over the last decades. An excellent starting point is the systematic literature review by Soares *et al.* (2022), covering 106 research publications reporting on the use of CI/CD. This review aimed at identifying and interpreting empirical evidence regarding how CI/CD impacts software development. It revealed that CI/CD has many benefits for software projects. Besides the

aforementioned cost reduction, quality and productivity improvement, it also comes with a reduction of security risks, increased project transparency and predictability, greater confidence in the software product, easier locating and fixing of bugs, and improved team communication. CI can also benefit pull-based development by improving and accelerating the integration process.

CI/CD services have been integrated into social coding platforms and their widespread adoption have been seen in both industrial and open-source contexts (Zampetti *et al.*, 2021). With GitLab CI/CD, GitLab already featured built-in CI/CD capabilities since November 2012. BitBucket supported automation pipelines since May 2016. In response to this support for CI/CD in competing social coding platforms, GitHub started to integrate CI/CD through GitHub Actions in August 2019, and the product was released publicly in November 2019. As Beller *et al.* (2017) reports, before the release of GitHub Actions, Travis used to be the most popular CI/CD cloud service for GitHub repositories. A more recent study by Golzadeh *et al.* (2021c) provided quantitative evidence that GitHub Actions has become the dominant CI/CD solution for GitHub repositories since 2020.

1.3. Thesis Statement and Research Goals

At the start of my PhD studies, GitHub Actions, GitHub’s native CI/CD solution, was rapidly gaining popularity, as quantitatively observed by Golzadeh *et al.* (2021c). Unlike its predecessors, GitHub Actions is deeply integrated into the GitHub ecosystem and introduces reusable building blocks called ‘Actions’, which allow for modular, community-driven workflow composition. These unique characteristics together with the dominance of GitHub itself as a social platform position GitHub Actions as a potentially transformative tool for automation in software development. However, the underlying factors contributing to its success, patterns of usage, and emerging maintenance evolution remain insufficiently explored.

This thesis investigates the adoption, usage, and evolution of GitHub Actions workflows in collaborative software development. Through mixed-method empirical research, it examines how developers integrate and maintain workflow automation, analyzes the evolutionary dynamics of workflows including how they change over time and which components are most frequently modified, and explores what this reveals about maintenance and adaptation practices of this CI/CD tool.

The research is structured around the following four goals:

- Goal 1:** Clarifying the underlying factors that shape how CI/CD tools, particularly GitHub Actions, are selected, combined, and replaced in software projects, thereby supporting developers, teams, and organizations in making informed decisions about workflow automation in collaborative development settings.
- Goal 2:** Characterizing how GitHub Actions workflows are actually constructed and relied upon in real-world repositories, with the objective of revealing common conventions, implicit practices, and integration strategies that can guide developers and tool builders toward more consistent and effective automation design.
- Goal 3:** Establishing an empirical understanding of how workflow automation behaves as a long-lived software artifact, by uncovering the recurring workflow patterns, usage conventions, and integration strategies that characterize automation behavior in real-world software projects.
- Goal 4:** Exposing the maintenance pressures and structural weaknesses that make certain workflow components more volatile than others, and examining how the adoption of AI-assisted development tools influences the frequency and nature of changes in workflow configuration files, in order to inform improved tooling, practices, and ecosystem support aimed at reducing maintenance effort and failure risk in GitHub Actions pipelines.

1.4. Thesis Structure

This thesis comprises eight chapters of which the current Chapter is the introduction.

Chapter 2 introduces the key concepts underlying the GitHub Actions ecosystem, providing the foundational knowledge required for understanding this thesis. This chapter offers an overview of the main components of GitHub Actions including workflows and their reusable components, and explains how they are defined, configured, and used within software repositories. Some of the material presented in this chapter is based on the following book chapter that I co-authored:

- Wessel, M., Mens, T., Decan, A., **Rostami Mazrae, P.** (2023). The GitHub development workflow automation ecosystems. In Software

Ecosystems: Tooling and Analytics (pp. 183-214). Springer International Publishing.

DOI: 10.1007/978-3-031-36060-2_8

Chapter 3 examines and explores the current status of those research domains that are the most closely associated with this thesis. The chapter starts with an overview of workflow automation in GitHub. Following this introduction, the discussion extends to studies pertaining to workflow automation in GitHub through GitHub Actions.

Chapters 4 to 7 are based on peer-reviewed scientific articles published in journals, conferences, and workshops. Each article has at least one outcome in the form of supplementary materials such as datasets, surveys, software tools, and replication packages that have been produced for conducting the empirical research performed to achieve the research goals. Figure 1.1 visually presents for each chapter, the articles on which they are based in, and the contributions toward reaching the goal of the chapter.

Chapter 4 presents a qualitative empirical analysis of the usage, co-usage, and migration of CI/CD tools in GitHub. The aim of this chapter is to understand the reasons behind the success and failure of different automation tools in collaborative software development and how developers combine multiple automation tools, or migrate to different automation tools to better accommodate their needs based on 22 interviews with experienced software practitioners which covers goal one. The content of this chapter is based on a co-authored scientific journal article:

- **Rostami Mazrae, P.**, Mens, T., Golzadeh, M., Decan, A. (2023). On the usage, co-usage and migration of CI/CD tools: A qualitative analysis. *Empirical Software Engineering (EMSE)*, 28(2), p.52.
DOI: 10.1007/s10664-022-10285-5

Chapter 5 delves into the landscape of CI/CD usage on GitHub, exploring the evolution of how GitHub repositories adopt GitHub Actions. The findings focus on the ways which repositories adopt and utilize GitHub Actions over time. This chapter presents an empirical analysis based on a large-scale dataset of over 68,000 GitHub repositories, revealing that 43.9% of them use GitHub Actions workflows. It explores which types of workflows are commonly automated and uncovers the most frequent usage patterns, including the widespread reuse of existing Actions which is the aim of goal two. The analysis also highlights how workflows refer to these Actions and uncovers critical concerns related to versioning and security. Overall, the chapter offers a foundational understanding of GitHub Actions adoption trends and common

practices, serving as an essential step toward comprehending its broader impact on automation and collaboration in GitHub. The results presented in this chapter are based on the following conference article:

- Decan, A., Mens, T., **Rostami Mazrae, P.**, Golzadeh, M. (2022). On the use of GitHub Actions in software development repositories. In IEEE International Conference on Software Maintenance and Evolution (IC-SME), Limassol, Cyprus. **Distinguished Paper Award**. DOI: 10.1109/ICSME55016.2022.00029

Chapter 6 presents an empirical analysis of the evolution of GitHub Actions workflows, offering a detailed examination of how workflow automation practices change and grow over time within the GitHub ecosystem, guided by Lehman’s (Lehman, 1996) laws of software evolution. This chapter investigates both the structural and syntactic dimensions of workflow evolution to cover goal three.

Drawing from a dataset of over 22,000 repositories and 4 million historical workflow snapshots, the chapter identifies large-scale trends in workflow adoption and modification. It examines when and how workflows are introduced, changed, renamed, or removed, as well as the nature and frequency of line-level code changes. These insights provide empirical evidence that GitHub Actions workflows evolve continuously and grow in size, reflecting the dynamic and adaptive nature of CI/CD practices.

To enable this analysis, the chapter introduces *gawd*, a purpose-built syntactic differencing tool for GitHub Actions workflows. Unlike generic differencing tools, *gawd* is tailored to the YAML-based structure of GitHub Actions, accurately detecting additions, deletions, modifications, and movements of workflow components.

Together, these studies not only validate theoretical principles of software evolution in the context of CI/CD workflows, but also provide foundational tooling to support future research on automation practices.

The content of this chapter is based on the two following co-authored scientific articles:

- **Rostami Mazrae, P.**, Decan, A., Mens, T., Wessel, M. (2023). A Preliminary Study of GitHub Actions Workflow Changes. In Seminar on Advanced Techniques and Tools for Software Evolution (SATToSE), Salerno, Italy, In CEUR Workshop Proceedings, Volume 3483. URL: <https://ceur-ws.org/Vol-3483/paper8.pdf>
- **Rostami Mazrae, P.**, Decan, A., Mens, T. (2024). *gawd*: A Differencing Tool for GitHub Actions Workflows. In IEEE/ACM 21th Inter-

national Conference on Mining Software Repositories (MSR), Lisbon, Portugal.

DOI: 10.1145/3643991.3644873

Chapter 7 investigates the evolution of GitHub Actions workflows within the scope of **Goal 4**, characterizing how they change over time and identifying the workflow components that are most prone to change. Using the gawd differencing tool, the study reports a large-scale mixed-methods analysis that combines qualitative examination of workflow modifications with quantitative analysis across a broad set of GitHub repositories and their workflow histories. The qualitative investigation identifies seven recurring types of conceptual changes, including adjustments to task configurations, dependency updates, and permission modifications. The quantitative results reveal that workflow files are frequently updated, though most edits are small and localized. An additional analysis examines whether the emergence of Large Language Model (LLM) based coding tools or other major technological shifts has altered the frequency of workflow changes. However, the results provide no conclusive evidence that these tools have had a substantive impact on the frequency of changes of GitHub Actions workflows at file or content level. Overall, the findings indicate the need for improved tooling and automation support to handle frequent, small-scale modifications and to ease the maintenance burden of GitHub Actions pipelines in modern collaborative development environments. The results presented in this chapter are based on the following journal article:

- **Rostami Mazrae, P.**, Decan, A., Mens, T., Wessel, M. (2026). An Empirical Study of the Evolution of GitHub Actions Workflows. *Journal of Systems and Software (JSS)*, Volume 236.
DOI: 10.1016/j.jss.2026.112824

Finally, Chapter 8 summarizes the contributions made to support the thesis statement and reflects on the studies conducted and tools developed, along with a discussion of their limitations and potential threats to validity. It also outlines perspectives and directions for future research.

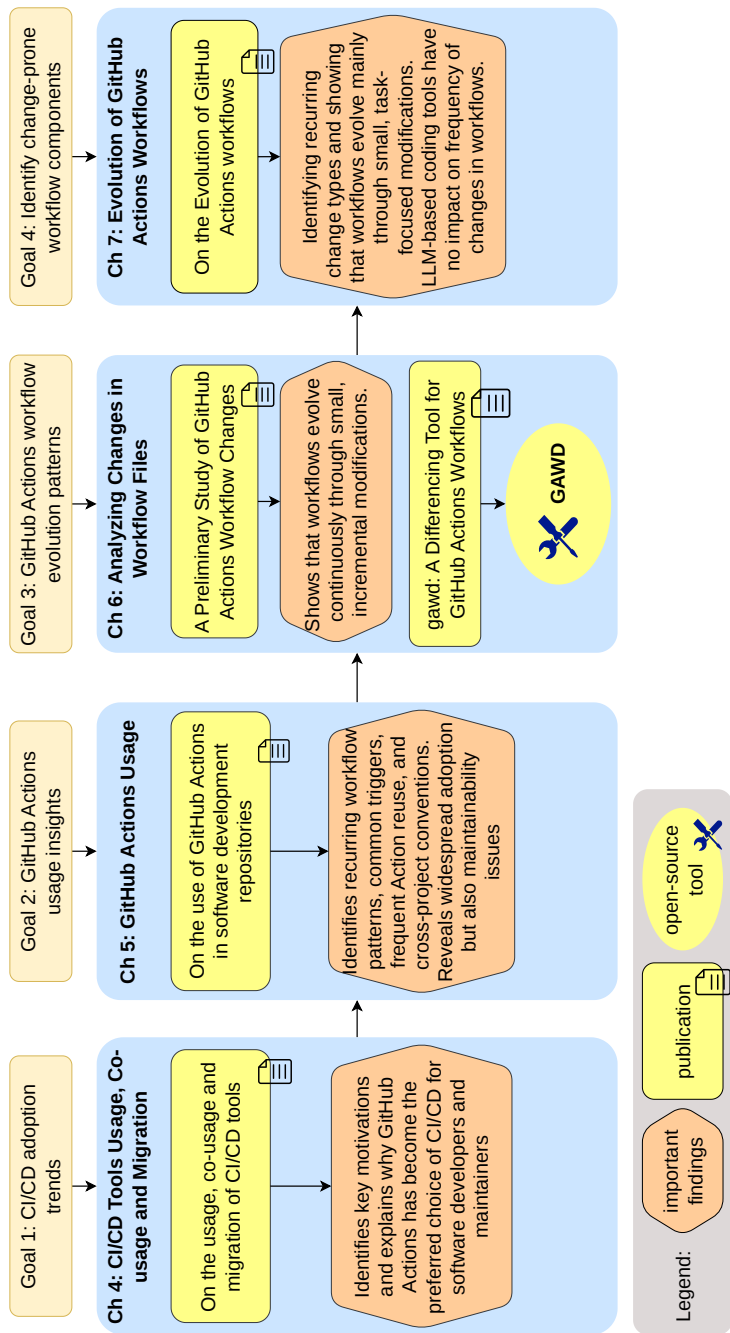


Figure 1.1: Structure, goals, and research outcomes of this dissertation.

Background

“It always seems impossible until it’s done.”

Nelson Mandela

In this chapter, we provide a broad overview of software development automation in the context of GitHub, establishing the conceptual and technical foundations required for the remainder of the dissertation. The chapter introduces key concepts, terminology, and mechanisms that are necessary for understanding the empirical analyses and discussions presented in subsequent chapters. We begin by outlining the historical rise of collaborative development platforms and the growing reliance on automation in software engineering practice. We then trace the evolution of automation approaches preceding the introduction of GitHub Actions, with particular attention to configuration-based CI/CD pipelines and their inherent limitations. This is followed by an in-depth discussion of GitHub Actions and GitHub Apps, which constitute GitHub’s primary automation mechanisms. Finally, we review complementary facilities such as webhooks, Application Programming Interfaces (APIs), and bots, thereby situating these mechanisms within the broader ecosystem of automation opportunities available to modern software development teams.

2.1. Version Control Systems

We begin by outlining the history and evolution of Version Control Systems (VCSs), which underpin modern software development and collaborative workflows. Version control provides the technical foundation for coordinating changes, managing parallel development, and maintaining traceability in multi-developer projects. Understanding the historical development of these systems helps explain both the emergence of contemporary collaborative platforms such as GitHub and the design choices that enable large-scale automation.

The earliest VCSs emerged in the early 1970s in response to practical coordination challenges in large software projects. The Source Code Control System (SCCS), developed at Bell Labs in 1972, was among the first widely adopted tools to systematically manage source code evolution (Rochkind, 1975). SCCS introduced core concepts that remain central to version control today, including the explicit recording of revisions, authorship, timestamps, and change rationales. While SCCS demonstrated the feasibility and value of systematic version control, its design reflected the technological constraints and prevailing assumptions about software development practices at the time.

In the early 1980s, the Revision Control System (RCS) built on the ideas introduced by SCCS and refined their implementation (Tichy, 1985). RCS improved the efficiency of revision storage and offered more flexible version labeling and branching mechanisms, making version control more accessible to a broader range of development contexts. Nevertheless, both SCCS and RCS remained fundamentally file-oriented and relied on centralized repositories with limited support for concurrent development, requiring developers to adapt their workflows to the tools' constraints.

As team-based development became more common in the late 1980s and 1990s, the limitations of file-oriented locking became increasingly problematic. To address this, Concurrent Versions System (CVS) was introduced as an extension of RCS, adding support for concurrent file editing and network-accessible repositories (Berliner, 1990). Rather than relying on exclusive locks, CVS allowed multiple developers to work on the same files at the same time, with conflicts handled through merging instead of being prevented upfront. This approach proved highly influential, and CVS became the dominant version control system for open source projects throughout much of the 1990s. However, it also inherited several limitations from RCS, including difficulties with directory renames, unreliable handling of atomic commits, and a still centralized architecture issues that would eventually motivate the development of the next generation of tools.

By the early 2000s, these limitations had become widely recognized, leading to efforts to replace CVS while retaining its centralized workflow model. Subversion (SVN), released in 2000, was designed explicitly as its successor, addressing many of its shortcomings, including the lack of atomic commits, limited support for renaming and moving files and directories, and rigid branching and tagging mechanisms (Pilato *et al.*, 2008). It introduced improvements such as atomic commits and more flexible versioning through cheap copies within the repository. As a result, SVN quickly replaced CVS as the dominant centralized version control system during the mid-2000s. However, it preserved the same centralized architecture, meaning developers still depended on constant server access and remained constrained when working offline or in highly distributed settings.

An earlier and important step toward distributed version control was BitKeeper, developed by Larry McVoy at BitMover and first publicly released in May 2000 (BitMover, 2000; McVoy, 1998). BitKeeper introduced a fully distributed model in which each developer maintained a complete local copy of the repository, enabling offline work and peer-to-peer synchronization without dependence on a central server. Its use for Linux kernel development in 2002 demonstrated the practical scalability benefits of distributed workflows compared to centralized systems, but its proprietary licensing quickly became a major point of contention within the open-source community. When BitMover ended free access in 2005, the Linux kernel project was suddenly left without a version control system, which directly motivated Linus Torvalds to create git (Spinellis, 2012). BitKeeper was eventually released as open-source software under the Apache 2.0 licence in 2016, although by that time git had already become the dominant standard.

As software projects grew in size and complexity particularly with the rise of open source development in the late 1990s and early 2000s, these limitations became increasingly apparent. Distributed Version Control System (DVCS) emerged to address scalability, availability, and collaboration challenges inherent in centralized models. Systems such as git (Spinellis, 2012) and Mercurial (O’Sullivan, 2009) gained traction by allowing developers to work with full local repositories, supporting offline development, flexible branching, and post-commit merging.

Alongside git and Mercurial, Canonical’s Bazaar emerged as another notable system in the DVCS landscape (Canonical, 2005). One of Bazaar’s distinguishing features was its flexibility: it supported both centralized and distributed workflows within the same tool, which made it easier for teams transitioning from systems like SVN. Its design emphasized usability and ac-

cessibility, and it saw adoption particularly within the Ubuntu and broader Canonical ecosystem. Despite these strengths, Bazaar did not achieve the same level of adoption as git, and Canonical discontinued active development in 2012. This outcome highlights how factors such as community adoption and ecosystem support can be as important as technical design in determining which tools succeed.

Another noteworthy system from this period was Darcs, first developed by David Roundy in June 2002 and publicly released in April 2003 (Roundy, 2005). Unlike snapshot-based systems such as git or Mercurial, Darcs represented repositories in terms of patches, treating them as first-class entities and modeling a repository as a partially ordered set of changes rather than a linear sequence of snapshots. This patch-theory approach enabled fine-grained cherry-picking and merging without the need for explicit branching commands, resulting in a more conceptually lightweight workflow. Darcs gained particular attention within the Haskell community due to its implementation in that language, but its merge algorithms exhibited exponential worst-case complexity, which limited its scalability and adoption in large projects. Like Bazaar, Darcs illustrates that technical innovation alone is not sufficient for widespread adoption: despite its novel theoretical foundations, it could not compete with the community momentum and tooling ecosystem that ultimately formed around git.

Over time, git achieved widespread adoption, driven by its performance characteristics, expressive workflow model, and strong community support. Its dominance was further reinforced by integration into major development platforms, where it became the de facto standard for both open source and commercial projects. More broadly, the evolution from early centralized systems to modern DVCSs reflects a shift toward tools that accommodate natural collaborative practices, rather than constraining them. Design limitations in early systems such as long-term locking, coarse-grained versioning, and limited merging support directly informed the development of later approaches that emphasized flexibility, decentralization, and automation (Rochkind, 2025).

2.2. The Rise of Social Coding Platforms

Building on the capabilities of DVCS, a new generation of collaborative development platforms emerged that combined version control with social and organizational features. Among these platforms, GitHub, founded in 2008, rapidly established itself as the dominant environment for social coding (Dabish *et al.*, 2012). Importantly, GitHub is not a version control system itself;

rather, it is a platform layered on top of git that augments version control with mechanisms for coordination, communication, and governance.

The emergence of platforms such as GitHub was made possible by a confluence of infrastructural and economic conditions that matured during the mid-2000s. The dramatic reduction in storage costs allowed platforms to host an essentially unlimited number of repositories at negligible marginal cost, removing what had previously been a significant barrier to public code hosting (Kurose & Ross, 2013). Simultaneously, advances in web scalability, including the widespread adoption of distributed caching, load balancing, and database sharding, enabled platforms to serve millions of concurrent users without degradation in responsiveness (Abbott & Fisher, 2015). The rapid expansion of broadband internet access and network bandwidth meant that transferring large repository histories became practical for ordinary developers, not merely those with access to institutional infrastructure. The concurrent rise of cloud computing further reduced the capital investment required to operate large-scale web services, allowing small teams to build and maintain platforms that could grow elastically with demand (Armbrust *et al.*, 2010). Beyond infrastructure, GitHub’s success was substantially shaped by its social design: by making repositories, contributions, and development activity publicly visible, it created reputational incentives that encouraged participation and transparency (Dabbish *et al.*, 2012). The pull request model in particular lowered the coordination cost of accepting outside contributions, enabling a contribution model that scaled to thousands of participants without requiring centralized oversight (Gousios *et al.*, 2014). Taken together, these technical, economic, and social factors produced conditions in which a platform like GitHub could not only exist but thrive as the central hub of open source collaboration.

GitHub’s success stems from its integration of Git-based repositories with a rich set of collaborative features, including pull requests, issue tracking, code reviews, wikis, and project management facilities (Gousios *et al.*, 2014). These features transformed GitHub from a code hosting service into a socio-technical hub where developers can coordinate work, negotiate changes, and make development activities visible to a broader community (Costa *et al.*, 2011). This integration reshaped OSS development practices by lowering barriers to contribution and enabling large, loosely coupled groups of developers to collaborate effectively (Herbsleb, 2007).

As projects hosted on such platforms grew in size and complexity, the need for automation became increasingly apparent. Routine activities such as building, testing, linting, and deployment occurred with growing frequency and

became impractical to perform manually without introducing delays or errors. This shift created fertile ground for CI/CD tools, which gradually evolved into integral components of modern software engineering workflows.

2.3. Automation in Software Development Before GitHub Actions

Automation in software engineering predates GitHub, but it became particularly important in the context of collaborative software development platforms. The central idea was to move repetitive tasks away from manual execution and into scripted pipelines that could be executed automatically in response to development activities.

2.3.1 Continuous Integration and Delivery Tools

By the early 2010s, CI/CD pipelines had become a widely accepted cornerstone of modern software engineering practice, building on more than a decade of prior development (Duvall *et al.*, 2007). Early CI systems such as Cruise Control, introduced in the early 2000s, demonstrated the feasibility and benefits of automated builds and tests triggered by source code changes. Subsequent tools expanded on these ideas by improving usability, scalability, and integration with emerging development workflows.

Within this evolving landscape, tools such as Jenkins, Travis, and CircleCI gained prominence by offering accessible automation for build and test execution in response to commits, pull requests, or scheduled events. These tools are not intended to represent the entirety of the CI/CD ecosystem; rather, they are discussed as illustrative examples of distinct stages in the evolution of CI/CD tooling. Jenkins exemplifies extensible, self-hosted CI servers, while Travis and CircleCI reflect the shift toward hosted and cloud-based automation services that lowered operational barriers for development teams.

Given the breadth and diversity of CI/CD tools developed over the past two decades, a comprehensive survey is beyond the scope of this dissertation. Instead, we focus on representative systems that highlight key design trends such as automation triggering, configuration-as-code, and integration with collaborative development platforms that directly inform the emergence of platform-integrated solutions. To situate these tools historically and clarify the relative recency of GitHub Actions, we complement this discussion with a timeline illustrating the evolution of CI/CD systems introduced over time which can be seen in Figure 2.1. These CI/CDs were selected based on their

historical significance and being reported by participants in interviews presented in Chapter 4. In the following text, we provide a brief overview of some of these tools which have been reported to have at least two mentions by the interviewees in Chapter 4, highlighting its contributions and limitations in the context of CI/CD evolution.

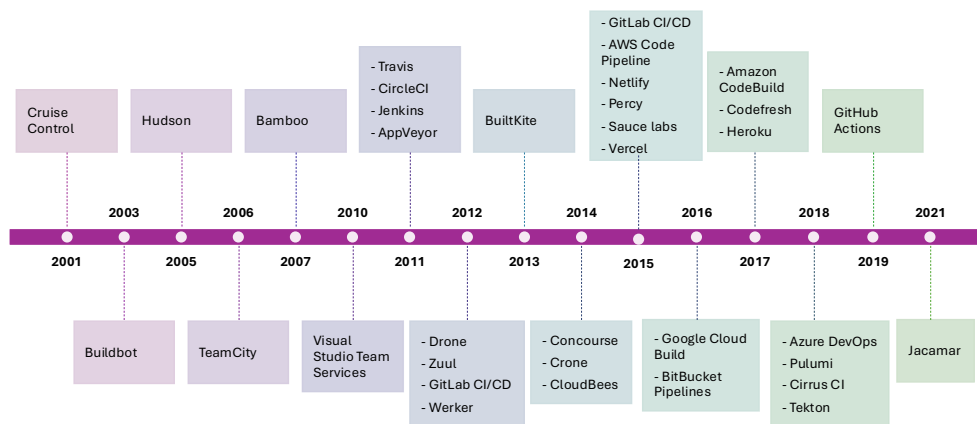


Figure 2.1: Introduction timeline of CI/CD tools reported in this thesis.

Cruise Control was one of the earliest CI services which was introduced in 2001, originally developed within ThoughtWorks to automate the repeated build and test cycles of software development and thereby improve feedback loops and code quality (CircleCI, 2023a; CruiseControl Project, 2001). Initially maintained as a proprietary tool, Cruise Control was later released as open source, enabling wider adoption and extension by the community. Written in Java, it offered a flexible plugin architecture, support for multiple version control systems, and a dashboard for monitoring build status, which helped establish many of the conventions later seen in CI/CD systems. Cruise Control's influence persisted even as newer tools emerged, and it is regarded as a foundational project in the history of CI practices.

Hudson was an early open-source CI service originally developed at Sun Microsystems in 2005 to automate the process of building and testing software in response to source code changes (Kawaguchi, 2007). Written in Java and running in servlet containers such as Apache Tomcat, Hudson supported a variety of version control systems and build tools, enabling developers to define automated jobs that monitored code commits, executed builds, and reported results through a web interface (Eclipse Foundation, 2025). The tool gained popularity as an alternative to Cruise Control and had been widely adopted

across organizations for CI tasks (Kawaguchi, 2007). However, development of Hudson declined after a community and governance dispute with Oracle led to the creation of the Jenkins project as a fork of Hudson. Hudson continued under the Eclipse Foundation for a time but was eventually archived and is now considered discontinued in favor of more actively maintained CI/CD solutions.

TeamCity is a CI/CD service developed by JetBrains in 2006 to automate build, test, and deployment processes across diverse software stacks (JetBrains, 2026). It monitors version control systems for changes and executes pipelines on distributed build agents, providing rapid feedback through detailed build and test reports. TeamCity supports parallel execution, customizable triggers, and agent pools, making it suitable for complex and enterprise-scale workflows (JetBrains, 2025a, 2025b). More recent features, such as TeamCity Pipelines, enable pipeline definition using both YAML and visual configuration, combining Configuration-as-Code (CaC) with flexible pipeline management.

Bamboo is a CI/CD service that automates build, test, and deployment processes through integrated workflows spanning from code commits to production releases in 2007 (Atlassian, 2025a). Bamboo enables teams to create multi-stage build plans with triggers that respond to version control activity, execute automated tests in parallel, and manage deployments with environment-specific controls and permissions. It offers deep integration with other Atlassian products such as BitBucket and Jira, providing traceability from feature requests through to deployment and streamlining the DevOps lifecycle (Curate Partners, 2025). Bamboo's use of distributed build agents allows scalable execution of jobs across platforms, and support for multiple source control systems (*e.g.*, git, Mercurial, SVN) facilitates adoption in heterogeneous development environments.

Jenkins, the community-driven successor to Hudson, introduced in 2011, has its core strength in high extensibility since it offers a modular architecture built around hundreds of plugins that support virtually any programming language, build system, testing framework, or deployment environment (Hilton *et al.*, 2016). This flexibility allowed organizations to design highly customized CI/CD pipelines long before cloud-native automation became mainstream.

Jenkins also supported both self-hosted infrastructure and distributed build environments, enabling large-scale enterprises to run jobs across multiple agents. This made it particularly suitable for complex, multi-language, and multi-platform projects. Despite its flexibility, the use of Jenkins as a CI/CD service typically required substantial configuration effort and ongoing operational maintenance, as teams were responsible for managing plugins, infrastructure, and upgrades. In response to these challenges, a range

of platform-integrated CI/CD solutions emerged that sought to reduce operational overhead through tighter integration with their surrounding ecosystems and declarative CaC workflows. Examples include GitLab CI/CD, Azure DevOps, Bitbucket Pipelines, and later, GitHub Actions, each offering differing trade-offs in terms of integration depth, execution models, and workflow expressiveness (Kinsman *et al.*, 2021).

Travis, which was also introduced in 2011, played a pivotal role in democratizing CI/CD for open-source projects. Unlike Jenkins, Travis was offered as a fully managed cloud CI/CD service, removing the need for developers to manage build servers or plugins (Travis CI, 2021, 2023a). Unlike Jenkins, which required self-hosted infrastructure and ongoing maintenance, Travis eliminated the need for developers to manage build servers or plugins, significantly lowering the operational barrier to entry (Atlassian, 2023). Its tight integration with GitHub repositories including automatic build triggers on pull requests made it the default CI solution for a large portion of the open-source ecosystem during the early 2010s (GitHub Docs, 2023).

Configuration in Travis was intentionally minimalistic, relying on a single `.travis.yml` file in which developers could declaratively specify language runtimes, dependency installation steps, and test commands. This simplicity lowered adoption barriers and helped shift CI/CD practices from specialized organizational setups toward widespread use by individual contributors and small teams (Travis CI, 2023b). In this sense, Travis can be seen as a precursor to the developer-centric, repository-native automation model later embraced by GitHub Actions.

CircleCI introduced a more cloud-native and performance-oriented approach to CI/CD in 2011 by emphasizing scalability, parallelism, and fast feedback cycles (CircleCI, 2023b, 2023d). From its early design, CircleCI supported isolated execution environments based on containers and virtual machines, enabling reproducible and highly parallelized build workflows. These features allowed teams to significantly reduce build times while maintaining consistent execution environments across runs.

CircleCI further provides advanced workflow orchestration, dependency caching, and deployment pipelines integrated with major cloud service providers, including Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure (CircleCI Documentation, 2023). Its strong focus on performance and developer experience made it particularly attractive for teams developing cloud-native and microservices-based systems (CircleCI, 2023c). In this respect, CircleCI can be viewed as an early adopter of flexible, event-driven, YAML-based automation concepts that were later popularized

by repository-native solutions such as GitHub Actions.

AppVeyor, introduced in 2011, is a hosted CI/CD service that automates the build, test, and deployment process for software projects across multiple platforms, with particular strength in supporting Windows environments alongside Linux and macOS (Postmake, 2025). Developers can configure AppVeyor pipelines using a YAML file stored in the repository or through the web UI, and builds are executed in clean, isolated virtual machines to ensure reproducibility and reliability. The service integrates with major version control hosting providers such as GitHub, GitLab CI/CD, and BitBucket, enabling build status reporting, artifact management, and deployment actions directly in response to repository events. Beyond hosted CI/CD, AppVeyor also offers a self-hosted server option that can be deployed on cloud or on-premise infrastructure, broadening its applicability to teams with specific infrastructure requirements. Due to its native support for .NET tooling and extensive integration options, AppVeyor remains a viable option for projects, especially those targeting Windows-centric stacks, that require flexible and cross-platform automation outside the context of integrated platform CI/CD services.

Drone is an open-source, container-native CI/CD service which was developed in 2012 to leverage Docker as the primary execution environment for build pipelines. Drone’s design treats each pipeline step as an isolated container, which simplifies environment setup, increases reproducibility, and enables easy scaling across distributed systems (Drone.io, 2025a, 2025b). Configuration is defined declaratively using a `.drone.yml` file within the repository, aligning with modern CaC practices and promoting versioned pipeline definitions alongside application code. Drone integrates with multiple version control systems, including GitHub, GitLab, and BitBucket, and supports parallel execution, extensible plugins, and flexible deployment strategies across cloud and on-premises infrastructures. Its lightweight architecture and Docker-centric model have made Drone popular with teams seeking a portable and extensible CI/CD solution outside tightly coupled platform services.

GitLab CI/CD provides a natively integrated CI/CD solution embedded directly within the GitLab platform in 2015, positioning automation as a first-class feature of collaborative software development (GitLab, 2023a). Automation pipelines are defined using a YAML configuration file (`.gitlab-ci.yml`) stored alongside the source code, enabling version-controlled, reproducible workflows that evolve together with the project. Pipeline execution is handled by GitLab Runners, which can be deployed on self-managed infrastructure or on major cloud platforms such as AWS, GCP, and Microsoft Azure, as well as container orchestration environments like Kubernetes.

By tightly integrating CI/CD with repository management, issue tracking, and code review, GitLab reduced adoption barriers and operational complexity compared to externally hosted CI services (GitLab, 2023b). This platform-centric approach intensified competition among collaborative development environments and formed an important backdrop for the later emergence of repository-native automation solutions such as GitHub Actions in GitHub.

Bitbucket Pipelines is the integrated CI/CD service built into BitBucket Cloud, introduced in 2016, enabling teams to automate building, testing, and deploying code directly where their repositories reside. It uses a YAML-based configuration file stored in the repository to define workflows that execute automatically on code changes, eliminating the need for separate CI servers and minimizing the operational overhead of managing build infrastructure (Atlassian, 2025b). The service supports container-based execution, parallel builds, artifact caching, and a wide range of third-party integrations through Pipes, which simplify connections to external services such as test platforms and deployment tools. Recent enhancements such as shared pipeline configurations and workspace-level dynamic pipelines further increase flexibility and governance, allowing organizations to standardize CI/CD workflows and inject custom logic at runtime across multiple repositories (Atlassian, 2025c, 2025d). Overall, Bitbucket Pipelines provides a scalable, integrated CI/CD solution within the Atlassian ecosystem that closely ties automation to repository activity and toolchain integration.

Netlify, introduced in 2015, is a cloud-based development platform that provides an integrated CI/CD pipeline designed primarily for web and frontend applications. By connecting a git repository to a Netlify site, developers can trigger automatic builds and deployments on every push, enabling seamless CD without requiring traditional CI server configuration or infrastructure management (Netlify, 2025a, 2025b). Netlify's CI/CD infrastructure, branded as Netlify Build, automates tasks such as installing dependencies, running build commands, and deploying artifacts to a global Content Delivery Network (CDN), while also generating preview environments for branches and pull requests to support early feedback and collaboration. Unlike many general-purpose CI/CD solutions, Netlify combines build, deployment, previewing, and CDN delivery into a single workflow, streamlining the development lifecycle for modern web projects and reducing operational burden for teams.

Azure DevOps, introduced in 2018, provides a comprehensive suite of DevOps services, including Azure Pipelines, its CI/CD solution. Azure Pipelines enables teams to automate the build, test, and deployment process for applications written in any language to diverse targets, including cloud platforms,

Table 2.1: CI/CD tools having been discussed in Section 2.3.

CI/CD tool	cloud based	self hosted	open source	date
Cruise Control		✓	†	Mar 2001
Hudson		✓	‡	Feb 2005
Bamboo		✓		Feb 2007
Jenkins		✓	✓	Feb 2011
CircleCI	✓	✓		Sep 2011
Travis	✓			Nov 2011
AppVeyor	✓			Nov 2011
GitLab CI/CD	✓	✓	✓	Nov 2012
Drone	✓	✓	✓	July 2014
Netlify	✓	✓	✓	Apr 2015
Bitbucket Pipelines	✓			May 2016
TeamCity	✓	✓		Oct 2016
Azure DevOps	✓	✓		Oct 2018
GitHub Actions	✓			Nov 2019

† Cruise Control was not open source when it was being commercialised by ThoughtWorks. The tool became open source after the company stopped maintaining it in March 2001 (PCQuest, 2025).

‡ Hudson was originally released as open source by Sun, but was commercialised when Oracle acquired Sun by 2011.

containers, and virtual machines, using YAML-defined workflows that can be version controlled alongside application code (Microsoft Azure, 2025; Microsoft DevOps Blog, 2025). It supports parallel builds across multiple operating systems, namely Linux, macOS, Windows, and integrates with a wide range of source repositories, such as Azure Repos, GitHub, GitLab, and BitBucket. In addition to broad platform support, Azure Pipelines offers container and Kubernetes deployment capabilities, artifact caching, and extensibility through a marketplace of extensions, making it suitable for both enterprise and open-source CI/CD practices.

Together, these platforms established fundamental CI/CD practices and popularized the principle of CaC, providing the conceptual and technical groundwork upon which GitHub Actions would later build. Table 2.1 summarizes the characteristics of the CI/CD tools introduced above, including their hosting models, open-source status, release dates.

2.3.2 Limitations of Pre-GitHub Actions CI/CD

Despite their success and widespread adoption, pre-GitHub Actions CI/CD systems exhibited three structural limitations when used in the context of projects hosted on GitHub, particularly as development practices evolved. Although platform-integrated alternatives such as GitLab CI/CD had already addressed many of these issues within their own ecosystems, adopting such solutions typically required migrating repositories away from GitHub.

For projects that remained on GitHub, CI/CD tooling generally operated as an external service. Repositories were therefore required to establish and maintain connections through webhooks, authentication tokens, or third-party integrations. These configuration steps introduced additional setup complexity and were prone to misconfiguration, especially as projects evolved or dependencies changed. For many teams particularly open-source projects with distributed or volunteer contributors this additional operational overhead created practical barriers to adopting and sustaining automated workflows.

GitHub's position as the de facto hosting platform for open-source development further reinforced this situation. The visibility, community reach, and ecosystem effects associated with GitHub often made migration to alternative platforms unattractive, even when those platforms offered more tightly integrated CI/CD solutions. As a result, many GitHub-hosted projects continued to rely on externally integrated automation tools despite the growing limitations of this model.

Economic constraints further amplified these challenges. The free-tier offerings of CI/CD services such as Travis, CircleCI, and similar platforms were frequently restrictive. While these services were initially generous toward open-source projects, limitations on compute resources, build concurrency, and job duration became increasingly pronounced over time as pricing models evolved. Private repositories faced even stricter constraints, making comprehensive CI/CD automation financially challenging for smaller teams, academic projects, or hobbyist efforts.

Another limitation concerned the lack of deep integration with GitHub's native development workflows. Although external CI/CD services could respond to GitHub events such as pushes or pull requests, their interaction with the broader GitHub ecosystem remained indirect. Integrating test results into pull request reviews, surfacing build metadata within GitHub's interface, or enforcing repository-level policies typically required additional configuration or custom tooling. Consequently, CI/CD often functioned as an external add-on rather than as a first-class component of the development process.

As collaborative development increasingly centered around GitHub fea-

tures such as pull requests, code reviews, issue management, and dependency updates, these shortcomings became more visible. This contrast was sharpened by the fact that major competing platforms including Azure DevOps, Bitbucket Pipelines, and GitLab CI/CD had already introduced tightly integrated, repository-native CI/CD solutions within their respective ecosystems.

The growing availability of such integrated automation on competing platforms underscored both the limitations of externally integrated CI/CD services for GitHub-hosted projects and the strategic importance of native automation capabilities. These dynamics created a clear opportunity for a solution that would embed automation directly within GitHub repositories, leverage GitHub’s user interface and APIs, and reduce the operational burden on maintainers.

It is within this combined historical and competitive context that GitHub Actions emerged as a unifying, repository-native automation platform, aligning GitHub’s collaboration model with the automation capabilities already offered by its major platform competitors.

2.4. GitHub Actions

GitHub has introduced GitHub Actions as a way to enable the specification and execution of workflows to automate activities in GitHub repositories. The product was officially released to the public in November 2019 integrating a fully-featured CI/CD service, answering the high demand of GitHub users to provide CI/CD support similar to what was already available in competing social coding platforms such as GitLab and Bitbucket (Dabbish *et al.*, 2012).

Since its introduction, GitHub Actions has become the dominant CI/CD service on GitHub according to a quantitative study by Golzadeh *et al.* (2021c), based on a dataset of more than 90K GitHub repositories. Figure 2.2 provides a historical overview of CI/CD usage in those repositories, starting from the first observation of Travis usage in the dataset in June 2011. Initially, GitHub repositories primarily used Travis as a CI/CD service. Over time, other CI/CD solutions were used more frequently, but Travis remained the dominant CI/CD. When GitHub Actions entered the CI/CD landscape, it overtook the other CI/CD solutions in popularity in less than 18 months after its introduction.

Our study in Chapter 4 complemented this quantitative analysis by qualitative interviews to understand the reasons behind GitHub Actions becoming the dominant CI/CD tool in GitHub, as well as why project maintainers decided to migrate to GitHub Actions primarily. The main reported reasons were the seamless integration into GitHub, the ease of use, and great support for

its reusable Actions components.

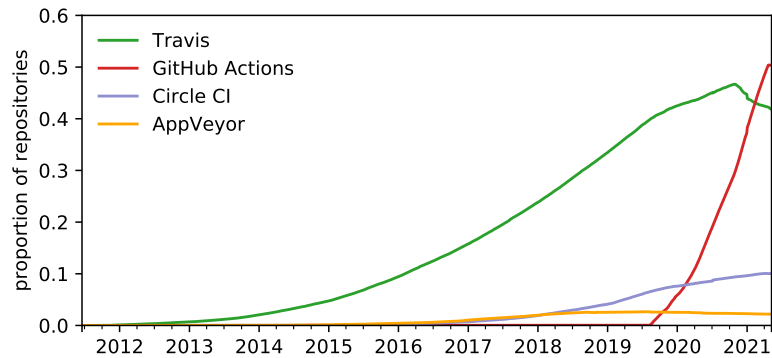


Figure 2.2: Evolution of the proportion of GitHub repositories using a specific CI/CD solution.(Golzadeh *et al.*, 2021c)

GitHub Actions workflows allow GitHub repository maintainers to automate a wide range of tasks. In addition to providing typical CI/CD services such as building code, executing test suites and deploying new releases, GitHub Actions’ tight integration with GitHub enables it to include better support of third-party tools, build support for well-known operating systems and hardware architectures in comparison to third-party CI/CD integrations (*e.g.*, Travis, CircleCI, etc.).

Executable workflows. GitHub Actions is based on the concept of executable *workflows* defined by maintainers of GitHub repositories. At a conceptual level, a workflow specifies when automation should be triggered, which tasks should be executed, and how these tasks are organized and executed. Figure 2.3 provides a schematic overview of this typical workflow structure, illustrating its main components and their relationships.

In essence, a workflow is composed of one or more jobs that are triggered by specific events, such as pushes, pull requests, or scheduled executions. Each job defines an execution environment and consists of an ordered set of steps, which may invoke reusable actions or execute custom commands. Dependencies between jobs can be expressed to control execution order and enable parallelism.

Building on this general model, we next examine a concrete workflow instance to illustrate how these structural elements are realized in practice. Figure 2.4 presents an example GitHub Actions workflow configuration, which is discussed in detail to clarify how abstract workflow concepts are instantiated in a real repository.

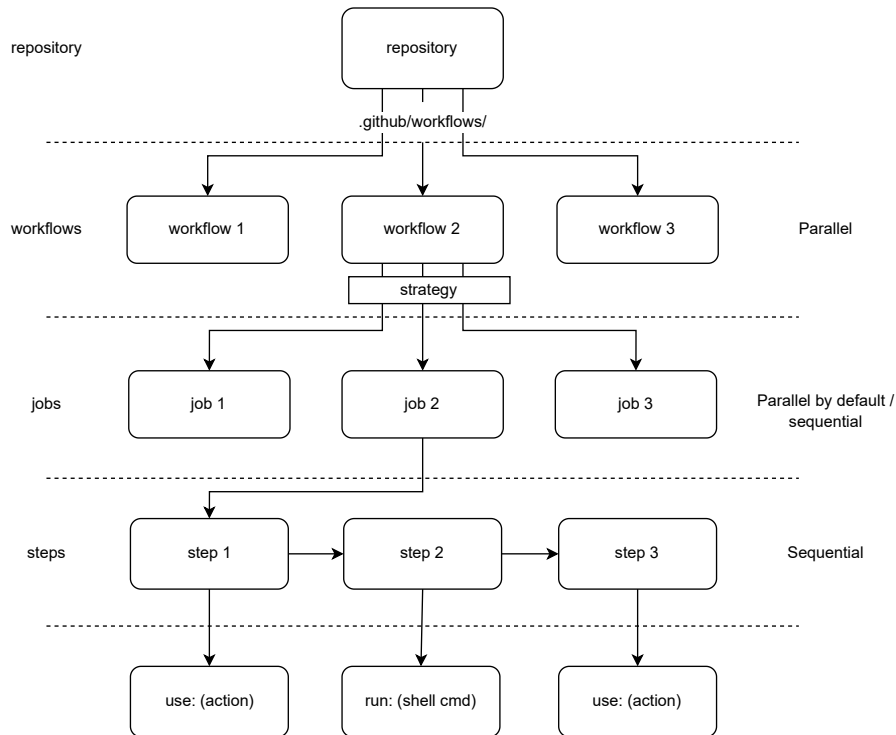


Figure 2.3: Schematic representation of the structure of a GitHub executable workflow.

The workflow `CI and Dependency Checks` shown in Figure 2.4 illustrates several core features of GitHub Actions and provides a representative case of how maintainers specify and automate repository tasks.

The workflow can be triggered (`on:`) by multiple types of events. It reacts to common repository activities such as commits (`push:`) and pull requests (`pull_request:`), which ensures that code changes are continuously tested and validated as they are introduced. In addition, it supports a `schedule:` defined through a `cron:` expression (line 5), which executes the workflow every Monday at 06:00 UTC. Scheduled workflows are particularly useful for recurring tasks such as dependency updates, regression testing, or periodic quality checks. The presence of the `workflow_dispatch:` trigger (line 7) allows developers to manually initiate the workflow when needed, offering flexibility for ad-hoc testing or verification.

To prevent redundant or overlapping runs, the workflow uses `concurrency:` control (lines 9-11). This ensures that only one instance of the workflow is

```
1 name: CI and Dependency Checks
2 on:
3   push:
4   pull_request:
5   schedule:
6     - cron: "0 6 * * 1" # every Monday 06:00 UTC
7   workflow_dispatch:
8
9 concurrency:
10  group: ci-${{ github.ref }}
11  cancel-in-progress: true
12
13 jobs:
14   build-and-check:
15     runs-on: ${{ matrix.os }}
16     timeout-minutes: 20
17     strategy:
18       matrix:
19         os: [ubuntu-latest, windows-latest]
20         python: ["3.9", "3.10"]
21     steps:
22       - uses: actions/checkout@v4
23       - uses: actions/setup-python@v4
24         with:
25           python-version: ${{ matrix.python }}
26       - uses: actions/setup-java@v4
27         with:
28           java-version: 21
29           distribution: "zulu"
30       - name: Run tests
31         run: pytest
32       - name: Check dependencies
33         run: mvn dependency:analyze
```

Figure 2.4: Example of a GitHub Actions workflow with scheduled triggers, concurrency control, matrix builds, and dependency checks.

executed for a given branch at a time. If a new run is triggered while a previous run is still in progress, the ongoing run is cancelled in favor of the new one. This mechanism avoids resource waste and ensures that results always reflect the most recent code state.

The workflow defines a single job, `build-and-check`, which exemplifies how GitHub Actions supports multi-platform and multi-version testing through a strategy matrix: (lines 17-20). The matrix instructs GitHub to execute the job

across different operating systems (Ubuntu and Windows) and two versions of Python (3.9 and 3.10), automatically generating multiple job instances. This feature is particularly important for projects that need to guarantee compatibility across platforms and runtime environments. Each job instance runs inside a clean, isolated virtual machine provided by GitHub's hosted runner infrastructure, with a maximum runtime of 20 minutes (line 16) to prevent excessively long executions.

The job itself is composed of sequential *steps* (lines 21-33), each specifying a discrete task. The first step checks out the project's source code using the widely adopted `actions/checkout` Action. Subsequent steps prepare the execution environment: one installs the desired Python version through `actions/setup-python`, while another configures a Java runtime environment (version 21 of the Zulu distribution) via `actions/setup-java`. These reusable Actions ensure reproducibility by abstracting away platform-specific installation details and by standardizing the setup process across all job instances.

The final two steps illustrate the kinds of automated tasks typically associated with CI/CD workflows and, more specifically, the use of the `run:` directive in GitHub Actions. Using this mechanism, a workflow can execute arbitrary shell commands directly within the runner environment, allowing maintainers to integrate existing tooling without requiring dedicated or reusable Actions.

In this example, the workflow first executes the project's test suite using `pytest`, ensuring that changes to the codebase do not introduce regressions. It then performs a lightweight dependency analysis with Maven, providing insight into potential dependency issues and helping maintainers maintain a healthy build configuration. Together, these steps demonstrate how GitHub Actions workflows combine testing and quality assurance with routine project maintenance through direct command execution.

In summary, this example highlights the expressive power of GitHub Actions workflows. It integrates multiple event triggers, enforces concurrency constraints, employs matrix-based parallelization for cross-platform assurance, and incorporates both reusable Actions and custom commands via the `run:` syntax. Through this single declarative specification, repository maintainers can automate a broad spectrum of activities, ranging from routine regression testing to dependency management. This dual focus on automation and reproducibility makes GitHub Actions a central enabler of modern collaborative development practices on GitHub.

Workflows can reuse any Action shared in a public repository. To facilitate finding such Actions, the GitHub *Marketplace* provides an interface for providers to promote their Actions, and for consumers to easily search for suit-

able Actions.¹ The number of Actions promoted on the Marketplace has been growing exponentially. By November 2025, the Marketplace listed over 45K reusable Actions falling under 33 different categories.

2.5. GitHub Apps and Automation Bots

Another automation tool provided by GitHub is GitHub Apps that are designed to extend the functionality of the GitHub platform (GitHub, 2025a). They interact with repositories, organizations, or individual accounts through GitHub's REST and GraphQL APIs, and operate under fine-grained, permission-based access. Unlike personal access tokens or OAuth apps, which typically act on behalf of a user, GitHub Apps function as independent entities with their own identity and permissions, following the principle of least privilege (GitHub, 2025d). This makes them particularly well-suited for secure, automated collaboration across repositories.

Once installed, GitHub Apps can perform a wide variety of tasks such as creating issues, reviewing or merging pull requests, running automated checks, or integrating external services like deployment platforms, security scanners, and monitoring systems. They can be installed on a single repository or across all repositories within an organization, allowing developers and maintainers to tailor their scope of operation (GitHub, 2025g). Many popular services, such as Dependabot for dependency updates or Codecov for coverage reporting, are implemented as GitHub Apps. Developers and organizations can browse, install, and configure these apps directly through the GitHub Marketplace, which provides a central catalog of both free and commercial offerings.

GitHub Apps and GitHub Actions play complementary roles. GitHub Actions are event-driven workflows defined within a repository, typically used for tasks like testing, building, and deploying software. They are ephemeral in nature, running within temporary execution environments managed by GitHub whenever a triggering event occurs. In contrast, GitHub Apps represent persistent integrations that continuously monitor activity, respond to webhooks, and can act across multiple repositories or organizations (GitHub, 2025f).

In this way, GitHub Apps provide persistent, service-oriented automation, while GitHub Actions enable fine-grained, workflow-based automation within repositories. Together, they illustrate the two main approaches for extending and customizing development workflows in GitHub.

Before the introduction of GitHub Actions, and even prior to the formal-

¹See <https://github.com/marketplace>. In addition to Actions, the marketplace also promotes Apps, which are applications that can contain multiple scripts or an entire application.

ization of GitHub Apps, a substantial portion of automation on GitHub was implemented through bots and external services reacting to webhook events and interacting with repositories via the GitHub API. These automation agents perform a wide range of maintenance and governance tasks, including labeling issues and pull requests, enforcing contribution guidelines, managing stale discussions, updating dependencies, and reporting quality metrics (Erlenhov *et al.*, 2019). Well-known examples such as Dependabot, Renovate, and Codecov exemplify this class of automation and are now typically implemented as GitHub Apps. Over time, GitHub Apps have largely subsumed earlier bot-based approaches, providing a standardized security model, fine-grained permissions, and first-class integration into the GitHub user interface (Wang *et al.*, 2022).

Despite their importance, automation bots and GitHub Apps are not the primary focus of this dissertation. The study of bots, their behavior, detection, and impact on collaborative development has already been the subject of extensive empirical research, including multiple doctoral dissertations within our research group. In particular, prior work by Golzadeh (Golzadeh *et al.*, 2021a; Golzadeh *et al.*, 2021b), Chidambaram (Chidambaram *et al.*, 2023, 2025), and others has examined bot usage, bot detection, and automation-mediated collaboration on GitHub in depth. In this thesis, our focus lies on GitHub Actions, as they represent the central mechanism for workflow automation within repositories and are directly tied to the research goals of adoption, evolution, and maintenance in collaborative software development.

2.6. Other GitHub Automation Mechanisms

Beyond Actions, Apps, and Bots, GitHub provides additional mechanisms that enable automation and integration.

Webhooks. Webhooks are among the earliest automation mechanisms supported by GitHub and predate both GitHub Actions and GitHub Apps. They allow repositories to notify external services whenever predefined events occur, such as a code push, the creation or merging of a pull request, issue updates, or release publications (GitHub, 2025k). When such an event is triggered, GitHub sends an HTTP POST request containing a structured JavaScript Object Notation (JSON) payload to a user-configured endpoint. This enables external systems to react immediately without needing to poll GitHub for updates (GitHub, 2025e).

Because of their simplicity and event-driven nature, webhooks have long

served as a foundational mechanism for integrating GitHub with third-party services. They are frequently used to trigger CI/CD pipelines in external systems such as Jenkins or CircleCI, synchronize repository activity with project management tools, update dashboards, or feed monitoring and analytics services. Although webhooks lack the higher-level abstractions offered by GitHub Actions, such as jobs, steps, or reusable actions, they remain an essential lightweight mechanism for connecting GitHub to the broader software ecosystem. Their flexibility, language-agnostic design, and low operational overhead make them particularly suitable for custom or infrastructure-specific automation needs.

GitHub API. GitHub exposes two powerful APIs namely REST (GitHub, 2025c) and GraphQL (GitHub, 2025h) that provide programmatic access to nearly every resource on the platform. These APIs allow developers, maintainers, and automated systems to interact with repositories and organizational assets in fine-grained ways: querying metadata, managing issues or pull requests, orchestrating releases, modifying repository settings, or retrieving workflow execution details. The GraphQL API offers even greater efficiency and expressiveness by allowing clients to craft structured queries that retrieve exactly the data they need in a single request (GitHub, 2025j).

GitHub CLI. Complementing these APIs is the GitHub Command Line Interface (CLI), available via the `gh` command (GitHub, 2025b). The CLI bridges the gap between interactive development workflows and automation by enabling developers to script routine tasks such as cloning repositories, creating pull requests, viewing workflow runs, or triggering Actions directly from the terminal (GitHub, 2025b). Together, the API and CLI substantially extend GitHub’s automation capabilities beyond what is possible with YAML-defined workflows alone, providing a foundation for integrating GitHub into custom development pipelines, large-scale data analyses, or research tooling.

2.7. Summary

In this chapter, we traced the evolution of automation in collaborative software development, from CI/CD tools like Jenkins and Travis, to the modern integrated GitHub Actions automation ecosystem within GitHub. We highlighted how the principles of CaC and Infrastructure-as-Code (IaC) laid the foundation for reproducible and maintainable pipelines.

We then provided a detailed account of GitHub Actions, focusing on its

structure of workflows, jobs, steps, and reusable Actions, and its rapid adoption as the dominant CI/CD service within GitHub. We complemented this with a discussion of GitHub Apps and bots, which provide service-oriented automation, and of other mechanisms such as webhooks, API, and CLI.

Together, these mechanisms illustrate the diverse strategies available for extending and customizing GitHub-based development. While all forms of automation contribute to repository productivity and governance, this dissertation focuses specifically on GitHub Actions, it is the central mechanism for workflow automation within repositories and directly aligns with the dissertation objectives on adoption, evolution, and maintenance.

Related Work

“If I have seen further it is by standing on the shoulders of giants.”

Isaac Newton

Numerous investigations have explored the positive and negative effects of workflow automation tools on software development processes. This chapter offers an overview of the current state of research related to the dominant CI/CD tool in GitHub. It is reporting on historical and contemporary studies concerning automation in GitHub including the use of CI/CD, bots, and apps, providing a comprehensive understanding of the impact of integrating this tool into collaborative software development practices.

3.1. Empirical Research on the Benefits and Challenges of CI/CD

Systematic literature reviews (SLRs) provide valuable entry points for understanding how CI/CD practices have been adopted, implemented, and evaluated across both industrial and open-source settings. Two major SLRs by Shahin *et al.* (2017) and Soares *et al.* (2022) synthesise the state of research on continuous practices prior to the emergence of GitHub Actions. Both reviews cover foundational work on CI/CD processes, tools, benefits, and challenges, but neither includes studies published after 2019, and therefore exclude the rapidly growing body of research centered on GitHub Actions, introduced in November 2019.

Shahin *et al.* (2017) conducted one of the earliest comprehensive SLRs on continuous practices, analysing 69 peer-reviewed studies published up to 2016. Their review synthesized empirical and conceptual research on CI, and CD identifying both technical and organisational factors that influence adoption. The review highlights several recurring challenges, including integration complexity, tooling limitations, and differences between development environments. At the same time, it reports substantial benefits, such as reduced build and test times, improved visibility into development progress, and more robust deployment pipelines with enhanced security, reliability, and scalability. Importantly, the authors emphasise the socio-technical nature of continuous practices, noting that organisational processes and team communication play central roles in successful adoption.

Complementing this perspective, Soares *et al.* (2022) reviewed 101 studies published until 2019 to examine the empirical effects of CI/CD adoption on software development outcomes. Their synthesis shows strong evidence linking CI/CD practices to improved productivity, higher developer confidence, faster release cadence, and increased transparency in development workflows. They further highlight that CI/CD plays a crucial role in modern pull-based development models, reducing integration friction and enabling faster, more predictable iterations. However, the authors note gaps in understanding certain sub-practices and call for deeper empirical research on the evolution and operational characteristics of CI/CD pipelines.

Several case studies have explored CI/CD usage, cost, and benefits in the context of companies. L. Chen (2015, 2017) reported a detailed longitudinal account of adopting CD practices at Paddy Power, where teams faced both technical and organisational obstacles when scaling continuous delivery. Their findings underscore benefits such as accelerated time-to-market, improved effi-

ciency, higher reliability of releases, and better communication between development and operations teams. Importantly, they draw attention to the interplay between tooling, team culture, and organisational maturity in sustaining continuous practices over time.

In a scientific computing context, Betz and Walker (2013) analysed the adoption of CI/CD for AMBER, a high-performance molecular dynamics package maintained by globally distributed developers. The authors show that CI helped teams rapidly detect performance regressions and correctness issues, while also improving contributor coordination in a geographically dispersed community.

Similarly, Lu *et al.* (2014) demonstrated how CI and automated testing improved quality assurance for D5000, a smart grid scheduling support system. Their case study provides evidence that well-integrated testing pipelines can reduce integration risks with minimal overhead, even in safety-critical domains.

Kulas *et al.* (2014) presented another domain-specific case involving ARGOS, a telescope image processing system. Using Jenkins for CI allowed teams to verify correctness under strict time constraints and highlighted CI's role in ensuring reliability in high-stakes scientific environments.

Organisational evolution over long time periods is captured in the six-year study by Gmeiner *et al.* (2015), which documents how an Austrian online business company introduced and refined CD practices. Their findings emphasise socio-technical challenges, including resistance to process change, the need for automation expertise, and difficulty in scaling pipelines as team and system complexity increased.

Large-scale industrial usage is also explored in Savor *et al.* (2016), who studied CI/CD at Facebook and OANDA. While both companies operated sophisticated pipelines, the authors highlight that organisational policies such as manual gatekeeping of deployments often prevented full adoption of CD, creating bottlenecks despite advanced tooling.

More recently, Jin *et al.* (2024) studied the effect of configuration-based CI/CD pipelines at ByteDance. They observed that introducing configuration files improved pipeline reliability, encouraged more frequent integrations, and supported a shift towards a more automated development culture. Their work provides evidence that even mature engineering organisations benefit from standardising pipeline configuration.

Finally, Elazhary *et al.* (2022) conducted interviews with developers from three software organisations to identify perceived benefits and challenges of CI/CD. They found that pipelines improved feedback cycles, reproducibility, and integration quality, but also introduced pain points such as long build

times, bottlenecks during pull-request review, limited scalability, and increased maintenance overhead. Their results highlight that while CI/CD brings measurable benefits, operational complexity grows as pipelines evolve.

A complementary perspective on the relationship between contribution practices and CI/CD is offered by Elazhary *et al.* (2019). In a mixed-methods study of 53 active GitHub projects using CI, the authors compared the contribution processes prescribed in project guidelines against observed developer behaviour. They found that CI tools were mentioned in only about 31% of contribution guidelines, and only as mechanisms for running tests, with no documentation of how CI fits into the broader workflow. More strikingly, approximately 68% of the studied projects diverged significantly from their own prescribed processes: while guidelines typically described pull-request-based workflows, the vast majority of actual activity involved direct commits to the main branch. This work highlights a recurring disconnect between the ideals encoded in project documentation and the realities of day-to-day development, underscoring that CI adoption alone does not guarantee adherence to best practices.

Others have studied the impact of CI/CD on OSS development. Zampetti *et al.* (2017) studied the usage of static analysis tools to enable early detection of potential faults, vulnerabilities, and code smells through their adoption of CI pipelines. By analysing 20 Java OSS projects hosted on GitHub and using Travis, they showed that static analysis tools are used to induce failing builds to highlight non-adherence to coding standards and missing licenses. Build failures to highlight potential bugs or vulnerabilities occur less frequently, and in some cases, such tools are activated in a “softer” mode, without making the build fail. The study also reveals that the aforementioned build breakages are quickly fixed by actually solving the problem, rather than by disabling the warning, and are often properly documented.

Hilton *et al.* (2016) looked into the usage, costs, and benefits of CI/CD in 34,544 OSS projects on GitHub. They observed that CI/CD is widely adopted by the most popular projects, with an increase in the adoption of CI/CD over time. CI/CD was observed to help projects release more often, and a wide variety of CI/CD tools were used by the projects, with Travis, CircleCI, AppVeyor, CloudBees, and Wercker being the most popular ones at that time.

Vasilescu *et al.* (2015) studied the quality and productivity outcomes ensuing from embracing CI/CD practices in OSS projects on GitHub. They observed that teams using CI/CD are significantly more effective at merging pull requests submitted by core members. CI/CD was also associated with external contributors having fewer pull requests rejected. Moreover, core de-

velopers in teams using CI/CD tools discover significantly more bugs than in teams not using such tools.

Additionally, some studies have deepened the understanding of CI/CD usage, particularly regarding monitoring practices, productivity impacts, and maturity of adoption. Researchers have increasingly examined not only whether CI/CD is used, but how well it is monitored, adhered to, and sustained over time. Santos *et al.* (2025b) investigated developers' perceptions of monitoring CI practices, arguing that although monitoring is essential for ensuring build health and process compliance, it remains largely overlooked in both tools and developer workflows. Through document analysis and a survey of developers across 121 open-source projects, the study showed that key CI practices such as fixing broken builds promptly are often insufficiently monitored. Moreover, they found that existing CI services provide limited native support for monitoring CI practices, requiring developers to rely on third-party tools with fragmented coverage. Their results underline a broader need for CI platforms to integrate actionable monitoring features directly into their dashboards.

Relatedly, Santos *et al.* (2022) analysed the impact of five CI practices on productivity and quality across 90 GitHub open-source projects over two years. Using regression models, they demonstrated that adherence to practices such as build activity, commit activity, and build health correlates with higher productivity (*e.g.*, more merged pull requests) and fewer bug-related issues. A complementary qualitative analysis showed that projects with stronger CI discipline experience fewer CI-related problems. The study concludes that maintaining robust CI practices is beneficial not only for build reliability but also for overall project quality.

Another line of work examined how open-source projects apply CI techniques in practice. Wróbel *et al.* (2023) conducted an exploratory study of 10 GitHub repositories comprising over 100,000 pull requests and nearly 400,000 commits. Their results indicated that many projects have not yet reached a mature level of CI adoption. Most failed CI builds were fixed through small code changes, smaller pull requests tended to succeed more frequently, and no clear relationship was found between developer experience and build success. These findings suggest that CI adoption varies widely across projects, and many contributors still submit insufficiently tested code, placing additional effort on CI services to detect issues.

Finally, Santos *et al.* (2025a) investigated CI practice monitoring in three Brazilian software companies through a mixed-methods industry case study. The authors developed a tool to monitor seven CI practices and found that

monitoring helps teams gain visibility into their CI health, motivates improvements, and enhances perceived software quality. The study highlights several organizational benefits of monitoring including improved communication and increased awareness of build reliability and recommends that CI services integrate monitoring support more explicitly.

Beyond adherence and monitoring, a growing strand of research has begun to examine the *resource cost* of CI itself. Zaidman (2024) conducted an exploratory study measuring the electricity consumption of continuous integration and automated testing across 10 open-source Java projects. Building and testing the most active project in the study, Elasticsearch, was estimated to consume approximately 161 kWh per year on a mid-range processor which is the equivalent of nearly 10% of the average annual household electricity use of a European citizen. The study also found that the testing phase accounts for between 20% and 88% of a full build’s energy consumption, depending on project characteristics.

Expanding on this baseline, Arntzenius *et al.* (2026) conducted a larger-scale replication across 204 open-source Java projects using Maven and Gradle, providing the first comprehensive characterisation of CI energy distribution. Their results confirm that energy use is highly skewed: while most projects consume modestly (median around 1.5 Wh per build), a small number of integration-heavy projects can reach annual footprints of hundreds of kilowatt-hours. Crucially, they showed that enabling dependency caching reduces energy consumption by approximately 30% on average for Maven projects and by over 90% in some Gradle cases, establishing caching as a practical and immediately actionable mechanism for greener pipelines. These findings collectively call for sustainability to be treated as a first-class concern in CI/CD engineering, and are directly relevant to the configuration practices examined in this thesis.

The mentioned studies are among many that have explored the benefits, challenges, and practices of CI/CD in software projects. These studies have demonstrated that CI/CD practices can significantly enhance productivity, efficiency, and quality in software development. Moreover, they reveal that while CI/CD tools are widely used, many teams lack systematic mechanisms for monitoring CI practices or ensuring adherence to best practices. Additionally, they have not focused on the evolution of CI/CD configurations over time, which is the focus of the following section.

3.2. Evolution of CI/CD Usage Prior to GitHub Actions

Many CI/CD tools that were in popular use long before the advent of GitHub Actions rely on (often YAML-based) configuration files for their CI/CD configuration pipelines. Examples of such tools are Travis, Jenkins, GitLab CI/CD, and CircleCI. Versioning CI/CD configuration files is an instance of the wider practice known as CaC, which involves encoding configuration settings in a human-readable format (*e.g.*, YAML) and syntax, aligning configuration management with modern software development practices. CaC provides better opportunities for versioning, code reviewing, and change automation. In the context of CI/CD, CaC allows configurations to be updated, tested, and deployed automatically alongside application code, ensuring that application and infrastructure changes are coordinated.

The popularity of Travis on GitHub has led many researchers to study this CI/CD tool. Gallaba and McIntosh (2020) investigated the usage and misuse of features in Travis configuration files in 9,312 GitHub repositories. They found that *job processing nodes* were the most frequently modified, indicating that Travis was predominantly used for CI rather than CD. They also developed tools to identify and remove anti-patterns in Travis configuration files. Similarly, Vassallo *et al.* (2019) created a tool to detect anti-patterns in Java projects using Travis. They based their identification of critical anti-patterns on Duvall’s work (Duvall *et al.*, 2007), which served as a benchmark for quality checking in CI.

Durieux *et al.* (2019) compiled a dataset of over 35 million Travis jobs from 272,917 projects. They discovered that the majority of the 709,000+ commits that modified Travis configuration files were related to debugging. This finding highlighted the need for more in-depth analysis of the nature of these changes.

Zampetti *et al.* (2020) identified 79 bad CI practices through semi-structured interviews with 13 experts and an analysis of over 2,300 Stack Overflow posts. They also studied the evolution of changes to Travis configuration pipelines, finding that jobs and steps were the most frequently changed components, and noted an increasing adoption of Docker over time (Zampetti *et al.*, 2021).

Saidani *et al.* (2021) investigated how the adoption of CI influences refactoring practices by analysing nearly 100k commits and close to 90k refactoring operations from 39 open-source projects using Travis. Their study reveals that after introducing CI, refactoring tends to become smaller in scope, aligning with recommended fine-grained improvement practices, yet both the frequency

of refactoring and the number of developers performing it decline. These findings suggest that despite CI's emphasis on rapid integration and feedback, it may inadvertently discourage developers from undertaking refactoring activities. The authors conclude that developers may require better tool support to safely apply refactorings within the fast-paced CI environment.

Gallaba *et al.* (2022b) presented an exploratory study of 22.2 million CircleCI builds across 7,795 open-source projects to understand how CI services perform in practice and where improvements are most needed. Their quantitative analysis shows that different types of service consumers spend disproportionate amounts of time on various build actions, though compilation and testing consistently dominate build duration. They also examined builds that terminate prematurely, identifying service availability issues, configuration errors, user cancellations, and timeouts as the primary causes. The study highlights significant opportunities for CI providers to optimize build performance, improve robustness by reducing misconfigurations and availability problems, and enhance efficiency by targeting the costly compilation and testing stages.

A particularly relevant line of work concerns the quality of CI/CD configuration files themselves. Vassallo *et al.* (2020) proposed CD-Linter, a semantic linter that detects process-level violations of CD principles which is referred to as *CD smells* in GitLab CI/CD pipeline configuration files. The tool targets four recurring anti-patterns: *Fake Success* (allowing jobs to fail without failing the build), *Retry Failure* (masking flaky behaviour by automatically retrying failed jobs), *Manual Execution* (requiring human intervention in stages that should be automated), and *Fuzzy Version* (under-specifying dependency versions in ways that threaten build reproducibility). Through a six-month longitudinal study in which 145 issues were opened in as many open-source projects on GitLab, the authors found that 53% of project maintainers reacted positively by either accepting or fixing the reported smells. A large-scale analysis of 5,312 GitLab projects further revealed that 31% of projects with long configuration files were affected by at least one smell, with Fuzzy Version being the most prevalent. CD-Linter achieves an overall precision of 87% and a recall of 94%, demonstrating that static analysis of configuration files is both feasible and valued by practitioners. This work is directly relevant to the present thesis: it establishes that CI/CD configuration files are first-class artifacts that deserve systematic quality assessment, and it provides a vocabulary of recurring misconfiguration patterns that inform our own analysis of how workflows evolve and degrade over time.

Finally, Golzadeh *et al.* (2021c) studied the adoption of CI/CD tools in over 91,000 GitHub repositories related to npm packages. They found that by

May 2021, more than 50% of the repositories used CI/CD tools, with GitHub Actions and Travis being the most prevalent. Remarkably, GitHub Actions replaced Travis as the leading CI/CD tool within 18 months of its introduction, and many repositories transitioned from Travis to GitHub Actions.

3.3. Adoption and Usage of GitHub Actions Workflows

This subsection reviews studies that examine how GitHub Actions workflows are adopted, maintained, and evolved within software repositories. Early investigations into the emergence of GitHub Actions focused primarily on adoption dynamics and developer perceptions. For example, Kinsman *et al.* (2021) analysed the growth of CI/CD usage within GitHub projects and documented early evidence of developers migrating from external CI services to GitHub Actions, citing convenience, integration depth, and community support as major motivators. Similarly, T. Chen *et al.* (2021) explored developers' attitudes toward adopting workflow automation, reporting that while GitHub Actions lowers the barrier to entry, developers often struggle with configuration complexity and uncertainty about best practices.

Complementing adoption-focused studies, a number of works have examined the maintenance and evolution of workflows in practice. Valenzuela-Toledo and Bergel (2022) provided one of the earliest empirical analyses of how developers modify workflows over time, identifying common change patterns and noting that workflow edits frequently correspond to adaptation needs, such as updating dependencies or adjusting execution environments. Expanding on this line of inquiry, Valenzuela-Toledo *et al.* (2024) highlighted how subtle misconfigurations and implicit behavioural assumptions can cause workflows to break in ways that are difficult for developers to diagnose. Their study revealed that even small changes such as altering triggers, permissions, or action versions may introduce unexpected behaviour, underscoring the fragility of workflow specifications.

Quality concerns have also been investigated. Khatami *et al.* (2024) introduced an approach for detecting workflow smells, cataloguing recurring anti-patterns that hinder clarity, correctness, and maintainability of workflow definitions. Their findings show that code smells in workflows mirror quality issues traditionally observed in source code, reinforcing the notion that workflows deserve the same systematic engineering attention as traditional software artifacts. Complementing this static quality perspective, Khatami *et al.* (2026) in-

investigated how GitHub Actions workflows are adopted and utilised in practice, going beyond the static analysis of YAML configuration files to examine actual workflow execution records. Analysing 258K+ workflow run records from 952 repositories and conducting an in-depth qualitative study of 21 projects, the authors identified three distinct failure response patterns namely, immediate fixing, deferred fixing, and an ignored or abandon approach. Crucially, the study uncovered a configuration-usage gap which indicates that the mere presence of workflow configuration files is a poor proxy for actual adoption, as many projects maintain configurations for workflows that are disabled, rarely triggered, or whose results developers routinely ignore.

Security-oriented studies further highlight the risks associated with workflow misuse. Onori Delicheh *et al.* (2024) conducted a large-scale analysis of JavaScript-based Actions and demonstrated that dependency chains introduce substantial attack surfaces, with more than half of the analysed Actions containing at least one detectable security weakness. Their work underscores that workflows not only orchestrate automation but also form a dependency ecosystem that may expose repositories to supply chain vulnerabilities.

Beyond workflow definitions, researchers have begun to examine workflow correctness and generation. More recent work has shifted attention toward workflow executions and operational behaviour. Moriconi *et al.* (2025) introduced GHALogs, the largest publicly available dataset of workflow run logs to date, comprising over 500K workflow executions enriched with fine-grained metadata. This dataset enables analyses of performance bottlenecks, failure trends, and execution patterns that were previously inaccessible due to the scarcity of log-level CI/CD data. Complementing this perspective, Zheng *et al.* (2025) provided the first empirical taxonomy of workflow failure types by manually analysing failed executions in Java repositories and validating their findings through developer surveys. Their taxonomy captures common root causes, including configuration errors, dependency issues, environment mismatches, and external service failures, offering valuable insights into recurring reliability problems in automation pipelines.

Building on prior failure analyses, Huang and Lin (2026) investigated how developers respond to unsuccessful GitHub Actions executions by analysing workflow reruns in 3,320 open-source Java repositories. They focused on cases where workflows or failed jobs are re-executed without repository modifications, quantifying the associated time and computational overhead. Through manual inspection of successful reruns, the study identified key sources of execution flakiness, including non-deterministic dependencies and unstable external services. The authors further evaluated machine learning-based models to

predict workflow outcomes, demonstrating the potential to reduce unnecessary reruns and improve CI/CD efficiency.

Zhu *et al.* (2023) introduced Actionsremaker, a tool for reproducing GitHub Actions workflow executions, addressing a longstanding challenge in empirical software engineering research. Reproducibility of CI/CD runs is critical for tasks such as fault localization, program repair, and the analysis of flaky tests, yet GitHub Actions complicates replication due to its dynamic execution environments and reliance on external services. The study shows that a substantial portion of failing and passing builds can be successfully reproduced and uses unreproducible cases to expose systemic limitations of modern, cloud-based CI pipelines. Overall, Actionsremaker provides an important foundation for large-scale, reproducible studies of GitHub Actions workflows.

Khelifi *et al.* (2025) proposed GHMiner, an open-source toolchain for mining workflow-level data from GitHub Actions executions. The work addresses the practical difficulty of extracting reliable CI metadata from heterogeneous logs and constrained platform APIs. By enabling systematic collection of build outcomes, execution times, and related metrics across thousands of repositories, GHMiner provides foundational infrastructure for empirical studies of CI practices and performance on GitHub Actions.

Saavedra *et al.* (2024) introduced GitBug-Actions, an approach for constructing reproducible bug-fix benchmarks by leveraging GitHub Actions as the mechanism for detecting and validating faults. The work responds to limitations of existing benchmarks, such as dependency drift and fragile execution environments, by relying on CI executions to ensure long-term reproducibility. Through a proof-of-concept benchmark for Go projects, the authors show that real bug-fix instances can be systematically collected from active repositories, illustrating how GitHub Actions can support the creation of realistic and up-to-date datasets grounded in modern CI practices.

Zhang *et al.* (2024) conducted the first systematic investigation into the capabilities and limitations of LLM for GitHub Actions workflows, focusing on both workflow generation and security analysis. Recognizing that workflows differ from general-purpose code in structure, semantics, and security constraints, the authors curated a corpus of roughly 400,000 real workflows and evaluated state-of-the-art LLMs across five tasks, including workflow synthesis, correction of syntactic errors, and detection of injection vulnerabilities. They further fine-tuned LLM variants on workflow-specific data to assess how domain adaptation influences performance. Their results reveal that, although LLMs can generate plausible workflow configurations, they frequently produce insecure or non-executable specifications, particularly under limited prompt

information. The study highlights both the promise and the risks associated with leveraging LLMs in CI/CD configuration engineering, emphasizing the need for improved models or guardrails when applying LLMs to workflow automation.

The environmental cost of workflow execution is an emerging but increasingly important concern. Verdecchia *et al.* (2025) introduced the concept of *energy-aware software testing*, arguing that the growing scale of automated test suites, already numbering in the hundreds of millions at organisations like Google, creates a non-negligible energy burden that the software engineering community has largely overlooked. As a concrete step, the authors designed and evaluated GREEDY_E, an energy-aware variant of a similarity-based test prioritisation algorithm, and demonstrated that it can reduce energy consumption by up to 54% when running a partial test suite, without significantly compromising fault-detection effectiveness.

Together, these studies establish GitHub Actions workflows as first-class configuration artifacts that play a central role in contemporary CI/CD pipelines. They collectively illustrate the richness of the GitHub Actions ecosystem from adoption and usage patterns, to configuration quality issues, security vulnerabilities, and execution behaviour. However, despite this growing body of research, the evolutionary dimension of workflows remains under-explored. Existing studies has focused on high-level adoption trends and runtime outcomes such as failures and performance. What is still missing is a comprehensive, fine-grained understanding of how workflows change over time: which components are most frequently modified, what types of conceptual changes developers introduce, and how these evolving configurations shape maintenance challenges in practice. Addressing this gap is a central objective of this thesis.

3.4. Studies on Reusable Actions

In addition to workflow-focused studies, a growing body of research has examined reusable Actions, one of the most widely adopted and influential artifacts in the GitHub Actions ecosystem. Reusable Actions provide modular, composable units of automation logic that can be shared across repositories, enabling developers to avoid repetitive configuration and to benefit from community-maintained solutions. Because these Actions are hosted in standalone repositories and versioned independently, they form a dependency network that closely resembles traditional software package ecosystems such as npm, PyPI, and Maven. This ecosystem perspective has motivated several empirical inves-

tigations into how reusable Actions are created, maintained, distributed, and consumed.

A number of studies have explored developer motivations and challenges associated with reusable Actions. Saroar and Nayebe (2023), for example, conducted surveys and interviews to uncover why developers choose to implement or adopt reusable Actions, identifying factors such as reducing duplicated effort, improving workflow maintainability, and leveraging community expertise. However, they also reported significant challenges, including the difficulty of assessing Action quality, unclear documentation, inconsistent versioning practices, and uncertainty about long-term maintenance. These findings highlight the need for better trust mechanisms which is an issue echoed in later work examining adoption signals such as verified publishers, popularity metrics, and security guarantees.

Beyond developer perceptions, researchers have proposed automated systems to support Action selection and maintenance. Huang and Lin (2023) introduced CIGAR, a recommendation tool that ranks reusable Actions according to workflow requirements, popularity trends, and usage patterns. Such tools aim to mitigate the cognitive load that arises from navigating the rapidly expanding Marketplace, where thousands of Actions vary widely in quality, maturity, and security posture. Complementing this work, Khatami *et al.* (2024) examined workflow and Action “smells” demonstrating that common anti-patterns such as unpinned versions, redundant steps, or overly permissive configurations frequently originate from reused components, thereby amplifying maintenance and security risks across dependent workflows.

Security concerns in the ecosystem of reusable Actions have received substantial attention. Onsoni Delicheh *et al.* (2024) conducted the most extensive analysis to date of JavaScript-based Actions, examining over 8,000 repositories and mapping their dependency chains to external npm packages. Using CodeQL analyses, they found that more than 54% of Actions contained at least one detectable vulnerability, often stemming from outdated or unmaintained dependencies. These results highlight that Actions, just like library packages, inherit transitive risks that propagate into the workflows relying on them. Related work by Onsoni Delicheh and Mens (2024) and Decan *et al.* (2023) reinforced these findings by studying the broader supply-chain attack surface of Actions, showing that outdated dependencies, unpinned Action versions, and insecure permission configurations expose workflows to privilege escalation, code injection, and other attack vectors. Their recommended mitigation strategies include dependency monitoring, automated vulnerability scanning, stricter permission management, verification mechanisms for Action publish-

ers, and improved tooling for detecting insecure Action usage.

Taken together, these studies highlight the ecosystem dimension of GitHub Actions: reusable Actions provide powerful mechanisms for extensibility and reuse, but they also introduce ecosystem-scale maintenance responsibilities, dependency risks, and security challenges comparable to traditional software package repositories.

Although this literature underscores the importance of reusable Actions in shaping the reliability and safety of CI/CD pipelines, the primary focus of this thesis lies in studying workflows themselves. Workflows remain the central orchestration artifacts governing automation in collaborative development, and understanding their evolution, maintenance patterns, and change dynamics is essential for improving the sustainability of CI/CD practices.

3.5. Summary

This chapter demonstrates the breadth of scholarship on software workflow automation, from early CI/CD practices to the rapidly expanding ecosystem surrounding GitHub Actions. Prior work has extensively examined the benefits and challenges of CI/CD adoption, documented socio-technical factors influencing continuous practices, and analyzed how configuration-as-code pipelines evolve in earlier systems such as Travis, Jenkins, and CircleCI. These studies collectively highlight that automated pipelines significantly improve development productivity, release frequency, and code quality, while also introducing new maintenance demands, configuration complexity, and risks of misconfiguration. They further reveal persistent gaps between prescribed and actual development practices, and a growing awareness that the energy footprint of CI/CD execution is a sustainability concern that tool builders, practitioners, and cloud providers can no longer ignore.

With the emergence of GitHub Actions, researchers have increasingly turned their attention to its adoption, usage patterns, ecosystem of reusable Actions, and operational behavior. Recent investigations have explored workflow maintenance, common change patterns, configuration smells, security vulnerabilities, and execution failures, as well as reproduced CI runs and developed tools to mine and analyze workflow logs. Research on reusable Actions has revealed both the benefits of modular automation and the risks associated with dependency chains, versioning practices, and security exposures. Quality-focused research on the detection of configuration smells in CI/CD configuration workflow and pipelines, has established that pipeline configurations are prone to recurring anti-patterns that hinder reliability and maintainability.

Despite these substantial advances, the evolutionary dimension of GitHub Actions workflows remains comparatively under-explored. Existing work largely focuses on static characteristics, developer perceptions, or runtime behavior, leaving unanswered questions about how workflows evolve over time, which components are most prone to change, and what maintenance challenges arise as workflows adapt to new tools, environments, and community practices. Addressing these gaps motivates the empirical investigations presented in the remainder of this dissertation.

CI/CD Tools Usage, Co-usage, and Migrations

“If you’re not willing to learn, no one can help you. If you are determined to learn, no one can stop you.”

Zig Ziglar

CI/CD practices have become widespread in modern software development, playing a central role in enabling rapid, reliable, and automated software delivery. Among the tools supporting these practices, workflow automation platforms such as GitHub Actions have gained significant traction, especially in collaborative development environments.

Despite their popularity, the motivations and decision-making processes behind how developers adopt and evolve these tools remain under-explored. In particular, we lack a clear understanding of how experienced developers choose, customize, and adapt workflow automation tools over time, and how their practices compare to patterns reported in prior research. This gap presents a challenge for tool designers and researchers aiming to improve the usability and effectiveness of automation solutions.

This chapter serves as the starting point of the thesis and presents an empirical study based on qualitative analysis that investigates the real-world use of workflow automation tools in collaborative software projects hosted on GitHub. The study pursues two main objectives: first, to examine the role of workflow automation in shaping contemporary development practices within

the GitHub ecosystem; and second, to understand practitioners' motivations, experiences, and perceptions regarding the adoption, evolution, and value of these tools. Additionally, this chapter directly addresses the first goal of the thesis, by examining how CI/CD tools are adopted and used across projects, and by identifying developers' motivations for tool selection, co-usage, and migration. Through this lens, the chapter provides the fundamental understanding needed to analyze subsequent patterns of CI/CD workflow usage, evolution, and maintenance within GitHub-hosted projects, as explored in later chapters of this dissertation.

4.1. Introduction

The introduction in November 2019 of GitHub Actions as a fully integrated CI/CD service on GitHub has led both new and existing GitHub repositories to adapt or migrate to this service as their primary CI/CD tool (Golzadeh *et al.*, 2021c; Kinsman *et al.*, 2021). At the same time, Travis has exhibited a progressive decrease in popularity on GitHub these recent years, due to a combination of quality of service problems and restrictions imposed on its free plan for OSS projects (Golzadeh *et al.*, 2021c). In light of these important recent changes in the CI/CD landscape of GitHub, this chapter has two research questions to answer.

As the first research question (RQ 4.1), we aim to answer: *how and why do experienced developers in both commercial and open-source projects select and rely on specific CI/CD tools, and how has this usage evolved compared to findings from prior research?* This question is broken down in five specific subquestions:

RQ 4.1.1 *Which CI/CD tools are being used?*

RQ 4.1.2 *What are the main reported reasons for using CI/CD?*

RQ 4.1.3 *Which activities are being automated by CI/CD tools?*

RQ 4.1.4 *What are the most valuable features of these CI/CD tools?*

RQ 4.1.5 *What are the reported shortcomings of these CI/CD tools?*

As the second research question (RQ 4.2), we ask: *What motivates developers to co-use multiple CI/CD tools simultaneously, and which factors drive their migration from one CI/CD tool to another?* This question is broken down in three specific subquestions:

RQ 4.2.1 *Why do some developers use multiple CI/CD tools simultaneously?*

RQ 4.2.2 *Why do some software projects decide to migrate to a different CI/CD tool?*

RQ 4.2.3 *What are the difficulties in carrying out a CI/CD migration?*

4.2. Methodology

In order to answer our research questions, we carried out a qualitative analysis by conducting semi-structured interviews with experienced software developers around the globe. The remaining of this section is structured as follows: Section 4.2.1 explains how we created our interview questionnaire, Section 4.2.2 how we selected the interview participants, and Section 4.2.3 how the interviews were conducted, processed, and coded.

4.2.1 Interview Questionnaire

The created interview questionnaire aims to capture all the aspects we wanted to cover the answers to our questions. To validate the questionnaire, pilots were carried out with three developers with experience in CI/CD. The results of the interviews with these developers were not included in the analysis, as they only served to further improve the questionnaire.

The final questionnaire is presented in Appendix 8.3.8. It includes about 30 questions, some being conditional to the answers to previous questions. These questions were structured along the following main themes:

1. General questions about the respondent
2. General questions about CI/CD usage
3. Questions about specific CI/CD tool usage
4. Questions about CI/CD migration
5. Questions about CI/CD tool co-usage
6. An open-ended closing question

The responses for themes 2 and 3 were used as a basis for the first research question (see Section 4.3), while themes 4 and 5 served as a basis for the second research question (see Section 4.4).

4.2.2 Selection of Respondents

We targeted interview candidates with experience in software development in OSS as well as in proprietary settings. Our main strategy to find interview candidates was through Twitter (Currently known as X) and LinkedIn channels, e-mails and direct messages to practitioners, and through referrals by colleagues and some interviewees. To increase diversity of interviewees and

to avoid being restricted by geographical constraints, we decided to conduct online in-depth interviews using video conferencing tools.

To qualify to participate in the study, candidates needed to meet at least two out of three inclusion criteria that were defined beforehand: (1) having actively contributed to, or having been responsible for a software project relying on CI/CD; (2) having sufficient knowledge about the reasoning and decision-making process about which CI/CD tool is used in that software project and how; (3) having been involved in setting up or maintaining the CI/CD process of the project.

We stopped selecting and interviewing candidates when we reached a point of saturation (Fusch & Ness, 2015; Guest *et al.*, 2006) where no new themes or codes emerged from the additional data collected. We observed such saturation after the 20th respondent when, except for the answers to the open-ended closing question and the specific work context of the respondents, little additional relevant information was gathered on top of what previous respondents had already provided. We therefore stopped the interview process after the 22nd interview. This is comparable to what has been used in some previous qualitative studies in empirical software engineering (Foundjem *et al.*, 2022; Kim *et al.*, 2016; Meyer *et al.*, 2019) that reported saturation after 16, 16 and 10 interviews, respectively. Nevertheless, we acknowledge that our inclusion criteria for selecting interview candidates were such that we only considered experienced developers with practical expertise in CI/CD usage. As a result, the opinions and findings reported in the paper do not necessarily generalise to inexperienced developers.

Table 4.1 summarises the demographics of the respondents. In the remainder of this chapter, the respondents are identified by a unique number R_n or simply n when it is clear from the context. The second and third columns of the table report on their number of years of development and CI/CD experience. On average, the respondents can be considered as very experienced, with an average of 12 years and 4 months of software development experience and 4.5 years of CI/CD experience. Not all respondents dissociated their years of CI/CD experience from their years of development experience, explaining the absence of the second number for some respondents.

Columns 4 and 5 of the table reveal that respondents were involved in a wide range of software development projects, including OSS and proprietary projects. Most of the respondents (12 out of 22 respondents) had both proprietary and OSS experience, while seven respondents had only been involved in proprietary software, and three of them were only in OSS projects. Furthermore, some of the respondents were or had been working on big OSS projects

Table 4.1: Characteristics and demographics of respondents.

ID	years of experience		proprietary	open source	continent
	development	CI/CD			
R_1	7	-	✓	✓	Europe
R_2	11	-	✓		Europe
R_3	6	-	✓		Europe
R_4	19	-	✓	✓	North America
R_5	22	14	✓	✓	Europe
R_6	19	-	✓		North America
R_7	8	-	✓		Europe
R_8	11	8	✓	✓	Europe
R_9	6	4.5		✓	Europe
R_{10}	20	10	✓	✓	North America
R_{11}	5	4	✓	✓	Europe
R_{12}	8	6	✓	✓	Europe
R_{13}	15	-		✓	Europe
R_{14}	4.5	3	✓	✓	Asia
R_{15}	10	3	✓		Europe
R_{16}	12	2	✓		Asia
R_{17}	15	-	✓		Europe
R_{18}	10	-	✓	✓	Europe
R_{19}	24	-	✓	✓	Europe
R_{20}	15	4	✓	✓	Europe
R_{21}	12	8		✓	North America
R_{22}	20	12	✓	✓	Europe

like curl and Conda-forge, or for big tech companies such as LinkedIn and Microsoft. The last column reveals that most of the respondents lived and worked in Europe (16 respondents spread over 7 different Western European countries), while 4 respondents came from North America and 2 from Asia.

4.2.3 Conducting and Processing the Interviews

The process we followed for conducting and processing each interview is summarised in Figure 4.1. Prior to each interview, the selected interview candidate was required to sign a consent form in order to meet the General Data Protection Regulation (GDPR) regulations to allow us to use the interview results for research purposes. After having received the consent form, a virtual meeting was fixed to carry out the online interview through a video-conferencing tool the candidate was comfortable with. One interviewer conducted the interview and made an audio recording, with the explicit permission of the candidate.

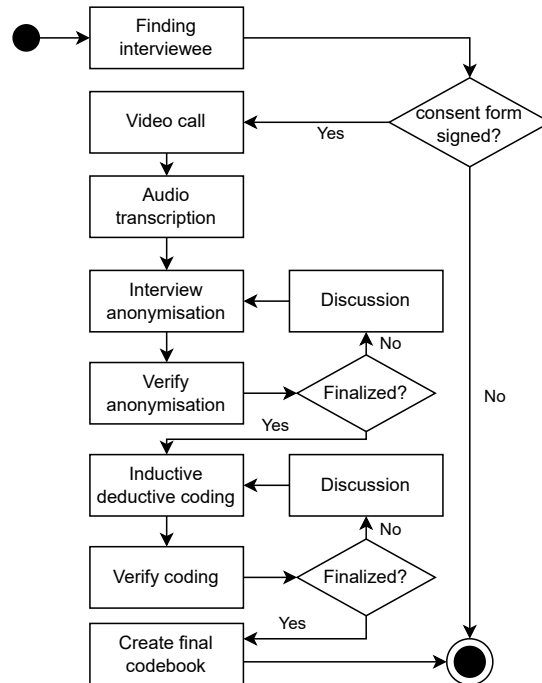


Figure 4.1: Schema of the interview process.

Each of the 22 interviews lasted roughly about 30 to 45 minutes, and the total set of interviews was spread over a four-month period, from November 2021 to February 2022.

The interviewer resorted to an automatic transcription tool for each interview. The resulting verbatim textual transcripts were cleaned and anonymised to hide privacy-sensitive information such as names of persons, companies, or specific software projects. This process was made by one researcher, and was checked and further improved by a second researcher. A third researcher was involved in case of doubt.

To structure the information gained from the interview transcripts we followed a process similar to Foundjem *et al.*, 2022, using a combination of *inductive* and *deductive coding* (Bernard *et al.*, 2016). In the first phase of inductive coding, a first researcher, the author of this thesis, assigned labels to the transcribed text, without any predetermined theory, structure, or hypothesis. After that, same researcher followed a top-down deductive coding process to create separate codebooks for each interview, deriving codes based on the research

questions and concepts under study, and using these codes to group and structure the inductive labels that were attached to the transcribed text during the inductive coding phase. A second researcher verified each of these codebooks and, in case of disagreement, a third researcher was involved in the discussion until a consensus was reached on the coding.

4.3. Why, How, and Which CI/CD Tools are Used

This section aims to understand the rationale behind how and why developers rely on specific CI/CD tools, and how this reported usage has changed in comparison with the existing body of research (RQ 4.1).

4.3.1 CI/CD Tools Usage

This section aims to reveal the diversity of CI/CD tools being used by respondents, and to determine which of them have been used more frequently by respondents (RQ 4.1.1). Overall, 31 different CI/CD tools have been reported by respondents. The full list of reported CI/CD tools can be found in Table 4.2. Throughout the chapter we use respondent IDs whenever we cite relevant quotes from them. In order to put these quotes in the right perspective, Table 4.2 also provides a mapping between these IDs and the CI/CD tools they reported having used.

Table 4.3 lists the 14 CI/CD tools that were used by at least two different respondents at some point in time, ordered in decreasing frequency of usage. One can observe the use of a large variety of CI/CDs, some of them being self-hosted (*e.g.*, Hudson, Jenkins), others being offered as a cloud service (*e.g.*, GitHub Actions, Travis, Bitbucket Pipelines) or both (*e.g.*, GitLab CI/CD) according to Table 2.1 in Section 2.3. In addition to the 14 CI/CD tools listed in Table 4.3, another 14 CI/CD tools were reported only once. These were, in alphabetical order: AWS CI/CD, Buildbot, BuildKite, Cirrus CI, Codefresh, Concourse CI, Heroku, Jacamar CI, Percy, Pulumi, Sauce Labs, Tekton, Vercel, and Zuul. Three respondents additionally reported resorting to *custom-built in-house* CI/CD solutions, since no existing CI/CD tool satisfied all of their needs. These solutions will not be considered in this chapter.

Table 4.3 also summarises which of the reported CI/CD tools are *currently* still being used by the respondents. In the light of the second research goal, we observe that GitHub Actions is the most frequently reported CI/CD tool by far, with the large majority of respondents (18 out of 22) using it currently. The opposite can be observed for Travis: nearly all respondents that were

Table 4.2: Mapping between CI/CD tools and respondents having reported to use them.

CI/CD tool	respondent IDs
GitHub Actions	1 2 4 5 6 8 9 10 12 13 14 15 17 18 19 20 21 22
Jenkins	1 2 4 5 7 8 11 12 13 14 15 16 17 18 21 22
Travis	2 4 5 8 9 10 11 12 13 16 18 19 20 21 22
GitLab CI/CD	1 2 3 4 5 7 8 11 12 13 14 15 16 22
CircleCI	2 4 6 8 9 10 12 13 18 19 20 22
Azure DevOps	2 4 5 9 12 14 15 17 18 19 20
AppVeyor	9 13 18 19 22
Hudson	4 5 6 15 22
TeamCity	1 7 13
Bamboo	2 4
Bitbucket Pipelines	7 20
Cruise Control	4 6
Drone	9 22
Netlify	10 14
AWS CI/CD, Codefresh, Concourse, Pulumi	5
Cirrus CI, Zuul	19
Heroku, Percy	8
Jacamar CI, Tekton	4
Buildbot	18
BuildKite	13
Sauce Labs	21
Vercel	10
in-house solution	6 12 18

using it at some point in time are no longer using it, despite Travis being still actively maintained. For instance, only 1 of the 15 respondents having used Travis is still using it. This corroborates the results of Golzadeh *et al.* (2021c) on the increasing popularity of GitHub Actions at the expense of Travis.

Jenkins falls in between GitHub Actions and Travis. It used to be a popular self-hosted CI/CD since the majority of respondents (16 out of 22) reported having used this CI/CD at some point during their software development experience. Yet, only 9 out of them are still using Jenkins. The reasons why such changes occurred will be explored later in Section 4.4.

GitLab CI/CD, CircleCI and Azure DevOps are three other popular CI/CD tools, having been used by at least half of the respondents, and are still used by most of them. On the other hand, none of the respondents are currently using Hudson or Cruise Control, two of the earliest commercial self-hosted CI/CD tools. The reason is that both Hudson and Cruise Control were discontinued

Table 4.3: CI/CD tools having been or being used by at least 2 respondents.

CI/CD tool	# all-time respondents	# current respondents
GitHub Actions	18	18
Jenkins	16	9
Travis	15	1
GitLab CI/CD	14	12
CircleCI	12	8
Azure DevOps	11	9
AppVeyor	5	3
Hudson	5	0
TeamCity	3	3
Cruise Control	2	0
Drone	2	2
Bitbucket Pipelines	2	2
Netlify	2	2
Bamboo	2	1

by their respective companies and replaced by a new alternative. For instance, Thoughtworks, the company owning Cruise Control, replaced it with a new CI/CD named Cruise in 2010. Since Cruise was not based on Cruise Control, the company decided to make the source code of Cruise Control publicly available after discontinuing its support. A few years later, Thoughtworks re-branded and renamed Cruise as GoCD, which was ultimately released as an open source CI/CD tool in 2014. Hudson, the open source Java-based CI/CD tool, used to belong to Sun Microsystems, until Oracle decided to acquire this company and to commercialise Hudson. The open source community reacted by creating Jenkins, an open source fork that became much more popular than its ancestor. Jenkins continued to grow and to increase its functionalities, while Hudson stagnated and ultimately became discontinued in February 2017.

Takeaways. To answer RQ 4.1.1, 31 distinct CI/CD tools have been reported by the respondents, of which 14 are used by at least 2 respondents. Some CI/CD tools, such as GitHub Actions, Jenkins, Travis, GitLab CI/CD, CircleCI and Azure DevOps were used by at least half of the respondents at some point in time. Only GitHub Actions and GitLab CI/CD are currently in this situation. While Travis and Jenkins were among the most used CI/CD tools, most respondents have stopped using Travis and, to a lower extent, Jenkins.

4.3.2 Main Reported Reasons for Using CI

This section aims to identify the reasons behind adopting CI/CD in software development projects (RQ 4.1.2). Based on a survey with several hundreds of developers, combined with interviews with 16 developers from 14 different companies, Hilton *et al.* (2017), Hilton *et al.* (2016) studied, among other aspects, the developer’s motivations and benefits of using CIs. They reported that developers use CI/CD for 8 different reasons: to help catch bugs earlier; to avoid breaking builds; to provide a common build environment; to deploy more often; to allow faster iterations; to make integration easier; to enforce a specific workflow; and to allow testing across multiple platforms. Many other qualitative studies have reported similar reasons for using CIs (Bernardo *et al.*, 2018; Duvall *et al.*, 2007; Fowler & Foemmel, 2000; Leppänen *et al.*, 2015; Rahman *et al.*, 2018; Ståhl & Bosch, 2013; Vasilescu *et al.*, 2015). The SLR (Soares *et al.*, 2022) mentioned the following reasons: improve software quality, stability, predictability, and transparency; faster build, integration, and release cycles; improve productivity, efficiency, and developer confidence; reduce workload; and detect and resolve defects faster. Our interview results align with these reasons since respondents reported adopting CI/CD to achieve the following goals:

- increased reliability: “*the intent was to ensure that we had reliable outputs. Whenever there’s a code change, we would know that it’s working.*” [R₁₇] and “*Basically there were plenty of people contributing, so [for] each pull request [we] needed to make sure that this pull request was not breaking the code and that the code was reaching production.*” [R₁₁]
- increased quality (e.g., through better reproducibility of bugs, increased testing, and performing quality checks): “*The reason was quality, we wanted to use CI/CD to run tests all the time and to deploy automatically to package automatically the software without depending on one person to do it. So the goal was really to have a common view on the build process of the tools and to improve the quality.*” [R₁₃]
- increased productivity: “*We very much invested into it that, if people want to contribute, they can really focus on the actual contribution, and there is very little overhead for them or fellow maintainers to do. We try to automate as much as possible.*” [R₁₀]
- faster delivery: “*The idea was to deliver value to the company in a quick time.*” [R₁₁]

- reduced cost and effort: “*The human costs have been reduced over time because of all the automation that arrived in those tools. [...] checked by the linting and [...] the maintainer does not need to do that extra work.*” [R₁₃]
- rapid feedback: “*[...] being able to have a quick feedback, it’s also why we work in parallel. We work to reduce the time of the pipeline so we can get early feedback for the people that contribute changes.*” [R₁₃]
- increased transparency of the build process: “*When we publish a release, people can check from which CI/CD tool it is coming. They can see the logs of the build. They can see that the build has not been tampered.*” [R₁₃]

While earlier research (Hilton *et al.*, 2017) revealed that developers consider security as a barrier for using CI, many of the respondents we interviewed actually mentioned *increasing security* by reducing security vulnerabilities as an important goal for CI/CD automation in their projects. For example, R₁ mentioned that “*on a more recent project we start to use Snyk, which is a tool for detecting vulnerabilities*”. R₂ referred to the importance of *DevSecOps* which is an approach to automation and platform design that integrates security as a shared responsibility throughout the entire development lifecycle: “*something that’s taking off right now is the addition of security in DevOps. It’s something called DevSecOps where in the DevOps pipeline we had static code analysis, or even dynamic code analysis and that’s one thing that is moving into DevOps area, which is how to integrate security operations and development.*” R₁₄ and R₁₅ highlight the importance of security testing/scanning: “*in my current organization we have automated security tests in our repo so when we want to deploy something in the production [...] we configured that thing to automatically test our application security and give us a report of what we need to fix and if the security tool fails*” [R₁₄], and “*We also have some scanning tools which we facilitate as a security scan to check the vulnerabilities or such things*” [R₁₅].

R₁₀ additionally reported that using a CI/CD enabled their open source project to retain contributors and attract new ones since CI/CD usage allows to reduce the maintenance overhead: “*We invested a lot of thought and time into how to attract and retain contributors. [...] we thought about how we can lower the barrier to contribute, but also remove as much overhead as possible based on the assumptions that as an open source project you really have to be fun, otherwise people will move on to other projects.*” It is interesting to note that this positive impact of CI/CD usage on contributor attraction and

retention was *not* confirmed in an empirical study by Y. Gupta *et al.* (2017) on 217 GitHub repositories using Travis. Surprisingly, they statistically observed that developer attraction and retention of a project were higher in the year *before* adopting Travis than in the year *following* Travis adoption. More research would be needed to ascertain the relationship between CI/CD usage and contributor attraction and retention.

*R*₈ reported the cloud-based nature as a reason for using Travis: it allowed the team to reduce cost and hardware resources, since they were able to use the CI/CD tools “as a service” compared to many other competing CI/CD tools at that time that mostly required self-hosting.

Takeaways. The main reported reasons for CI/CD adoption (RQ 4.1.2) are to increase reliability, productivity and security, to improve speed of delivery, and to reduce cost and human effort.

The reasons reported by interview respondents are in line with earlier findings in the scientific literature. For instance, they were reported in the SLR by Soares *et al.* (2022) and Elazhary *et al.* (2022). Savor *et al.* (2016) also reported that CI/CD tools allow software development companies to increase their team size by a factor of 20 and their code base by a factor of 50 without decreasing developer productivity or software quality. These findings are confirmed by our respondents who argued that CI/CD usage increases reliability, quality, and productivity. On the other side of the coin, interview respondents highlighted that adopting CI/CD tools introduces an additional layer of complexity into the development environment which needs to be carefully considered.

4.3.3 Activities Being Automated by CI/CD Tools

Section 4.3.2 revealed that CI/CD automation is used for different reasons (RQ 4.1.3). As a consequence, one may expect that CI/CD tools are used for a variety of activities, such as building or testing code, managing dependencies, etc. Vassallo *et al.* (2018) even suggested continuous refactoring as an additional activity to automate by CI/CD tools to control the complexity of software changes.

Table 4.4 reports on the activities that respondents reported for automation as part of their CI/CD tool usage, distinguishing between the activities that were initially automated as part of the CI/CD process, and the ones that were automated later on. As can be seen from the second column of Table 4.4, *build automation* is unsurprisingly the most popular activity (mentioned by 9 respondents) that is initially part of a CI/CD process. An equally popular

Table 4.4: Activities being automated by CI/CD tools based on respondents' ID reports.

activity	initially	added later
build automation	9 12 13 14 15 16 17 18 19	–
unit testing	1 2 3 4 5 6 7 9 19	13
integration testing	1 6 8	5 11 18 19
end-to-end testing	3 6	18
code coverage analysis	1 4 5 10	17 19
code quality analysis	7 10 13	2 6 15 17 19 21 22
security analysis	–	1 2 4 13 14 15 17 22
packaging and deployment	–	6 9 11 13 14 18 21
checking non-code artifacts	–	13 16 18 19 20 21 22
dependency management	–	8 13 19 21 22
comm. channel integration	–	16 21
license verification	–	17 21 22
other	1 2 16 20	6 21

activity (also mentioned by 9 respondents) is *unit testing*. Respondents also mentioned other testing-related activities during the initial phase of CI/CD usage, namely *code coverage analysis* (4 respondents), *integration testing* (3 respondents), and *end-to-end testing* (2 respondents). As an example of how respondents use CI/CD tools for testing, R_4 mentions: “we merged 600 pull requests from 170 people and I’m not going to run the tests manually, there’s no way to scale a project like that unless you have automation behind the testing”. Another 3 respondents report automated *code quality analysis* as one of their initial reasons for using CI/CD tools. For instance, “I use a lot of other stuff like linting, automated code formatting, coverage” [R_{10}]. Other reasons for initially using CI/CD tools were *generating documentation*, *server provisioning*, *checking browser compatibility*, and *creating multiple builds* (e.g., “we are also using an environment variable on GitLab CI/CD to use different configs for each project” [R_{16}]). Furthermore, R_9 reports using CI/CD tools to make sure open source contributions are not breaking any previous functionality in the program. This respondent emphasises that “it is especially important to do that for cross-platform programs, because developers usually only work on one system.”

Column three of Table 4.4 reports on those activities that had been added later on to the CI/CD automation process. The most popular of those activities was *security analysis*, being reported by 8 respondents. The *packaging and deployment* phases of the CI/CD process also tend to be added in a later

phase (7 respondents). The same holds for *code quality analysis*, mentioned by 7 respondents as being added later on to the CI/CD automation (as opposed to 3 respondents that started automating it in the initial phase of using a CI/CD tool). More advanced testing activities (beyond unit testing) were also reported more frequently to be added later on.

Other activities that were reported to be added later on to the CI/CD automation were *checking non-code artifacts* (7 respondents), *dependency management* (5 respondents), *license verification* (3 respondents), and *integration with communication channels* (2 respondents). For instance, R_{16} integrated the CI/CD tool with a Slack communication channel: “*I also integrated our CI/CD with Slack. After the build is successful and the APK is generated successfully, we upload the APK to a different channel of Slack for our customer or testers*”. R_{21} uses this integration to learn about forced pushes: “*We have one that notifies our Slack channel when someone forced pushed*”.

Related to *checking non-code artifacts*, R_{13} reports checking the format of commits to be the same as the expected commit format. R_{16} , R_{20} , and R_{22} reported using linting tools for checking non-code related artifacts. Additionally, R_{18} reported, “*doing style checks, formatting checks of the files, and also typing checks*”. R_{19} indicated “*in addition to these CI/CD services that run our particular jobs, there’s also these services that do, for example, code analysis that may be not exactly CI/CD services, but they are services that run and do things on the code based on commits. Maybe they would qualify as CI/CD services. [...] That’s sort of a popular thing these days, for example, to do a static code analyzer service*”.

The following activities were mentioned by only one respondent to be added later on to software automation:

- software verification: “*as a team grows to 10, 15, 20, 40 people, now, it becomes a place to introduce constraints and system checks and verifications that go beyond what you could do from tribal knowledge. So now it almost becomes a system where you take the guidelines that you would write down in a document and you put them into automation.*” [R_6]
- labelling/closing pull requests: “*We have a GitHub action that labels pull requests with the appropriate labels*” [R_{21}] and “*Someone just added one in the last week that closes stale pull requests.*” [R_{21}]
- detecting inactive contributors: “*So I wrote a couple of scripts that are run via GitHub action. [...] It finds anybody who hasn’t landed or reviewed a commit in the last 18 months and flag them as a collaborator*”

that should probably be removed and opens a pull request to remove them.”
[R₂₁]

Takeaways. To answer RQ 4.1.3, CI tools are initially used for basic CI/CD tasks like build automation, automated unit testing and code coverage analysis. More advanced activities are added later on to the CI/CD automation, such as more advanced testing activities, security analysis, code quality analysis, dependency management, packaging, and deployment.

These insights align with the findings in the research literature. For example, Soares *et al.* (2022) report that CI/CD automates boring repetitive tasks such as basic automated building, testing and deployment. Our findings also confirm the initial reasons for CI/CD usage reported by Savor *et al.* (2016) who studied the usage of CI/CD tools in two different companies, as well as the initial reasons reported by Elazhary *et al.* (2022) who studied the use of CI/CD tools in three different organisations. In addition, they report that more complicated automation tasks tend to be added later on, such as testing in a clone of the production environment. Specifically in the context of GitHub Actions, Kinsman *et al.* (2021) and Decan *et al.* (2022) report that many of the reusable Actions support the basic CI/CD activities of building, testing and deploying.

4.3.4 Most Valuable Features of CI/CD Tools

This section aims to understand why project maintainers rely on specific CI/CD tools (RQ 4.1.4). In other words, we are interested in knowing the most valuable features offered by the CI/CD tools that have been used by respondents, as these features are likely to play an important role in why these CI/CD tools have been used in their projects. We therefore asked each respondent what were the most valuable features of the CI/CD tools they had used.

Table 4.3 already revealed a difference between self-hosted and cloud-based CI/CD tools, and between open source and commercial solutions. These differences may have played a role in the choice of CI/CD tools by some developers.

Table 4.5 summarises the most valuable features of each CI/CD tool, as reported by the respondents that use them. Only tools for which at least two valuable features had been reported are listed. It is worth noting that these features have to be interpreted in their historical context: they do not necessarily reflect what are the *current* valuable features offered by a CI/CD

Table 4.5: Most valuable features of CI/CD tools as reported by respondents.

valuable features	GitHub Actions	Travis	Jenkins	Azure DevOps	GitLab CI/CD	CircleCI	Drone	Hudson	TeamCity	AppVeyor
good integration with hosting platform	1 2 5 6 9 10 11 13 14 16 17 18 19 21 22	10 13 19 22		12 14 15 17	1 2 3 5 11 13 16	13				
ease of use	1 2 10 11 16 17 21 22	8 22	1 2 8 22	14 15 17 18	16 22	13	9 22	15	1	
support for specific architectures/OS	9 14 19 22	14	16 18 21	9 12 19	9	12 13 19 22	9			9 13 18 19
popularity / familiarity	9 10 16 18 22	2 8 10 11 12 13 19 21 22	1 8 11 16 17 18	14				15		
good free tier	2 5 9 10 13 14 17 21 22	5 8 9 13 16 21 22		9	3 5 11					
good plugin support	6 11 13 15 21 22	8	2 13 14 17	14 15 17						
self-hosting ability	6		16 21		2 4 14		9	22	1	
useful features	5 9		11	9 15	13					
customizability	5 11 19 22		14 17							
security	18	8				18				
speed	5 11 22					13				

tool, but they reflect what were these features when the respondents used the CI/CD tool. Due to the qualitative nature of our analysis, the table is inevitably incomplete, since the absence of a respondent mentioning a valuable feature does not imply that the feature is absent from the CI/CD tool. As a consequence of this, the valuable features of less popular CI/CD tools are less likely to be mentioned, simply because there were fewer respondents to report about them. Below we report on the valuable features listed in Table 4.5.

Good integration with hosting platform. Many projects use a CI/CD tool on top of a hosting platform (such as GitHub, GitLab, BitBucket or Azure) that is used to store and manage the project development history. In those cases, it is important for the CI/CD tool to be well integrated into the hosting platform

in order to make it as easy as possible to configure and use the CI/CD tool. Table 4.5 shows that respondents appreciated the good integration of GitLab CI/CD into GitLab, the good integration of Travis and GitHub Actions into GitHub, and the good integration of Azure DevOps into Azure.

Ease of use. A good user experience makes the use of CI/CD tools easier, smoother and more enjoyable. As such, it was considered by many respondents as one of the most valuable features of the CI/CD tool they were using. They mentioned a variety of factors that affected the ease of use of CI/CD tools. One such factor was the simplicity of the user interface. In addition, the presence of good documentation was also important, since it helps developers find and use the available features to their full potential. Yet another one was the ease of configuring the CI/CD tool or the workflows or pipelines created with it, for example by providing the ability to use default settings for configurations. The variety, clarity, and above all, the stability of the available configuration options also affected the ease of use.

Support for specific architectures and/or operating systems. CI/CD tools enable building and deploying software in specific environments, and facilitate the deployment on multiple environments. These environments typically include a particular operating system (*e.g.*, Ubuntu or some other Linux variant, macOS, Windows, Solaris, FreeBSD) and a specific hardware architecture or processor (*e.g.*, Intel or ARM CPUs, and AMD GPUs). Since the required environments may strongly vary from one project to another, and since not every CI/CD tool supports all possible environments, this may affect the choice of a CI/CD tool. For example, the ability to support Windows builds was the main reason for four respondents to use AppVeyor at the time when most of the other CI/CD tools did not provide any (or any decent) Windows support. CircleCI was also particularly appreciated by respondents for its support for a wide variety of different build environments.

Popularity and familiarity. Several respondents reported using specific CI/CD tools in their project out of *familiarity*. Often, developers just continue to use tools that have already been in place in the project based on some earlier decisions by former project maintainers. A related frequent reason to prefer some CI/CD tool over another one is because of its *popularity*. If some tool is more popular than another one, it becomes more likely that it will be found or recommended by someone. Popularity was one of the main reasons raised by respondents for choosing Travis or Jenkins. Until GitHub Actions entered the landscape, Travis remained the default choice for software projects hosted on GitHub, while Jenkins used to be the default choice for Java projects (Golzadeh *et al.*, 2021c).

Good free tier. Most CI/CD tools are commercial, requiring their customers to pay for the services they offer. On the other hand, many CI/CD tools also provide what is called a *free tier* or *free plan* of their cloud-based service. This allows projects (mostly open source projects) to use the cloud resources to run the CI/CD for free. Depending on the CI/CD tool, the free tier may impose limitations on the number of supported users/projects, the number of minutes to execute builds, the number of monthly builds, the computing resources, type of OS, and/or the number of jobs that can be executed in parallel. Sometimes, the free tier also restricts the available functionalities of the CI. Table 4.5 shows that respondents particularly appreciated the free tier offered by Travis, GitLab CI/CD and GitHub Actions. As will be discussed later, the restrictions imposed on the free tier may change over time and cause projects to migrate to other CI/CD (cf. Section 4.4.2).

Good plugin support. Many respondents found it valuable that several CI/CD tools come with the possibility to create and use reusable components for creating CI/CD workflows or pipelines. For example, GitHub Actions distributes a large set of Actions on the GitHub Marketplace, CircleCI comes with a public registry of reusable Orbs, and Jenkins provides a large index of community contributed plugins. The amount, quality and availability of these reusable components determine to which extent a CI/CD tool can be considered to feature good plugin support.

Self-hosting ability. As can be seen from Table 4.3, some CI/CD tools can be self-hosted and be used “on premises” without needing to resort to any cloud-based service. Some companies prefer to use a self-hosted CI/CD solution because it offers increased security, since it reduces the risk of company-sensitive information getting exposed or even compromised through cloud-based solutions: “*You can run self-hosted runners, which is a way for you to run on your own machines, but then you need to implement a whole ecosystem of security constraints because you can potentially be running arbitrary third-party code in your data center, so you need to make sure that you’ll lock down that environment to make sure that the environment itself is actually secure. That’s a significant investment.*”

Useful features. Many respondents reported useful features related to their CI/CD tool of choice. For example, R_5 reported being happy with GitHub Actions since it included many useful features since its beginning, and more features are added regularly to continue making it a better tool. R_9 appreciated GitHub Actions’s artifact upload support: “*they give you like 5 gigabytes or something of storage. And your CI/CD run can upload some files. And as an administrator of that CI/CD pipeline, you can access that file and download it*

like a compilation output or something.” For the same reason, R_9 appreciated Azure DevOps. R_{15} liked the access to deployment history in Azure DevOps: “you had a facility when you wanted to go back in time and just deploy one release that you had, for example, one year ago. You could go to the history and just click on the history and redeploy that”. R_{11} appreciated Jenkins’ versatility: “it’s really powerful, and you can do plenty of things.” R_{13} liked GitLab CI/CD’s ability for each repository to have its own pipeline, combined with the concept of cross-project CI/CD with multi-project pipelines.

Customisability. Several respondents considered customisability as a valuable feature, even if the interpretation of this concept varied a lot depending on the considered CI/CD tool and respondent. The customisability of GitHub Actions was mostly referred to as the ability to use this tool for non-CI related stuff like updating the Slack channel based on the results of the runs. Respondent R_{22} even claimed that GitHub Actions had “more options of customisation” in comparison with other CI/CD tools. Two respondents reflected on the customisability of Jenkins, appreciating its ability to customise the user interface with different themes. *Speed.* Fast building and running times were mostly valued for GitHub Actions (three different respondents). Respondent R_{13} particularly valued the speed of CircleCI due to its facility for creating complex parallel pipelines. The ability to run multiple pipelines in parallel can lead to significant speed improvements.

Security. Security aspects are of crucial importance for CI/CD tools since the automation task they support has the potential of being used by a large number of projects and developers worldwide. This huge attack surface might cause security issues to escalate very quickly. As a valuable feature of GitHub Actions and CircleCI, R_{18} explained their ability to secure user credentials from being accessible by other developers: “In CircleCI there is a way you can build the artifacts to a staging area and then you can move the artifacts with the second workflow to where you want to deliver it. That’s what we do with GitHub Actions too. We build in one workflow and we have a second workflow which does the upload, shipping or delivery with credentials.” The availability of a public registry of third-party plugins for the CI/CD tool also introduces an important potential risk (Decan *et al.*, 2022), since there is little control over the contents of these plugins. For this reason, R_8 valued the way Travis avoids this problem by only providing closed-sourced plugins that are verified by the company itself, therefore reducing the risk of introducing malicious code.

In the following, we present this set of valuable features from the point of view of specific CI/CD tools. Such information will be useful in the context of

Section 4.4.1 to understand why developers decide to use multiple CI/CD tools simultaneously (*e.g.*, because they have complementary valuable features) and Section 4.4.2 to understand why developers decide to migrate to a different CI/CD tool (*e.g.*, to benefit from valuable features of this CI/CD tool).

GitHub Actions. The most recent CI/CD in the list seems to have attracted a lot of attention from respondents for multiple reasons. Since it was developed by GitHub itself, it naturally has very good integration into GitHub. The popularity of the GitHub platform itself among open source developers was reported as a determining factor of choice by 5 respondents. Respondent R_6 decided to select GitHub Actions out of familiarity: “*we made the decision at the time that we better move to GitHub instead of Azure DevOps because of the developer familiarity with GitHub over Azure DevOps as a system. So the trade-off there was developer familiarity.*” Moreover, GitHub Actions offers free runners, supports a wide range of operating systems (including Linux, Windows, and macOS), and was praised for its ease of use (“*GitHub Actions are so easy to use for CI*” [R_{21}]), its good plugin support through a wide range of actions available on the GitHub Marketplace, its support for many different languages (including JavaScript, Ruby, and Python), its reliable runners providing fast builds, its support for self-hosted runners, as well as its security mechanisms to avoid exposing user credentials.

Travis. Respondents appreciated Travis’ good documentation and the availability of many built-in features. Many respondents appreciated its good integration into GitHub. Indeed, given that its integration with GitHub used to be better than the other available alternatives, Travis used to be the default choice for GitHub repositories at the time. Nowadays, GitHub Actions has outperformed Travis in terms of integration with GitHub. Many respondents also praised its good free tier support at the time they were using it (often many years ago). R_8 reported a clear and simple interface, easy configuration, and good default setting for Ruby projects. R_{22} agreed that “[*Travis*] was extremely easy to set up with one single configuration file at the root of the project”.

Jenkins. Some of the valuable features of Jenkins that were praised by respondents were its ease of use, its customizability, the availability of many plugins, and it brings a good user experience, even for non-technical users. It also offers self-hosting ability, which is attractive to companies that want to exert full control over the CI/CD automation, notably in order to comply with their service-level agreements related to downtime, service provider response time, security, and turnaround time.

Azure DevOps. This CI/CD tool was mostly reported by respondents working

in companies that had a contract with Microsoft for their infrastructure. The valuable features of Azure DevOps were its integration with other Azure tools (4 respondents), good standard built-in features, good support for plug-ins, a better integration with GitHub compared to CircleCI, good runners for Windows and macOS, easy step-by-step configuration, a history of work items and deployments¹, ease of defining multiple build environments, and good separation between CI and CD configuration. With respect to the latter feature, *R*₁₈ specifically appreciated Azure DevOps' "Release pipelines"² as a way to facilitate the deployment automation.

GitLab CI/CD. Unsurprisingly, GitLab CI/CD was only mentioned by developers using the GitLab social coding platform. Respondents appreciated its good free runners, a secure debug process, its ease of use in comparison with Jenkins for using private resources, its support for Docker containers. They also appreciated its self-hosting ability which distinguishes GitLab CI/CD from its competitor GitHub Actions.

CircleCI. Respondents specifically appreciated CircleCI's support for Windows, macOS, ARM, and Docker containers. They also valued its user interface with good visualisations, its nice feedback loop with GitHub, its speed, its facility for creating complex parallel and conditional pipelines, and the concept of workspaces.³

Drone. The reported valuable features for Drone were its support for ARM architectures, its self-hosting ability and an intuitive interface.

Hudson. *R*₂₂ reported having used Hudson for a long time for closed source projects and private repositories, and appreciated the CI's self-hosting ability. *R*₁₅ appreciated the easy setup and configuration because Hudson was developed in Java and the team had experience working in Java environment.

TeamCity. Only one respondent reported on the valuable features of TeamCity, valuing the user-friendliness of the tool, as well as its self-hosting capabilities.

AppVeyor. This CI/CD tool was reported by four different respondents as the CI/CD tool that historically used to have the best support for Windows. Since no other valuable features were reported for AppVeyor, this seems to be the main reason that caused developers to use it.

Other CI/CD tools. Considering the CI/CD tools being mentioned by sin-

¹See <https://docs.microsoft.com/en-us/azure/devops/boards/queries/history-and-audit> and <https://learn.microsoft.com/en-us/azure/azure-resource-manager/templates/deployment-history>

²<https://docs.microsoft.com/en-us/azure/devops/pipelines/release/?view=azure-devops>

³<https://circleci.com/docs/2.0/workspaces>

gle respondents (and hence not shown in Table 4.3), R_5 valued Concourse CI because of its good visualisation and ability to set personal triggers for pipeline activation. The ability to have completely independent pipelines in Concourse CI also enables to connect multiple repositories or Docker images to one pipeline and use user-defined or pre-defined triggers to start the pipeline. Percy was praised by R_8 as the only available CI/CD tool with specific visual CI/CD abilities: “*It basically would render your web page or you define a number of views that you want to test, take a screenshot essentially and then in your branch that you’re working on it would do the same and it would compare the screenshots between the branches [...]. If you’re assuming when you deploy or when you make a change, if you’ve broken something, say in your CSS, then you could have some visual bugs that wouldn’t show up in the automated testing. So this is a good way of catching some of those more obscure CSS bugs*”.

Takeaways. The CI/CD tools that were most popular among respondents came with the biggest set of valuable features as the answer of RQ 4.1.4 (such as ease of use and support for a wide range of hardware architectures and operating systems). With the exception of Jenkins, the most valued CI/CD tools are cloud-based solutions (GitHub Actions, Travis, Azure DevOps and GitLab CI/CD) that come with a good integration with their hosting platform and a good free tier. These solutions also feature a good support for reusable plugins, with the exception of GitLab CI/CD that on the other hand offers a self-hosting ability.

Less popular tools were still considered valuable because they offered specific features that were not available at the time in the other competing CI/CD tools, such as Windows build support, visual testing for UI design, or support for multiple repositories in a single pipeline.

These results differ from earlier studies in that we provide a tool-specific analysis (Hilton *et al.*, 2016; Kinsman *et al.*, 2021; Widder *et al.*, 2018). We also observe a clear shift of the CI/CD landscape towards more cloud-based solutions, with a free tier offer for open source projects, tight integration in the social coding platform, and a registry of reusable components to facilitate creating CI/CD workflows.

4.3.5 Reported Shortcomings of CI/CD Tools

We asked respondents about the shortcomings they experienced in the CI/CD tools they had used (RQ 4.1.5). Table 4.6 reports on these shortcomings,

Table 4.6: Shortcomings of CI/CD tools as reported by respondents. (CI/CD tools that were used by only one respondent are not listed in the table.)

shortcomings	GitHub Actions	Travis	Jenkins	GitLab CI/CD	CircleCI	Azure DevOps	Other
hard to configure	8 15		6 11 12 17 21	8			1:TeamCity
too slow	10	5 18 19 22	16	22			22:Hudson
unsatisfactory user experience	2 5 8 10 15 18	8	16	2 22		17	
restrictions of free tier	4 21	4 9 10 11			22		
security issues	6 8 15	13					4:Bamboo
lack of scalability		18	1 12	4 13			
plugin problems	8	8	12			15	
architectures/OS support problem	4 21	19					
feature stagnation		4 13					9:AppVeyor, Drone 5:Concourse CI
lack of reliability		4 5 8 9 13 21					
insufficient access to logs	15						13:TeamCity
lack of GitHub integration						9	19:Zuul

grouped into various categories that we described hereafter. Some reported shortcomings were considered so severe by the respondents that they caused the project to migrate to a different CI/CD tool. Those migration reasons will be discussed in more detail in Section 4.4.2.

It is worth to mention that this list of shortcomings is inevitably incomplete, since respondents may have forgotten to report some shortcomings while focusing on the major ones they had experienced. Moreover, it may be the case that some reported shortcomings are no longer relevant today, given that CI/CD tools continued to evolve and improve.

Hard to configure. Configuration difficulties were reported for several CI/CD

tools. The initial build configuration was reported to be difficult to create in TeamCity. GitLab CI/CD did not have a simple workflow. GitHub Actions was reported by two respondents to be difficult to configure because too many options are available, and because of the lack of an appropriate default initial configuration. Jenkins was reported by five respondents as difficult to configure.

Too slow. Given that speed is considered as a valuable feature of CI/CD tools (see Table 4.5), it is not surprising that many respondents mentioned slow runners as a shortcoming of some CI/CD tools. This was the case for Hudson, GitLab CI/CD, Travis and Jenkins. One respondent also mentioned that even GitHub Actions was too slow for their specific needs, even though three other respondents explicitly acknowledged the speed of GitHub Actions as a valuable feature.

Unsatisfactory user experience. Different CI/CD tools were reported to have an unsatisfactory user experience for a wide variety of reasons. Jenkins was reported to have an outdated user interface design. GitLab CI/CD was reported to have a cluttered user interface and no web interface for defining workflows.

For Travis, one respondent reported a bad user experience since most configuration tasks for integrating the CI/CD tool into GitHub needed to be done manually. For Azure DevOps, one respondent regretted the absence of YAML-based configurations of workflows. For GitHub Actions, two respondents mentioned a too sparse user interface for workflow configuration and four respondents reported no good visualisation of workflows. Additionally, one respondent mentioned the difficulty to start using GitHub Actions due to insufficient documentation (especially in the early days of GitHub Actions).

Restrictions of free tier. Respondents reported restrictions imposed by the free tiers of CI/CD tools on the build time, the amount of available memory, and the number of runners that could be used in parallel. This was the case for Percy, CircleCI (which did not support macOS under its free tier), GitHub Actions and Travis. Travis in particular was agreed upon by many respondents to have imposed many restrictions on its free tier after the company's decision to change its policy towards support for OSS projects. The reasons for these imposed restrictions will be discussed in detail in Section 4.5.3. In a nutshell, Travis replaced its free tier for OSS project builds, that used to offer a fixed number of minutes *per month*, with a higher fixed number of minutes *for life*. At the same time, Travis restricted the set of projects that they qualify as OSS, as reported by *R*₁₉: “*because how they defined open source, they wouldn't even define [OUR PROJECT] as an open source project [...] because according to their requirements, if someone was paid to work on the project like I am, it*

wouldn't qualify for the open source tier at Travis." Given that OSS projects have a very limited budget, R_{19} saw no other choice but to migrate to another CI/CD tool.

Security issues. Several respondents mentioned security concerns related to CI/CD usage. They did so for Travis, GitHub Actions and Bamboo, but any other CI/CD tool is likely to suffer from security issues to some extent. Travis was reported by R_{13} to lack correct communication about an important data breach "*they had a security breach [...] and they did not communicate properly about this*". GitHub Actions was reported by three respondents to have security issues related to working with credentials and self-hosted runners. R_6 explains that "*if somebody already developed a [GHA] Action, you can just plug it into your project and that works great for an open source project because the software is open sourced. You're not worried of the vulnerability and GitHub takes on the responsibility for all the security problems that you would kind of encounter if you tried to run your own CI/CD platform.*" However, "*you can't do that in enterprise. You basically don't trust your software to run on anybody else's machines, or on virtual machines you don't control.*" In addition to this, the reliance on reusable components (*e.g.*, Actions) to automate development activities in software projects increases their attack surface considerably.

Lack of scalability. Scalability refers to the ease of seamlessly and transparently increasing the capacity of CI/CD tools to accommodate for bigger builds, for example by offering longer build times, more parallel runners, and more computing resources (processing power and memory) for the CI/CD process. Scalability issues were reported for three CI/CD tools: Jenkins, GitLab CI/CD and Travis that was reported by one respondent to have a memory bottleneck. Respondent R_6 acknowledged that scalability necessarily comes at a certain cost. Moreover, it is more difficult to achieve in self-hosted solutions, compared to cloud-based CI/CD tools: "*Often you evaluate for capability. Hey, is this system going to be able to do what we need to do? You evaluate for its scalability, meaning yes, it can run one build, but can it run 1000 builds per day? And third is going to be cost, and that comes in two flavors. There's the actual out of pocket operational expenditure of running this system. And then there's the maintenance and continuous support for the system from the developer or maintainer perspective. Usually one of those evaluation criteria is not met by the target system for whatever reason. Most frequently it's a scalability one.*" Respondent R_9 reported another scalability issue related to data bandwidth and the absence of automated caching of compiler outputs: "*If you run CI/CD a lot of times, you download quite a lot of data from the Internet. Because you have packages that you install during your CI/CD run [...] and you basically*

use up a lot of bandwidth and data. And, again, when you compile software, for example C++ projects, it takes up a lot of time because it's a computationally intensive process."

Plugin problems. Problems with plugins were reported for multiple CI/CD tools for different reasons. Jenkins was reported to be too barebone, requiring the user to need many plugins from the start. For Travis, plugins are only updated by the company itself, providing limited freedom to the user. Azure DevOps was reported not to have the ability to write and customize plugins (as is possible in GitHub Actions, for example). In case of GitHub Actions, R_8 reported not being fond of having community plugins and preferred those CI/CD tools that only offer built-in plugins: "*Travis did provide [...] a good amount of things already installed on the CI/CD machine. So I didn't need to use [...] all of the different plugins that you can use. [...] Most of the GitHub Actions plugins are kind of community ran on open source repositories. That makes me very nervous of using them.*"

No support for specific architectures or operating systems. R_{19} regretted Travis' lack of support for the FreeBSD OS and ARM hardware architectures. R_{21} regretted GitHub Actions's lack of support for specific OS and hardware architectures, and R_4 regretted the absence of support for HPC binaries. Several respondents agreed that many CI/CD tools have recently become better in supporting the major operating systems (Linux, Windows, macOS). Still, most CI/CD tools remain limited when it comes to less common operating systems (*e.g.*, Solaris and FreeBSD) and hardware architectures (*e.g.*, specific GPU processors). Moreover, as pointed out by R_{19} , some projects require to build and deploy on such a wide diversity of OS that no single CI/CD tool is able to satisfy these needs: "*[name of project] is a portable project. It runs on so many architectures and operating systems that we don't have nearly that coverage in CI/CD services*".

Feature stagnation. Four CI/CD tools (Travis, Drone, AppVeyor, and Concourse CI) were reported by respondents as suffering from a lack of new features being introduced, causing projects to move away from them. For example, R_5 reported that: "*one of the drawbacks of Concourse CI is that I don't see a lot of active development anymore on the tool itself. [...] It has been in development for a number of years, but now in the past year, it also became stagnant.*"

Lack of reliability. Travis was the only CI/CD tool reported by many respondents to have become less reliable for a variety of reasons. Two respondents reported a decrease in service quality. Respondent R_8 complained about the company changing its way to support webhooks "*They stopped all of their webhooks that basically just stopped doing CI/CD for almost all of my projects on*

any builds.” R_5 reported problems with the CI’s customer service, since they took a long time to answer or not even answering about quality of service problems. Two respondents complained about the flakiness of Travis. For example, R_4 stated that he was “*seeing a bunch of reliability problems in Travis where jobs would flake out and we would have to rerun them.*” Two other respondents mentioned unreliability of Travis without pinpointing specific reasons.

Insufficient access to logs. For TeamCity, respondent R_{13} regretted not having access to build logs and build results: “*we did not have access to the build logs and the build results. It was quite painful to not have that feedback loop, but still knowing it was running in the background.*” For GitHub Actions, respondent R_{15} reported the absence of a deployment history as problematic: “*GitHub Actions does not support the history. In Azure DevOps I remembered we had the history. I mean that you had a facility when you wanted to go back in time and just deploy one release that you had.*”⁴

Lack of GitHub integration. R_{19} reported a lack of integration with GitHub for Zuul: “*their integration with GitHub has some kind of flaw. In many cases when we run CI/CD jobs on Zuul they don’t show up like the other jobs do on GitHub.*” Azure DevOps was also reported by one of the respondents to lack proper integration with GitHub.

Takeaways. Different CI/CD tools suffer from different shortcomings (RQ 4.1.5), such as configuration problems, slowness, unsatisfactory user experience, restrictions on its free tier, security issues, insufficient scalability, plugin problems and many more. Travis was considered to be the most problematic by respondents, mainly suffering from lack of reliability, slowness, restrictions on its free tier, and feature stagnation.

GitHub Actions was also reported to exhibit several shortcomings, mostly in relation to an unsatisfactory user experience and security issues, as well as some missing desirable features. For Jenkins, the main reported shortcoming was its configuration difficulties.

These findings are in line with those of earlier studies. Without focusing on CI-tool-specific shortcomings, Hilton *et al.* (2017) identified general shortcomings of CI/CD usage, such as configuration problems, slowness, security issues, and lack of good integration. They identified some additional shortcomings that were not reported by our respondents such as the difficulty of

⁴Azure’s Deployment History feature enables to go back in time to allow to redeploy a release that was for example available one year ago, just by selecting that deployment in the

troubleshooting CI/CD build failures. On top of this, Elazhary *et al.* (2022) identified some other shortcomings such as lack of features for UI testing, bottlenecks for committing due to PR reviews, and scalability issues due to resource restrictions. One of the shortcomings that we did not observe in earlier studies were the plugin problems mentioned by several respondents. Specifically for Travis, our findings are in line with Widder *et al.* (2019) who reported Travis being slow, having unsatisfactory user experience, not supporting specific architecture or OS, and feature stagnation. Our respondents reported all these shortcomings, as well as several others. Specifically for GitHub Actions, Kinsman *et al.* (2021) reported discussions about problems and frustrations with broken builds, errors and other problems. However, they did not discuss these shortcomings in depth, making it difficult to compare them with our own findings.

4.4. Co-Usage of CI/CD Tools and Motivations for Migration

This section tackles the second research question of this chapter, aiming to understand the reasons for using different CI/CD tools together, as well as the reasons and difficulties for migrating to another CI/CD tool (RQ 4.2).

4.4.1 Using Multiple CI/CD Tools Simultaneously

In their quantitative study of CI/CD usage in 92K GitHub repositories, Golzadeh *et al.* (2021c) found that co-using CI/CD tools (*i.e.*, making use of several CI/CD tools at the same time) is common practice. This is surprising since one might intuitively expect all CI/CD tools to provide similar services. We are not aware of any published qualitative analysis aiming to understand the reasons behind such co-usage. We therefore inquired the interview respondents about the reasons behind this phenomenon. 13 out of 22 respondents confirmed that, in at least one of the projects they were involved in, multiple CI/CD tools were being used simultaneously. In this section, we explore all CI/CD co-usages that have been mentioned by the respondents in order to understand the need for such co-usage (RQ 4.2.1).

Figure 4.2 reports on the number of respondents making use of multiple CI/CD tools simultaneously.⁵ We observe that the co-usage of CI/CD tools

history. See <https://learn.microsoft.com/en-us/azure/azure-resource-manager/templates/employment-history?tabs=azure-portal>.

⁵Whenever 3+ CI/CD tools are used together, each pair of CI/CD tools is reported

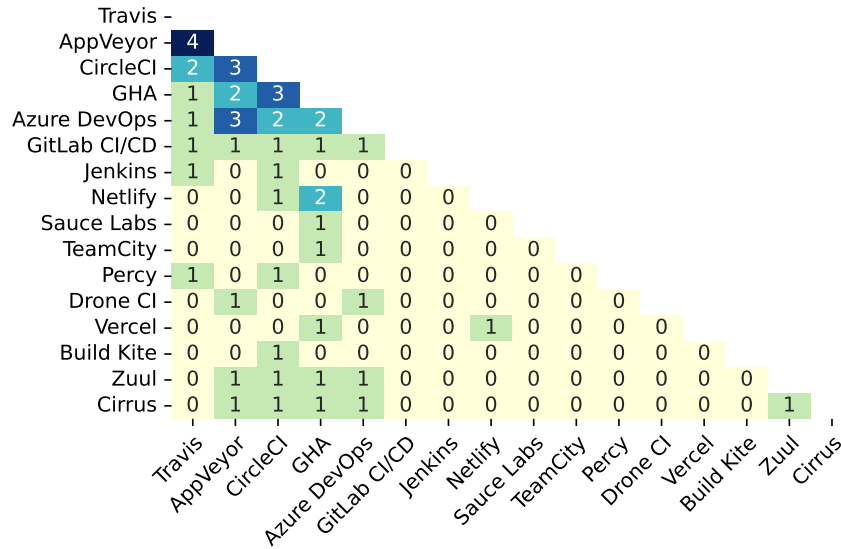


Figure 4.2: Number of respondents co-using a pair of CI/CD tools

is not restricted to the most popular ones. The combination of (Travis, AppVeyor) was reported 4 times, and the combinations of (CircleCI, AppVeyor), (GitHub Actions, CircleCI), and (Azure DevOps, AppVeyor) were reported 3 times. These findings corroborate the ones of Golzadeh *et al.* (2021c) that already observed that Travis and AppVeyor was the most frequent case of co-usage, and that Travis, AppVeyor, GitHub Actions and CircleCI were involved in most (92.1%) co-usages.

Table 4.7: CI tool co-usage reasons reported by respondents.

reason for CI/CD co-usage	respondent IDs
Multiple operating system support	4 9 13 19 22
Complementary functionality	8 10 13
Having a backup CI/CD tool	1 18 19
Countering resource limitations	4 19 21
Specific hardware architecture support	4 13
Testing a CI/CD for potential migration	1

Focusing on the need for co-using CI/CD tools, Table 4.7 summarises all reported reasons for co-usage of CI/CD tools. We observe that *supporting mul-*

multiple operating systems is the most frequently mentioned reason for co-usage (5 respondents). Most of the respondents mentioned the need to build software products at least for Linux, macOS and Windows. Older versions of many CI/CD tools had limited support for some of these OS. For many years, most CI/CD tools have only supported a single operating system (usually Linux, macOS or Windows), and that is the reason why R_9 reported to use AppVeyor for Windows builds, and Travis for Linux and macOS. Nowadays, most CI/CD tools support the three main operating systems. As an example, GitHub Actions initially started with support for Linux only, and added support for macOS and Windows later on. In the case of Travis, support for macOS was added since April 2013⁶ and support for Windows in October 2018⁷. For GitLab CI/CD, Windows runners were added in beta version in January 2020⁸ and macOS support was added in August 2021.⁹

Additionally, some respondents required support for more specific operating systems. For instance, while R_{13} used CircleCI for Linux builds, this respondent required BuildKite for FreeBSD builds. Two respondents also mentioned the need to support specific hardware architectures (*e.g.*, specific CPU or GPU processors) such as ARM64 and specific AMD or Intel processors. R_9 reported using Drone for ARM 64 CPUs, and lately also Azure DevOps to leverage better support for these targeted platforms. R_4 , who is involved in offering HPC as a service, has to use many CI/CD tools simultaneously in order to support the specific architectures required by their project: “*We need GPU nodes. We need AMD GPUs which no cloud has. We need Intel GPUs which no cloud has.*”

The fact that CI/CD tools tend to offer *complementary functionalities* is another reason for co-using CI/CD tools. This could be either because the CI/CD tool put in place does not offer some specific features required by the project, or because these features cannot be used in the way the developers expect. As an example, R_8 reported using Percy in complement of Travis and CircleCI because Percy is one of the few CI/CD tools that provides support for *visual testing*, a technique that helps developers to ensure that a graphical user interface appears to the end-user as originally intended. Another example reported by R_{10} and R_{13} concerns Netlify and Vercel, two CI/CD tools that are designed specifically for deployment of web applications. They notably

individually.

⁶<https://saucelabs.com/blog/announcing-travis-ci-for-mac-and-ios-powered-by-sauce-labs>

⁷<https://blog.travis-ci.com/2018-10-11-windows-early-release>

⁸<https://about.gitlab.com/blog/2020/01/21/windows-shared-runner-beta>

⁹<https://about.gitlab.com/blog/2021/08/23/build-cloud-for-macos-beta>

facilitate the dynamic scaling of the application based on the number of connected users, or based on the number of database requests. R_{10} relies on both Netlify and Vercel in complement of GitHub Actions: “*If this is some kind of website or web app then we use Vercel or Netlify for the deployment aspect of it.*” R_{13} also co-uses Netlify alongside CircleCI and GitHub Actions for a better overall CI/CD experience: “*we are very happy with the resources that we have in CircleCI [...], but we are also happy with the integration that we have in GitHub, the caching that we can have in GitHub Actions and we are also very happy with the specialised nodeJS features that we get at Netlify*”.

Another frequently reported reason for co-using CI/CD tools is *having a backup CI/CD tool* in the case the main CI/CD tool being used becomes out of order. R_{19} indicated relying on a custom-built in-house CI/CD tool “*so we still have that too as a sort of additional backup way of testing stuff*”. R_{18} reported they kept using Travis alongside CircleCI and Jenkins as a backup for six months until the project team decided that it was no longer needed.

Some respondents mentioned the need for more (free) resources as the reason to co-use CI/CD tools. For example, R_4 “*realised that co-usage can help you to have more runners which lets you increase the amount of jobs you are running*”. R_4 uses self-hosted CI/CD tools side-by-side with cloud-based solutions to counter the time limitations imposed by the free tiers of CI/CD tools: “*we attach our own resources and we do it via GitLab CI/CD because the time limit is greater than what GitHub Actions allows*”. Similarly, R_{19} reported that “*we added more CI/CD services, so we got more parallelism so that we would complete all jobs sooner. That has been one of the primary reasons why we still use a lot of them because it makes sure that we can run more jobs in parallel until they complete*”. R_{21} co-used Sauce Labs in complement to GitHub Actions because at that time GitHub Actions did not yet provide the ability to use the Windows VM during development for free in open source projects. According to this respondent, “*[our project has to] work in all the browsers and [...] Sauce Labs gives free stuff to open source projects [...] they let me run tests on Windows*”

A last reported reason for co-usage is to *test a CI/CD for potential migration*. R_1 reported introducing TeamCity in complement to GitHub Actions since “*it would allow us to migrate if necessary*”.

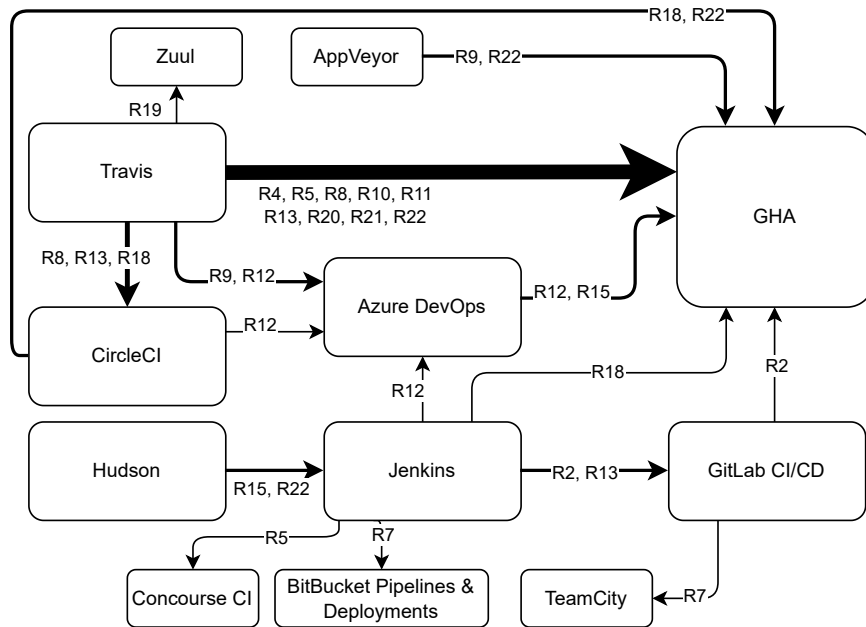


Figure 4.3: Reported completed migrations between CI/CD tools.

Takeaways. It is quite common practice to use several CI/CD tools in parallel. Most combinations involve Travis, AppVeyor, GitHub Actions or CircleCI. The most frequent reasons (RQ 4.2.1) are to cover more operating systems, to access complementary features, and to benefit from more computing resources.

4.4.2 Software Projects Migrating to a Different CI/CD Tool

We observed in Section 4.3.1 that most respondents have used several CI/CD tools through time. We asked them explicitly whether they co-used these different CI/CD tools (see Section 4.4.1), and whether they migrated from one tool to another. Since developers may decide to migrate to another CI/CD tool for different reasons, we also asked the respondents to share their experience on why they migrated (RQ 4.2.2).

Overall, respondents reported a total of 32 completed migrations in different projects (some respondents reported multiple migrations), involving 12 different CI/CD tools. Figure 4.3 shows the migration paths. We observe that

the reported migrations originate from 7 distinct CI/CD tools and lead to 9 distinct CI/CD tools. Most migrations originate from Travis (15 out of 32) and lead to GitHub Actions (17 out of 32). The most frequently observed migration pattern is from Travis to GitHub Actions (9 migrations), corroborating the findings of Golzadeh *et al.* (2021c).

Overall, we observe a general tendency to move from self-hosted CI/CD tools to cloud-based solutions since most of the completed migrations are to cloud-based CIs (*e.g.*, GitHub Actions) and at least half of migrations from Jenkins (a well known self-hosted CI/CD tool) are towards cloud-based CI/CD tools (namely GitHub Actions, Azure DevOps, and Bitbucket Pipelines). Free tier cloud services have the advantage of not needing to configure and maintain a local CI/CD server, which can be quite costly for small teams and projects in terms of personnel and hardware resources. R_9 therefore prefers using the free tier of a cloud-based CI/CD service, rather than spending any budget on that resource: “*since everything we’re doing is open source and most of these CI/CD providers have an offer for open source projects to provide them with free hardware or CPU time we usually don’t spend any money on hardware*”. Moreover, cloud-based CI/CD services are usually more scalable and adaptable to the actual resource needs of the project, as mentioned by R_5 : “*just to make sure, we run on cloud infrastructure, so if we need to scale, we scale.*”

We asked respondents to share the reasons why they migrated. Table 4.8 shows the 10 reported reasons to migrate, as well as the source and target of each migration. Some migrations are reported more than once since there were multiple reasons that drove the decision to migrate.

The most frequent reason that was reported to migrate is to *have less restrictions on the free tier*, mentioned by 8 different respondents for 10 different migration cases. All these cases correspond to migrations away from Travis. As discussed in Section 4.3.5, the change in Travis’ free tier imposes so many restrictions on open source projects that it leads them to migrate to another CI. For example R_{19} mentioned “*I figured the project could use sponsored money or donations to pay for it, but I felt it would be more responsible for our project to not spend that money on Travis, but rather to save the money and just move to another free service instead.*” Half of the migrations away from Travis lead to GitHub Actions (5 out of 10 cases), the remaining ones being to Azure DevOps, CircleCI and Zuul. Another reason to migrate away from Travis is to “*use a more reliable CI/CD tool*”, reported by 7 respondents. This is a consequence of the many reliability issues identified in Section 4.3.5 for Travis. This had led respondents to migrate away to more reliable CI/CD tools such as GitHub Actions (5 reported cases) and CircleCI (2 reported cases).

Table 4.8: Reasons for completed migrations.

migration reason	migration from	migration to	respondent IDs
Having less restrictions on the free tier	Travis	GitHub Actions	10 11 13 21 22
		Azure DevOps	12 19
		CircleCI	8 13
		Zuul	19
Using a more reliable CI/CD tool	Travis	GitHub Actions	4 8 13 20 21
		CircleCI	13 18
Better integration with hosting platform	AppVeyor	GitHub Actions	9 22
	CircleCI	GitHub Actions	18 22
	CircleCI	Azure DevOps	12
	Azure DevOps	GitHub Actions	12
	Jenkins	GitHub Actions	18
	Travis	GitHub Actions	22
Better support of multiple platforms	AppVeyor	GitHub Actions	9 22
	CircleCI	GitHub Actions	22
	Jenkins	GitHub Actions	18
	Azure DevOps	GitHub Actions	12
	CircleCI	Azure DevOps	12
Decreasing the amount of CI/CD tool co-usage	CircleCI + Jenkins	GitHub Actions	18
	Travis + AppVeyor	Azure DevOps	9
Having better features	Azure DevOps	GitHub Actions	15
	Jenkins	Concourse CI	5
Moving to a successor CI	Hudson	Jenkins	15 22
Making the project open source	GitLab CI/CD	GitHub Actions	2
Moving to a new ecosystem	GitLab CI/CD	TeamCity	7
Reducing the maintenance burden	Jenkins	Azure DevOps	12

The second most frequent reason to migrate to another CI/CD tool is to obtain a “*better integration with hosting platform*” (8 cases). This reason refers to the integration of CI/CD tools within the social coding platform used by the respondent, typically GitHub, GitLab, Azure or BitBucket. The target of these reported migrations is almost exclusively GitHub Actions, likely due to its deep integration within GitHub, the most popular hosting platform.

The need for “*better support of multiple platforms*” also explains several migrations. For 5 out of 6 cases, GitHub Actions was the target of choice, as it supports the most common operating systems such as Linux, Windows, and macOS. One respondent (R_{12}) actually migrated from CircleCI to Azure DevOps with the aim of a better integration with GitHub and a better support for Windows, but this was before GitHub Actions existed. They migrated from Azure DevOps to GitHub Actions a bit later.

Another reported reason to carry out migrations relates to the CI/CD co-usage that we analysed as part of Section 4.4.1, namely to “*decrease the amount of CI/CD tool co-usage*”. While the results of Section 4.4.1 highlighted the need to co-use multiple CI/CD tools for specific reasons, it comes at a certain cost and increased effort.

“*Co-usage introduces at least two difficulties. You need to maintain both, they have sometimes different syntax in the YAML files, so you have to have more knowledge so it’s more work, that’s sort of the first issue. The second issue I see is for example for code coverage. If you do code coverage on both platforms and you want to merge your code coverage, it might be difficult, [...] whereas when you have only one CI/CD provider, it’s much easier because there is only one workflow*” [R₁₈]. Therefore, project maintainers keep track of the evolving functionalities of the available CI/CD tools in order to seize the opportunity to reduce the maintenance overhead caused by using multiple CI/CD tools. For example, R₁₈ replaced a combination of CircleCI and Jenkins by GitHub Actions, while R₉ replaced Travis and AppVeyor by Azure DevOps since they wanted “*to unify our pipelines and have everything in one place*”. In both cases, the new CI/CD tool supported all the needs that were previously covered by the two CI/CD tools being co-used.

“*Having better features*” was reported as a migration reason twice. R₅ reported moving from Jenkins to Concourse CI because “*I really want a good visualization from the whole flow. Concourse gives me that, and not only per single repository. That’s the critique I have to most tools. Pipelines are linked to single git repos. Concourse not. Pipelines are completely independent from your all of your code basis*”. R₁₅ reported a migration from Azure DevOps to GitHub Actions because “*[they] knew some news that Microsoft was going to invest on GitHub Actions and not investing lots of effort on Azure DevOps. The features that GitHub Actions provides for writing and customizing plugins [...] encouraged our team to decide to have a migration.*”

Other noteworthy reported migration reasons were:

- “*To move to the successor CI/CD tool*”. This reason was mentioned by two respondents (R₁₅ and R₂₂) that migrated from Hudson to its next generation open source successor Jenkins.
- “*Making the project open source*” was the reason that caused R₂ to migrate from GitLab CI/CD to GitHub Actions: “*we wanted to have something that we can show the open source code for the DevOPS pipeline. Since GitHub provides a free runner, and the open source code of the application is on GitHub, we went there.*”

- R_7 reported “*moving to new ecosystem*” as the reason to migrate from GitLab CI/CD to TeamCity since they wanted to make use of the full JetBrains tool suite.
- R_{12} reported “*reducing the maintenance burden*” as the main reason to migrate from Jenkins to Azure DevOps: “*So what drove a migration from Jenkins to Azure DevOps was the maintenance burden of Jenkins. I think we almost had one person full time, just maintaining Jenkins.*”

Takeaways. Respondents are constantly looking for more appropriate CI/CD tools. Most of the reported migrations are away from Travis and towards GitHub Actions, a consequence of Travis’ change in free tier and reliability issues. Migrations towards GitHub Actions are primarily (RQ 4.2.2) due to its generous free tier, its deep integration with GitHub, and its support for the most common operating systems.

4.4.3 Difficulties in Carrying Out a CI/CD Migration

In Section 4.4.2 we observed that many respondents migrated from one CI/CD tool to another one. Because the different CI/CD tools have different philosophies, approaches and configuration files, it might be difficult to migrate to another CI/CD tool. We therefore explicitly asked the respondents to report on their experience (RQ 4.2.3).

Many respondents ($R_8, R_{10}, R_{13}, R_{21}$) reported having faced no real problems in moving from one CI/CD tool to another one. For instance, R_2 reported “*I think it was around 3 days. And the reason it was short is because the DevOps pipeline was very simple so we just installed*”. Similarly, R_5 described their migration was not a hard process. Given that the destination CI/CD tool was already being used by their company, so they had a basis on which to build specific pipelines for their project: “*we have to create pipelines for ourselves for our code [...] but there is already quite some investment in a number of standardized pipelines, it’s not really that we have to start from zero. We can start by duplicating a pipeline and adapting here and there for some tooling that we are running.*” [R_5].

The remaining 18 respondents did mention having faced difficulties during CI/CD migration, but the reported reasons were very diverse. One recurrent reason had to do with the *learning curve* to master the syntax of the new CI/CD tool. Many contemporary CI/CD tools use a YAML-based syntax to describe their workflows or pipelines (*e.g.*, GitHub Actions and Travis), while

others may use a totally different way to specify CI/CD pipelines, and the differences in syntax and semantics were reported to cause migration difficulties by several respondents:

R₁₃: “There is no standard way to publish libraries because you want to still reuse pipelines between jobs, between software. In CircleCI it’s called orbs, in GitHub Actions this is the Action libraries, in Jenkins it’s another one, so there is always a learning curve, even if you know the command to type in the CI. You need to learn the CI/CD tool [...] that really takes a lot of time”.

R₉: “Sometimes for example environment variables are differently set in the CI/CD systems or some other minor differences between the providers.”

R₁₈: “Something interesting you might want to look at are the commits of somebody doing a migration. You will see that you do a lot of typos and try to run the CI/CD 20 times until it works once. You copy-paste some examples from the Internet, you adapt it, but you forget to like there’s a lot of details. It’s often YAML files that are really prone to mistakes. So you make [lots of] commits until you get to the result you want to have. And there’s no way to pre-test it on your local machine. So you just commit, push, wait for the build to run, and then look at the results. So that’s why a migration might take some time”.

Solutions are being proposed to reduce this learning curve. For instance, **dagger.io** provides a way to unify workflow specifications across different CI/CD tools, by offering cross-language instrumentation through dedicated APIs. For example, developers may use the Go SDK to develop all of their CI/CD pipelines using the Go programming language, or the CUE SDK to use the CUE configuration language. In both cases, it avoids needing to know and learn the specific YAML (or other) variants being used by CI/CD tools.

R₁₀ reported on the lack of an easy way to ensure the correct execution and behaviour of pipelines within the CI/CD migration target: *“Difficulties when you migrate a CI/CD tool is just the time it takes to verify that it works. Because usually [...] you have to do changes to your repository, and then you have to wait until the CI/CD tools pick up the changes, run the script and tell you back. So the feedback cycle is just slow. Or when I move the automated releases script, semantic release, from Travis to GitHub Actions, I actually *had to* do a release to test, to verify that it works. So that took time.”*

R₁₂ experienced migration difficulties due to *completely different architecture and security models*: *“The worst migration that I’ve done is from Travis*

to Azure DevOps. That was so difficult because they are completely different systems with different security models and different architectures and that took a whole team working for a week to migrate.”

Three respondents actually considered migrating to a new CI/CD but, in the end, decided not to for various reasons. R_6 considered Azure DevOps as a replacement of a proprietary in-house CI/CD tool used in a big commercial company. The “*lack of developer familiarity with the new CI*” and the “*lack of scalability of Azure DevOps*” were the main reasons holding the company back from migrating. Azure DevOps’ capabilities did not match the company’s needs for handling a high number of builds per day and having enough flexibility and scalability. R_{19} explained that their company was using Azure DevOps, GitHub Actions and a custom-built CI/CD tool at the same time. They were considering a definitive migration towards GitHub Actions to reduce the number of co-used CI/CD tools but eventually decided against the migration, in order to keep the benefit of running multiple builds in different CI/CD tools in parallel. Similarly, R_{18} decided not to perform a migration from Azure DevOps to GitHub Actions because the latter missed an important feature for CD: “*we pay the Microsoft service [for Azure DevOps] because it’s a company [and] so you get the support and everything. If we would go to GitHub Actions, we would switch to the professional paid plan [...] but there is something in Azure DevOps which GitHub Actions does not have. It’s the Release Pipelines, which is much more evolved*”.

Takeaways. Respondents reported a wide range of difficulties during migration to a new CI/CD tool (RQ 4.2.3): the learning curve, fundamental differences between the source and target of the migration, the trial-and-error nature of configuring a new CI/CD tool, the lack of familiarity with the new CI/CD tool, and important missing features for continuous delivery and deployment.

4.5. Discussion

This section discusses important additional insights about CI/CD usage that we have been able to gain from the interviews. Some of these insights did not align with specific interview questions, and others emerged as side-remarks that we consider nevertheless important to discuss here, since they provide additional insights into why developers use specific CI/CD tools or why they decide to migrate to other CI/CD tools.

Section 4.5.1 starts by discussing the many aspects surrounding GitHub Actions that have caused it to become one of the dominant CI/CD tools today. Section 4.5.2 explains how the open source nature of projects or the CI/CD tools used by them can play an important role in the CI/CD tool being used. Section 4.5.3 discusses why some CI/CD tools have been subject to restrictions on their free tier. Section 4.5.4 presents some potential future directions for CI/CD tools, as suggested by interview respondents. Finally, Section 4.5.5 reflects on the need to have a sufficiently diverse CI/CD landscape in order to satisfy the varying needs of CI/CD tool users, as well as to reduce the risk of certain CI/CD tools taking a monopoly position.

4.5.1 On The Use of GitHub Actions

We aimed to understand how and why developers have migrated to different CI/CD tools and, in particular, why GitHub Actions has become the dominant CI/CD tool in the current CI/CD landscape for GitHub. The CI/CD tool usage reported in Table 4.3 and the migration cases reported in Figure 4.3 signal the increasing popularity of GitHub Actions. This corroborates the quantitative study by Golzadeh *et al.* (2021c) who observed that only 18 months after its introduction, GitHub Actions has become the dominant CI/CD on GitHub.

Anticipating the popularity of GitHub Actions among respondents, the interview questionnaire included a question about valuable features of GitHub Actions that were appreciated by respondents, and that caused some of them to migrate to GitHub Actions as their CI/CD of choice. Respondents mentioned a variety of reasons for doing this migration: the excellent integration of GitHub Actions into GitHub; the fact that it provides a generous free tier for open source projects; its support of a wide range of operating systems and hardware architectures; the availability of a large marketplace of reusable Actions; and the availability of better features than some competing CI/CD tools. Each of these valuable features have contributed to GitHub Actions’s popularity. Another driver for this popularity was the increasing dissatisfaction with Travis (as reported in Section 4.3.5).

Among many other reasons, the company behind Travis failed to correctly communicate about important security issues. Some of those issues can be very dangerous and impactful, such as the exposure of customer-specific secret environment data such as signing keys, access credentials, and API tokens for a duration of 8 days in 2021.¹⁰ Using GitHub Actions instead of Travis is of course no guarantee that security concerns will not arise. For instance, in

¹⁰<https://nvd.nist.gov/vuln/detail/CVE-2021-41077>

Section 4.3.5 we reported some potential problems and examples of important security issues related to GitHub Actions as well.

We also conjectured that the acquisition of GitHub by Microsoft in June 2018 may have played a role in GitHub Actions popularity. Given that GitHub is the most popular hosting platform for OSS projects, its acquisition by a big tech company is likely to have at least some impact on the use of its integrated CI/CD service GitHub Actions that was publicly released in November 2019. We asked the interview respondents whether the acquisition by Microsoft was perceived as positive or negative. While 14 out of 22 respondents answered that GitHub’s acquisition did not have any impact on their CI/CD choice, 8 respondents did say that it somehow played a role, raising both arguments in favour or against this acquisition. On the negative side, some respondents raised concerns against the acquisition. For instance, *R₁₅* prefers using GitLab for his personal projects because of that: *“Personally, I’m using GitLab for my own project. I don’t like the Microsoft concept and owning the company or something like that.”* On the positive side, Microsoft’s attitude toward OSS has improved recently. *R₂₂* reported that *“These last years, Microsoft is really doing huge changes internally to make their reputation change about open source. I think Microsoft is changing its point of view on open source and I think it’s for the greater good of open-source developers”*. In a similar vein, *R₅* reported *“For me there are two versions of Microsoft, before and after Satya Nadella became the CEO. With Satya Nadella as CEO that’s for me, the V2 of Microsoft as it became much more open source friendly. [...] So when I heard the news that Microsoft bought GitHub I wasn’t really too afraid. I would have been more concerned of that acquisition if it would still have been the V1 Microsoft with Steve Ballmer and all that.”* Another positive aspect of the acquisition is that it has enabled GitHub Actions to grow rapidly in functionality and performance. *R₁₁* appreciated the fact that *“every month GitHub is releasing something new: the code Explorer, GitHub Actions, [...] So at least the switch to Microsoft was good to get a bit more into the business.”* *R₁₃* confirms this: *“Now we actually have GitHub Actions. We have a lot more performance things in GitHub that we had in the past, so for me that change was kind of fine.”* This vision is shared by *R₉*: *“I think the impact of Microsoft buying GitHub so far has been pretty positive, and it has only made GitHub more useful.”*

Next to GitHub Actions, Microsoft is offering Azure DevOps as a competing CI/CD product that is part of the Azure cloud-based ecosystem, and many respondents reported having used it. One could wonder how sustainable it is for a company to continue supporting two competing CI/CD solutions with

similar functionalities. We observed two cases of respondents having migrated from Azure DevOps to GitHub Actions (see Figure 4.3). Respondents that used Azure DevOps valued its tight integration in the Azure ecosystem and the technical support offered by Microsoft. For example, R_{14} reported: *“In my current company we are using whatever tool Microsoft is providing. One reason is that we are using Azure Portal, Azure DevOps, Azure anything. So we consider that it’s better to build our pipeline using Azure [...] In Azure, there are more plugins and more options for using it in the Microsoft world. Because we are using Azure DevOps, all the repositories are in the same place as pipelines and also all the Scrum boards are in there”*.

4.5.2 Open Source Nature of CI/CD Tools

The open source nature of their software project and/or their CI/CD tool was reported by multiple respondents as playing an important role in the choice of CI/CD tool. Some OSS projects specifically select or impose the use of open source CI/CD tools, as it matches the nature and mindset of their open source policy. For instance, R_3 mentioned that *“for some practical reason, but also maybe ideological, we like to use open source solutions and have to control on our software that we use.”* Similarly, respondent R_{10} argued *“Philosophically, we don’t like the company that bought Travis. They just have different values then we have in our open source projects.”*

OSS projects may also select specific CI/CD tools for more pragmatic reasons. For example, R_2 chose GitHub as a platform to demonstrate their open source code, not because GitHub itself is open source (it is not), but because it is the most popular platform for hosting OSS projects: *“The reason is that we wanted to have something that we can demonstrate, like show the open source code for the DevOps pipeline. And since GitHub provides a free runner, and the open source code of the application is on GitHub we went there.”*

R_5 also argued that the choice of a CI/CD tool depends on whether the project using it is open source or commercial: *“The requirements for an open source project are usually quite different from a commercial project. [...] If it is] only a CI/CD for an open source project and you need to build a package that you want to have published in registries, then for example GitHub CI/CD would be completely satisfying. Because GitHub Actions gives me all the building blocks which I need at that moment. But if I want to build a product with all the testing, all the quality gates validation in a number of environments before going to have a real live environment then I would still search for something which gives me the full flow and have more ability to model that full flow.”*

The interviews revealed that 8 respondents involved in OSS projects pre-

ferred using the free tier solution of a commercial CI/CD tool (mostly GitHub Actions, Travis and GitLab CI/CD) since OSS projects often have very limited financial resources to develop and maintain their software. R_{19} explained the decision-making process for choosing another CI/CD tool due to restrictions imposed on the free tier of Travis: *“I had the choice to either pay for it or not remain on Travis. And then I figured out I had a lot of other options. We could use sponsored money or donations to pay for it, but I felt it would be more responsible for our project to not spend that money on Travis, but rather save the money and just move over to another free service instead”*. R_{11} also *“decided to change to go to GitHub Actions that was free”*. This shows that changes in the pricing policy or restrictions imposed on the free tier of the CI/CD tool may incite or even force OSS projects to migrate to other CI/CD tools.

On the other hand, commercial projects tend to prefer using paid, commercial CI/CD tools because they offer a better service-level agreement and technical support in case of reliability problems. R_{14} who used Azure DevOps in a company said *“The problem with Microsoft Azure is money. You need to spend a lot of money, but the tools that you are getting from Microsoft [...] are more powerful than the others. [...] If I had money, I’d migrate [my projects] to Azure.”*

Some respondents were not satisfied with any of the existing open source or commercial CI/CD tools. As a consequence, they rely on custom-built CI/CD tools in their companies. These CIs were created to support the specific needs of the company. For example, R_6 mentions that their company *“built its own proprietary CI/CD pipeline which now operates to deploy thousands of deployments per day. It was loosely based on Jenkins for a while.”* More specifically, *“it started as a kind of deployment of Jenkins that evolved over time into an actually proprietary system that’s built from scratch”*.

One aspect in favour of open source CI/CD tools would be the ability to contribute changes back to the OSS project. However, R_6 argued that it not always easy to do so, due to the community’s latency of accepting changes: *“Contributing back to the community is not always possible because volunteer-based projects face the challenge that you have of contributing back to popular open source projects. Not everybody’s contributions are going to be able to be accepted. So I think a latency is introduced by this.”*

4.5.3 Restrictions on the Free Tier of CI/CD Tools

Among the main shortcomings that were identified in Section 4.3.5, the restrictions imposed on the free tier of CI/CD tools were frequently reported by the

respondents. These restrictions cause many projects to select an alternative CI/CD tool offering more computation resources or more build time. Travis was frequently reported by the respondents as being too restrictive on its free tier. Its decision to impose more restrictions on its free tier in 2020 was even one of the main reported reasons for OSS projects to migrate to another CI. We investigated the rationale behind imposing such restrictions, and we found that the decision was mostly driven by abusive cryptominers.

Li *et al.* (2022) studied the phenomenon of *CI-jacking* which, in other words, correspond to the abuse of CI/CD tool resources for mining cryptocurrencies. Through an empirical analysis of GitHub repositories and log files on CI/CD platforms, they found 1,974 instances of CI-jacking, with an estimated revenue of over \$20,000 per month using the computational resources of the CI/CD tools' free tier. This abuse has led CI/CD providers to impose stronger limitations on their free tiers.¹¹

For example, Travis motivated its decision to change its pricing model as follows (Mendy *et al.*, 2020): “[...] *we have encountered significant abuse [...] (increased activity of cryptocurrency miners, TOR nodes operators etc.). Abusers have been tying up our build queues and causing performance reductions for everyone.*” Similarly, in February 2021, the Director of Product Management of GitHub publicly announced strong restrictions on the free tier of Azure DevOps due to “*a high percentage of new public projects in Azure DevOps being used for crypto mining and other activities we classify as abusive*” (Machiraju, Vijay, 2021).

Two interview respondents confirmed this abuse by cryptominers, and the harm it is causing to OSS projects whose functioning often depends on the ability to benefit from the resources offered by the free tiers of cloud-based CI/CD tools: “*in recent years people have been abusing a CI/CD solution for mining bitcoins. This is annoying for the open source community because we rely on those tools. Those tools are really critical for the open source communities to continue to build and secure the toolchain*” [R₁₃]. In addition to this, R₂₁ explained that cryptominers not only affect computational resources, but also impact human resources that need to check for the presence of cryptominers: “*a human has to review the code [of new contributions] and make sure that someone is not trying to install a cryptocurrency miner on our Jenkins installation.*”

More recently, the integration of artificial intelligence and machine-learning-based tasks into development workflows has introduced new forms of computational pressure on CI/CD infrastructures. Unlike traditional build

¹¹<https://webapp.io/blog/crypto-miners-are-killing-free-ci/>

and test jobs, AI-related workloads often require sustained computation, large memory footprints, and in some cases, specialised hardware such as GPUs (Amershi *et al.*, 2019). Recent work shows that AI-augmented CI/CD pipelines increasingly embed predictive analytics, anomaly detection, and intelligent test generation directly into automation workflows, improving reliability and release frequency but also increasing resource demands (Allam, 2025; Vemuri *et al.*, 2024). As these pipelines evolve into continuous learning systems that rely on persistent data processing and model inference, they challenge the cost and scalability assumptions of cloud-hosted and free-tier CI/CD environments (Zampetti *et al.*, 2020). This trend mirrors earlier tensions observed with cryptomining abuse, highlighting the ongoing difficulty of balancing open automation infrastructures with finite computational resources.

4.5.4 Future of CI/CD Tools

Since their inception, CI/CD tools have come a long way, continuously adding new automation facilities to support an increasing range of software development activities. It is beyond doubt that CI/CD tools are widely adopted and play an important role in both OSS and commercial software development (Golzadeh *et al.*, 2021c; Soares *et al.*, 2022).

As part of the open-ended closing question of each interview, respondents were asked to share important remarks related to CI/CD tools. Some respondents used this opportunity to share their opinion on the expected future of CI/CD tools and how these tools will become integrated with other software development components.

*R*₁₀ expressed the idea of having a whole *physically independent* infrastructure that can use the full existing features of a social coding platform, including a software development environment, version control system, issue tracking system, online coding environment, and a GitHub Actions-like CI: “[companies and developers] want someone like GitHub to host a version of GitHub for them. That means that you would get your own thing hosted on GitHub’s own hardware, but it will be physically separated but still integrated with the github.com platform. The main benefit of that is that you don’t need to think about your custom actions runner and things like Codespaces.¹² Development in the cloud will become very relevant in future. And when you want to keep self-hosting, it’s not only about hosting the git platform. It will be more and more about also hosting all these other things like the CI/CD environment, which is GitHub Actions for GitHub, and the cloud development

¹²<https://visualstudio.microsoft.com/services/github-codespaces/>

platform, which is Codespaces. It's going to get harder and harder to self-host [while] a service hosted version will become much more attractive." Other companies, such as `gitpod.io` have also started providing similar cloud development environments, that can be integrated with one's preferred social coding platform (e.g., GitHub, GitLab or BitBucket).

R_6 suggests considering CI/CD in the full software development lifecycle: "CI/CD is part of a broader system of continuous delivery of software. Looking at it in isolation is like looking at just a portion of a full pipeline. Delivering software begins at this ideation phase and ends when a user interacts with it. And CI/CD has this role to play, but it is not the entire spectrum. So CI/CD needs to be looked at in the context of the rest of the engineering system being used to deliver software in a particular place, whether it be open source software or in different proprietary setups." R_5 shared this point of view: "If I want to build a product with all the testing, all the quality gates validation in a number of environments before going to have a real live environment, then I would still search for something which gives me the full flow [...]".

R_9 also considered there is room for improvement, notably to address the amount of data that needs to be downloaded each time a CI/CD process is executed: "there would be automated ways to reduce the amount of data that's downloaded from the providers. Some sort of automatic caching. And the other thing is to also have some automatic caching of compiler outputs. But both things are probably not so easy to do in a generic fashion". Indeed, CI/CD tools are known to incur a high cost, because of the computational resources they require, combined with the frequency of running builds. Minimizing execution time of CI/CD workflows is crucial, as it enables timely feedback to developers, avoiding them to switch to other tasks while waiting for the CI/CD workflows to complete, which is known to be a costly operation for knowledge workers (Zheng *et al.*, 2025).

Recent software engineering research has proposed several techniques that could further improve the efficiency of modern CI/CD systems. Much of this work focuses on reducing build time and cost by avoiding redundant computation through selective execution strategies. Studies on build- and test-skipping based on change impact analysis, dependency modeling, and historical build outcomes show that significant performance gains can be achieved without substantially degrading failure detection (Jin & Servant, 2021, 2022).

Machine learning based approaches have also been increasingly explored to predict whether commits, tests, or pipeline stages are likely to affect build outcomes, enabling CI systems to skip executions with low risk of failure (Abdalkareem *et al.*, 2020). Empirical evaluations on large CI datasets indicate

that these techniques can substantially reduce CI workload while preserving early failure discovery.

In addition, recent work has investigated cache-aware and environment-aware optimizations, such as reusing build artifacts and dependency caches across executions to mitigate the overhead of ephemeral CI environments (Gallaba *et al.*, 2022a). Together, these advances highlight opportunities for repository-native platforms such as GitHub Actions to better integrate research-driven CI optimization techniques.

4.5.5 On The Diversity of The CI/CD Landscape

Section 4.3.1 revealed a wide diversity of CI/CD solutions having been used by respondents. Section 4.3.4 further revealed that, even if the most popular CI/CD solutions covered most of the desirable features, there were still valid reasons for using less popular CI/CD tools because they were offering specific valuable features that could not be found in the more popular CI/CD tools. This was confirmed in Section 4.4.2 where respondents identified many reasons for co-using CI/CD tools, such as the need to support specific hardware platforms or operating systems, access to specific features, and the ability to use more computing resources by running multiple CI/CD tools in parallel. This shows the importance of maintaining a wide diversity of CI/CD tools, each having their own set of features, advantages and shortcomings. *R*₅ even goes one step further by claiming that there still is plenty of room for new contenders in the CI/CD landscape: “*a lot of people think that tools regarding CI/CD is already a well-equipped market. But personally I think there is still a lot of room for improvement. If there would be a contender really thinking out of the box [...] I think he would still make a fair chance of getting a decent market share*”.

Nevertheless, in response to Section 4.4.3 as well as in Table 4.3 we observed a general tendency to migrate towards the more popular cloud-based CI/CD tools (such as GitHub Actions, Azure DevOps and GitLab CI/CD). The ever stronger integration of these CI/CD tools in their social coding platform, compounded by the fact that workflows and pipelines are increasingly relying on reusable building blocks (such as Actions, Orbs and plugins) makes it more difficult to migrate away from them. This leads to an increased risk of vendor lock-in, that may lead to a monopoly position of some CI/CD providers, ultimately resulting in a lack of innovation due to absence of competition.

GitHub, the *de facto* solution for OSS projects nowadays, is a good example of potential vendor lock-in by a private company owning the dominant platform for distributing OSS. *R*₉ was concerned about this risk: “*The only kind of*

concerns, obviously, to have the vendor lock-in and kind of monopoly situation.” R_{11} shared this viewpoint: “From an ethical point of view it’s a pity that GitHub and Microsoft joined together.” At some point in the future, Microsoft might change its strategy to try to profit from its monopoly: “There are intangible benefits that Microsoft gets and we’ll see if changes happen in the next few years, to where GitHub makes changes to be more profitable and that don’t necessarily serve the free software folks. [...] I have mixed feelings about it, on the one hand, it really is convenient having everything integrated at one place. On the other hand, how much do we really want to invest all of open source in a single company?” [R_{21}]

Nevertheless, it seems like respondents are aware of this risk and still willing to use GitHub, while keeping their options open to move to other CI/CD alternatives: “so far I think, for us, it’s been a positive experience. But we are aware of these dangers and we would be ready to move to another platform if we have to” [R_9]. Similarly, R_1 reported to “have alternatives in case there is something that changes in the GitHub CI/CD user conditions. For instance, If GitHub Actions becomes irrelevant or not practical given the conditions of the project, then we know that there is a simple way to just use TeamCity instead of GitHub Actions.”

4.5.6 Platform Choices, AI Integration, and Potential Migration Effects

Collaborative development platforms evolve through strategic design choices that can influence long-term adoption, user satisfaction, and ecosystem stability. A notable recent shift within the GitHub ecosystem is the increasing integration of generative AI tools, particularly GitHub Copilot, across the platform’s user interface and development workflows. Empirical studies indicate that AI-assisted programming tools can affect developer productivity and participation, while also introducing changes to coordination dynamics and integration effort within projects (Amershi *et al.*, 2019; Song *et al.*, 2024). These changes suggest that AI integration does not merely augment development tasks but reshapes how developers interact with the platform.

At the same time, anecdotal and industry reporting point to growing friction around the perceived intrusiveness of AI features, including concerns about automatic interventions and limited control over disabling them. While systematic academic evidence of large-scale migration driven by AI integration is still limited, such reactions echo earlier moments in software ecosystem history where platform decisions influenced community trust and engagement. Historical precedents, such as the decline of SourceForge and Google Code,

illustrate that shifts in platform dominance can occur unexpectedly as a result of strategic or organizational changes.

Emerging alternatives such as Forgejo and the continued presence of GitLab demonstrate that viable non-GitHub ecosystems exist, each offering different perspectives on automation, governance, and user control. Recent empirical work confirms that projects increasingly operate across multiple platforms and CI/CD services, and that migration or diversification is a realistic option rather than an exception (Zampetti *et al.*, 2021). Because CI/CD solutions like GitHub Actions are tightly coupled to their host platforms, any sustained migration away from GitHub would necessarily imply a corresponding shift away from GitHub Actions.

Although it remains difficult to predict which platforms will dominate in the future, these dynamics highlight an important point: some workflow automation tools such as GitHub Actions or GitLab CI/CD cannot be studied in isolation from the platforms that host them. Platform-level decisions, particularly around automation and AI, have the potential to shape not only developer experience but also long-term adoption patterns of CI/CD ecosystems.

4.6. Threats to Validity

Here we discuss the threats to the validity of our work.

Internal validity relates to whether an experimental treatment or condition makes a difference or not Ampatzoglou *et al.*, 2019. Given that our analysis is based on subjective interviews, the main threat pertains to which questions we asked, how, and in which order. A different set of questions, different order, or even different phrasings could have led to different responses. We reduced this threat by carefully verifying the interview protocol, and carrying out pilots on three different persons, before actually starting the study. Moreover, the open-ended nature of the interviews provided ample opportunities for respondents to provide additional contextual information that was not necessarily directly related to the questions being asked.

External validity is concerned with the generalisability of the approach and the representativeness of the results Ampatzoglou *et al.*, 2019. We strove to have a good balance in respondent profiles covering both open source developers and industrial practitioners. While we strove to have a balanced selection of respondents in terms of geographical distribution, we acknowledge that Eastern Europe as well as the Australasian and African regions are underrepresented in our population. The inclusion criteria that have been used for selecting interview candidates also introduced a deliberate selection bias towards developers

with proven practical experience with CI. As a result we cannot claim that the results generalise to less experienced developers.

Construct validity concerns the relation between the theory behind the experiment and the observed findings Ralph and Tempero, 2018. Given the qualitative nature of our study, the main threat pertains to how we have interpreted the responses obtained from the interviews. To mitigate this risk, we have relied on the well-established process of deductive and inductive coding, which involved multiple researchers in order to further lower the risk of making incorrect interpretations.

Conclusion validity deals with the degree to which reasonable conclusions have been reached from the collected data Maxwell, 1992. One might argue that having more respondents could have increased the support of our findings. Since we continued interviewing respondents until we reached saturation in the responses, we believe that the conclusions made are reasonable, especially given the qualitative nature of this paper, as we do not aim to show any statistical significance of our observations. Moreover, some of the qualitative findings have been triangulated with quantitative results reported in earlier work.

Geopolitical reasons may have implicitly affected some of the received responses. For example, European respondents are subject to other privacy regulations (GDPR) than non-European ones, which could have influenced their preference toward CI tools maintained in Europe. As another example, the acquisition of GitHub by Microsoft, an American company, could have influenced some respondents in favour or against the use of GitHub Actions as a CI tool. Also, two respondents reported not having been able to use some popular commercial CI tools since a ban was imposed on certain countries (such as Iran). This is not a problem per se, since the conclusions drawn from the interviews are actually supposed to reflect and capture this diversity in decisions.

4.7. Summary and Conclusions

This chapter presented a qualitative investigation into the usage patterns, motivations, and migration behaviors surrounding CI/CD tools in modern software development. The analysis is grounded in semi-structured interviews with 22 experienced software practitioners who collectively reported using 31 different CI/CD tools, 14 of which were used by at least two respondents. The participants were drawn from both open source and commercial contexts and had proven expertise with CI systems.

Our findings reveal a diverse and rapidly evolving CI/CD landscape. Developers reported using CI/CD tools to enhance reliability, productivity, security, and development speed, while also aiming to reduce cost and manual effort. The most commonly automated tasks included build and test automation, security and quality analysis, dependency management, release engineering, and automated deployment.

One clear trend was the increasing adoption of cloud-based, platform-integrated CI/CD tools, particularly those embedded within developer ecosystems. Among these, GitHub Actions emerged as a dominant player, frequently mentioned both as a migration target and as an entry point for teams adopting CI/CD for the first time. Developers cited its tight integration with GitHub, support for reusable Actions, multi-platform build capabilities, and a generous free tier as decisive advantages.

In contrast, Travis was a common migration source, with users citing reasons such as declining reliability, feature stagnation, and restrictions on the free tier. Migration was not always smooth; practitioners reported a significant learning curve due to differences in configuration models and execution semantics between tools.

Notably, CI/CD tool co-usage was common. Several respondents reported employing multiple CI/CD systems in parallel to support different architectures or operating systems, to combine strengths of specific tools, or to work around limitations in resource quotas.

Taken together, these findings directly address Goal 1 by clarifying the underlying factors that shape how CI/CD tools, particularly GitHub Actions, are selected, combined, and replaced in software projects. The results show that CI/CD tool choices are influenced not only by technical capabilities, but also by ecosystem integration, cost structures, migration effort, and support for reuse and scalability. By uncovering these decision drivers and trade-offs, this chapter provides empirically grounded insights that can support developers, teams, and organizations in making informed decisions about workflow automation in collaborative development settings.

This qualitative study complements prior quantitative research like the one of Golzadeh *et al.* (2021c) by providing deeper insights into the motivations and challenges faced by practitioners. It also highlights new research opportunities, such as exploring the long-term maintainability of reusable workflow components (e.g., GitHub Actions, Jenkins plugins, CircleCI orbs), mitigating risks related to vendor lock-in, and developing tools to improve CI/CD performance and cross-tool migration.

GitHub Actions Usage

“Success is walking from failure to failure with no loss of enthusiasm.”

Winston Churchill

This chapter shifts focus to how GitHub Actions is actually used in practice across software development repositories on GitHub. Specifically, we investigate the adoption of GitHub Actions at scale to understand which repositories make use of it, what types of workflows are being automated, what are the most common jobs within those workflows, and the broader automation practices that developers follow. We also analyze patterns of Action reuse, how versioning is handled, and the degree to which community-maintained or custom Actions are integrated into development pipelines. In line with the second thesis goal, this chapter aims to characterize the real-world usage of GitHub Actions by uncovering recurring workflow patterns, usage conventions, and integration strategies employed by developers. By identifying these common structures and practices, the chapter provides a grounded understanding of how automation is embedded into everyday development work. This lays the empirical foundation needed to study workflow evolution and maintenance in subsequent chapters.

5.1. Introduction

The emerging GitHub Actions ecosystem is worthy of being empirically studied in its own right, since it is likely to suffer from the issues related to dependency management, security vulnerabilities, outdated or obsolete components, backward compatibility, and so on. This chapter therefore quantitatively studies the use of GitHub Actions in a large number of repositories on GitHub. We analyse which workflows are automated and identify the most frequent automation practices. We show that reuse of Actions is a common practice and identify which Actions are reused and how. As such, we provide an overview of the use of GitHub Actions workflows, a necessary first step towards a better understanding of the emerging GitHub Actions ecosystem and its implications on software development in GitHub repositories. More concretely, we answer the following research questions:

RQ 5.1 *What are the characteristics of repositories using workflows?*

RQ 5.2 *Which kinds of workflows are automated?*

RQ 5.3 *What are the most frequent jobs in workflow configurations?*

RQ 5.4 *What are the automation practices?*

RQ 5.5 *Which Actions are reused and how?*

RQ 5.6 *Which versioning practices are being used?*

5.2. Data Extraction

To conduct an empirical study on the use of GitHub Actions in software development repositories, we need a large collection of GitHub repositories. The dataset should exclude repositories that are used only for experimental or personal reasons, or that show no or little traces of actual software development activity (Kalliamvakou *et al.*, 2014). We relied on the SEART GitHub search engine (Dabic *et al.*, 2021) to obtain a list of candidate repositories. We queried the tool on 2022-01-24 to get all non-fork repositories that were created before 2021, which were still active in 2021, and had at least 100 commits and 100 stars. We obtained 69,147 repositories satisfying these criteria.

On 2022-01-24 we locally cloned these repositories to look for the presence of YAML files in the `.github/workflows` folder of their default branch. We parsed these files to check whether they define a GitHub Actions workflow,

Table 5.1: Number and proportion of repositories using GitHub Actions workflows, grouped by main programming language.

language	repositories		using GitHub Actions workflows	
	#	%	% language	% repository
JavaScript	13,542	19.6%	34.9%	15.9%
Python	12,319	17.8%	45.9%	19.0%
TypeScript	6,362	9.2%	58.5%	12.5%
Java	6,105	8.8%	39.2%	8.0%
C++	5,701	8.2%	40.9%	7.8%
Go	4,988	7.2%	57.2%	9.6%
C	4,314	6.2%	36.1%	5.2%
PHP	4,005	5.8%	48.2%	6.5%
C#	3,630	5.3%	34.6%	4.2%
Ruby	2,599	3.8%	50.8%	4.4%
Shell	2,327	3.4%	33.2%	2.6%
Swift	1,411	2.4%	34.4%	1.6%
Kotlin	1,150	1.7%	56.9%	2.2%
<i>other</i>	<i>694</i>	<i>1.0%</i>	<i>17.7%</i>	<i>0.5%</i>
total	69,147	100%	–	100%

and if applicable, we extracted the relevant data about the workflow (*e.g.*, name, events), about the jobs configured in the workflow (*e.g.*, name, `uses:` key, steps) and about the steps defined in these jobs (*e.g.*, name, commands, `uses:` key). At the end of this process, the dataset covered 69,147 repositories containing 70,278 workflows, 108,500 jobs and 576,352 steps. The remainder of this chapter presents what we found in these repositories, workflows, jobs, and steps (including their Actions). The data and code to replicate the analysis are available on Zenodo.¹

5.3. Characteristics of GitHub Repositories Using Workflows

Not all GitHub repositories make use of GitHub Actions. Out of the 69,147 repositories contained in the dataset, 29,778 of them (*i.e.*, 43.9%) contain at least one workflow file. Table 5.1 reports on the number and proportion of repositories having a workflow file, distinguishing repositories based

¹<https://doi.org/10.5281/zenodo.6634682>

Table 5.2: Comparison of characteristics for GitHub repositories with and without GitHub Actions workflows.

characteristic	median		effect size	
	with	without	Cliff's δ	interpretation
pull requests	124	41	0.384	<i>medium</i>
contributors	20	11	0.277	<i>small</i>
commits	598	344	0.229	<i>small</i>
issues	105	59	0.227	<i>small</i>
branches	5	4	0.139	<i>negligible</i>
age (months)	71	77	-0.082	<i>negligible</i>
stars	398	334	0.078	<i>negligible</i>
size (MB)	5,878	5,099	0.025	<i>negligible</i>
forks	84	80	0.018	<i>negligible</i>
watchers	24	25	-0.013	<i>negligible</i>

on their main programming language (as specified in the repository metadata on GitHub). For each language, *% language* indicates the proportion of repositories within that language repositories that have adopted GitHub Actions workflows, while *% repository* indicates that language's share of all repositories using workflows in the dataset. For instance, a language with a high *% language* but a low *% repository* is one where GitHub Actions adoption is relatively common within its set of repositories, but whose overall contribution to the GitHub Actions ecosystem remains limited due to the smaller size of those repositories in the dataset.

We observe from the fourth column that the proportion of repositories with GitHub Actions workflows varies from one language to another. It ranges from 33.2% (for Shell) to 58.5% (for TypeScript). For recent languages such as TypeScript, Go, Kotlin and Ruby, the majority of repositories are using GitHub Actions workflows. The top three languages (JavaScript, Python and TypeScript) together account for nearly half of all repositories in the dataset (46.6%, third column) and nearly half of the repositories defining a workflow (47.4%, last column).

Since workflows help developers to automate some of the repetitive tasks that are inherently part of the software development process, we expect larger repositories (*e.g.*, those having more contributors or pull requests) to rely more frequently on workflows. We compared the distributions of several characteristics between GitHub repositories with and without workflows: size in MB, number of pull requests, commits, issues, contributors, branches, stars, forks

and watchers. We also considered the age of the repositories in months. A statistically significant difference was consistently confirmed for all characteristics by applying Mann-Whitney-U tests (Mann & Whitney, 1947), using a global $\alpha = 0.01$ significance level across all 10 hypotheses tests after controlling for the family-wise error rate with the Bonferroni-Holm correction (Holm, 1979).

Table 5.2 reports the median value for each considered characteristic, as well as the effect size of the observed difference using Cliff's δ (Cliff, 1993) and its interpretation (Romano *et al.*, 2006). We observe with *medium* or *small* effect size that repositories with workflows exhibit a higher number of pull requests, contributors, commits and issues. We also observe, though with *negligible* effect size, that repositories with workflows tend to have more branches, stars, forks and a larger size. Repositories with workflows also tend to be younger and have fewer watchers.

Takeaways. Looking at the characteristics of repositories using GitHub Actions workflows (RQ5.1), one can observe that more than 4 out of 10 GitHub repositories use workflows. Projects mostly written in JavaScript, Python or TypeScript account for nearly half of the repositories and one third of the repositories using workflows.

5.4. Kinds of Workflows Being Automated

We found a total of 70,278 workflows in 29,778 repositories, hence an average of 2.4 workflows per repository. Nearly half of these repositories (49.3%) defined a single workflow. The remaining 50.7% repositories define two (22.6%), three (11.7%) or four workflows (6.3%), even if we found dozens of repositories with 20 or more workflows.

A workflow can be triggered by one or more events. The events that trigger a workflow can be chosen from a large list of events corresponding to the different ongoing activities within a repository (*e.g.*, commits pushed, pull requests created or updated, comments created). GitHub Actions also proposes special events such as `schedule` to execute workflows on a regular basis or `workflow_dispatch` and `repository_dispatch` to manually trigger a workflow.²

Table 5.3 reports the 10 most frequent events occurring in workflows and the proportion of workflows using them. Since repositories can have more than

²The complete list of supported events can be found on <https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows>.

one workflow, we also report on the proportion of repositories having one of their workflows triggered by these events.

Table 5.3: Top 10 event types in GitHub workflows and repositories.

event	% workflows	% repositories
push	41.8%	63.4%
pull_request	34.1%	56.3%
workflow_dispatch	8.3%	15.4%
schedule	8.1%	16.1%
release	3.0%	6.2%
pull_request_target	1.3%	2.6%
issues	1.0%	2.0%
repository_dispatch	0.7%	2.0%
issue_comment	0.6%	1.2%
workflow_run	0.4%	0.8%
total	99.3%	–

The most frequent events, `push` and `pull_request`, are used by more than half of the repositories. These events correspond to what is typically monitored by traditional CI/CD services and are tightly related to the technical aspects of software development (*e.g.*, test, build and deploy code). The next most frequent events are `workflow_dispatch` and `schedule`. The former allows to manually trigger a workflow using the GitHub API or user interface, while the latter allows to trigger a workflow at a scheduled time.

The most frequent events that can be associated with more social activities are `issues` and `issue_comments`. These events, used by 2.0% and 1.2% of the repositories respectively, react to the creation of an issue or a comment. They can be used, for example, to welcome newcomers, triage issues or ensuring adherence to the Contributor License Agreement (CLA).

In order to gain some insight into the purpose of these workflows, we analysed the most common workflow names. Although the `name:` key of a workflow is optional, nearly all workflows in our dataset define it (99.1%). We found 23,056 distinct names out of the 69,618 workflows defining one. There are 554 names used by at least 10 repositories, and some of them are frequently used by a large number of repositories, such as `ci` (8.2% of the repositories), `test(s)` (5.1%), `build` (3.9%), `codeql` (3.5%), `release` (2.9%) and `lint` (1.0%). The first name explicitly referring to a “social” purpose is `mark stale issues and pull requests` and is used by 0.5% of the repositories.

Takeaways. Looking at the type of workflows that are being automated (RQ5.2), one can observe that half of the repositories using GitHub Actions define two or more workflows. Workflows are mostly triggered by push and pull request events which are used for development-related purposes.

5.5. Most Frequent Jobs in Workflows

A workflow defines one or more jobs that will be executed in parallel (by default). We found 108,500 jobs in the 70,278 workflows in the dataset, hence an average of 1.5 jobs per workflow. The large majority of workflows define a single job (77.8%) or two jobs (10.5%). However, since a repository can define multiple workflows, the number of jobs per repository is higher (3.6 on average). 65.5% of the repositories have two or more jobs defined in the totality of their workflows, and 21.3% even have 5 or more jobs.

As explained in Section 2.4, a job either defines a list of steps that will be executed to achieve its purpose, or can use (through the `uses:` key) another workflow from any public repository. In the latter case, the jobs defined in the other workflow will be executed. The ability to reuse other workflows in a job is not commonly exploited by the repositories in our dataset. Only 823 jobs (0.8%) from 254 distinct repositories (0.9%) refer to another workflow. Analysing the origin of the reused workflows, we found that the large majority (84.4%) have the same owner as the calling job. More specifically, 279 jobs (33.9%) use a workflow within the same repository and 416 jobs (50.5%) use a workflow located in another repository of the same owner. Only 128 jobs (15.5%) use a workflow located in another public repository.

Similar to Section 5.4, we analysed the names of the jobs to gain insight into their purpose. Only 40.7% of the jobs define the optional job name field. For those that do not, we considered their identifier instead (*i.e.*, the key used to define the job). We found 32,265 distinct names, of which 851 are used by 10 or more jobs. The most frequent names ($> 1\%$) were *build* (15.5%), *test* (4.2%), *analyse* (2.2%), *lint* (2%), *release* (1.7%) and *deploy* (1.5%). The first most frequent name for a social activity was *stale* (0.9%), corresponding to the task of closing issues or pull requests that have not exhibited any recent activity.

Takeaways. Studying the most frequent jobs in workflows (RQ5.3), we found that the large majority of workflows define a single job, but most repositories have two or more jobs defined in the totality of their workflows. Jobs mostly implement technical activities. It is very uncommon practice to reuse workflows in jobs.

5.6. Automation Practices in Workflow Files

Steps represent the smallest unit of work in a workflow. They correspond to individual tasks in a job that are sequentially executed to achieve the job goal. For example, in order to publish a new release on a package registry such as PyPI, a job will define steps to (1) checkout the code, (2) setup Python, (3) install dependencies, (4) execute tests, (5) build the package, and (6) upload it on PyPI. We found 576,352 steps within 108,500 jobs. On average, there are 5.2 steps per job (median is 5), 8.2 steps per workflow (median is 5) and 19.4 steps per repository (median is 10).

A step either explicitly lists the commands to be executed (through the `run:` key) or delegates this task (through the `uses:` key) to an action or a Docker image. Table 5.4 reports on the proportion of steps and repositories in function of their step type. For steps relying on the `uses:` key, we distinguish between steps referring to an action by mean of a local path, a reference to a Docker image or by specifying the name of a repository containing the action. For the latter category, we distinguish between references to the same repository, to a repository of the same owner or to another public repository.

Table 5.4: Proportion of steps and repositories w.r.t. step type and action target.

step type	Action target	% steps	% repositories
<code>run:</code>		49.9%	93.5%
<code>uses:</code>	local path	0.8%	2.0%
	Docker image	0.1%	1.8%
	same repository	0.2%	0.4%
	same owner	0.7%	4.3%
	other repository	48.3%	99.3%
total		100%	—

Around half of the steps (51.1%) use an *Action*. Nearly all repositories have at least one step referring to an Action. We observe that these steps

mostly refer to another public repository (48.3% of the steps and 99.3% of the repositories) and, to a much lower extent, to a repository belonging to the same owner (0.7% of the steps and 4.3% of the repositories). There is little to no use of Docker images (0.1% of the steps).

The other half of the steps (49.9%) run their own local commands. Analysing the content of `run:` keys, we found 139,501 distinct commands among the 287,868 steps that define one. There are 134 commands being duplicated in 100+ steps, of which 10 are duplicated in more than 1,000 steps (e.g., `npm ci`, `yarn build`, `npm test`). On average, steps execute 2.9 command lines (median is 1) for a total of 29.8 command lines per repository (median is 4).

Looking at the step names, we found 92,853 distinct names of which 360 are used by at least 100 steps. The most frequent ($> 1\%$) names are *install dependencies* (3.7%), *checkout* (2.9%), *build* (2%), *run tests* (1.4%), *test* (1.3%), *checkout repository* (1.2%) and *checkout code* (1.1%). When we distinguish names based on the type of the steps using them, we find that the use of some step names is strongly related to the presence of commands (i.e., `runs:`) or to the reuse of an existing Action (i.e., `uses:`). For example, we found that *install dependencies*, *build*, *tests* (and its variants) are mostly used for steps executing commands ($\geq 97\%$ of the steps with these names) while *checkout* (and its variants) and *setup python/node/php/etc.* are mostly used for steps relying on a reusable Action ($\geq 99\%$).

Overall, considering the names being used in at least 100 steps, we found that 47.2% of them are specialized (i.e., $\geq 75\%$ steps having that name) towards steps executing commands, and 45.3% towards steps using an Action. The remaining 7.5% names are used indiscriminately for commands and Actions, and include terms like *deploy*, *release* and *publish* among others.

Takeaways. Looking at the automation practices (RQ5.4), we found that a job defines five steps on average. Half of the steps are running their own local commands while the other half use a reusable *Action*. Nearly all repositories rely on Actions, mostly originating from other public repositories. A large majority of the most frequent tasks are implemented either almost exclusively by steps executing commands or almost exclusively by an Action.

5.7. Characteristics of used Actions

Section 5.6 revealed the common practice of steps using Actions. From the 278,122 steps relying on a Action from a public repository, we found 2,964 distinct Actions of which 724 are used at least 10 times. Table 5.5 lists the 10 most frequent Actions observed in steps and repositories.

Table 5.5: The 10 most frequent Actions in steps and repositories.

Action	% steps	% repositories
actions/checkout	35.5%	97.8%
actions/cache	7.2%	21.6%
actions/setup-node	6.6%	26.3%
actions/upload-artifact	5.9%	18.7%
actions/setup-python	5.8%	21.0%
actions/setup-java	2.6%	10.0%
actions/setup-go	2.5%	9.1%
actions/download-artifact	2.1%	6.4%
shivammathur/setup-php	1.3%	5.7%
codecov/codecov-action	1.3%	9.4%
total	70.8%	—

The most frequently used Action is `actions/checkout`, used by 35.5% of the steps and 97.8% of the repositories. The high proportion of repositories using it should not be surprising since `actions/checkout` aims to ease checking out a repository, a necessary first step for executing most of the CI/CD tasks. Other frequently used Actions are mainly related to the deployment of a specific programming language environment (*e.g.*, `setup-node` or `setup-python`). Overall, 24.2% of the steps use an Action of the form `setup-*`.

For each Action in the dataset, we identified its provider (*i.e.*, the name of the user or organization owning the repository). Considering the top 10 of Table 5.5, we observe that 8 of the most frequent Actions are officially proposed by GitHub (distributed by the `actions` provider) and used by 71.2% of the steps. The first third-party provider is `shivammathur` with the `setup-php` Action.

In the entire dataset we found 2,037 different providers, but the majority of the Actions used by steps come from just a few providers. There are only 8 providers whose Actions are called by 1,000+ steps, and 103 providers whose Actions are called by 100+ steps. The `actions` provider alone distributes 24 Actions that account for 71.7% (*i.e.*, 199,549) of the steps calling an Action. The second most frequent called provider is `docker` (7 Actions and 3.8% steps), followed by `github` (9 Actions and 3.1% steps) mostly for their CodeQL Actions,

and by `shivammathur` (2 Actions and 1.4% steps) for its `setup-php` Action.

We sought to find out more about the purpose of these Actions. However, the metadata that are required to distribute an Action on a public repository do not include anything about the Action’s purpose or category.³ The only place where this information is (partially) available is on the GitHub Marketplace.⁴ To publish an Action on the Marketplace, one has to provide additional metadata, including the primary and secondary categories the Action belongs to. Even if GitHub allows anyone to upload to the Marketplace an Action available from a public repository, to increase its visibility and reuse, not all Actions are published on the Marketplace. We managed to find 917 of the 2,964 Actions of our dataset on the Marketplace, based on the assumption that the name of the repository where an Action is developed corresponds to its unique identifier on the Marketplace. We will discuss the threats related to this assumption in Section 8.2. Out of these 917 Actions we found, 752 were correctly mapped, in the sense that the repository mentioned on the Marketplace for an Action does indeed correspond to the repository that is called by the step. These 752 Actions are used in 158,441 steps (*i.e.*, 57% of the steps relying on a public Action).

Table 5.6 reports on the most frequent ($> 1\%$) categories for these Actions, as well as on the proportion of steps and repositories using them. Since we did not observe major differences between primary and secondary categories, we report exclusively on the former.

Table 5.6 reveals that most categories correspond to technical aspects of software development. The primary categories containing the highest proportion of Actions (as well as steps and repositories) are *Utilities* (23.9% of Actions) and *Continuous integration* (13.3% of Actions). These two “catch-all” categories include very diverse Actions to check out repositories, set up environments, create releases, etc. The third most widely used category is *Container CI* even if only 2.3% of the Actions are part of this category. It includes Actions to log in to a Docker registry, to run a Dockerfile, or to set up specific services (*e.g.*, a MySQL database or a Redis instance). A few categories include socially related Actions, such as *Project management* and *Code review*. These two categories notably propose Actions to create or triage issues, to detect and lock stale issues, or to add specific comments in existing issues or pull requests. Other social categories include *Chat*, *Reporting* and *Community*. The two former ones propose Actions to notify on Slack, Discord, IRC,

³<https://docs.github.com/en/actions/creating-actions/metadata-syntax-for-github-actions>

⁴<https://github.com/marketplace?type=actions>

Table 5.6: Most frequent (> 1%) primary categories and the proportion of Actions, steps and repositories using them.

Action category	% Actions	% steps	% repositories
Utilities	23.9%	88.2%	99.4%
Continuous integration	17.3%	4.9%	12.9%
Publishing	7.2%	0.7%	2.9%
Deployment	6.9%	0.2%	0.6%
Code quality	6.1%	0.3%	1.0%
Project management	5.2%	0.4%	1.6%
Dependency management	4.4%	1.0%	2.5%
Code review	4.1%	0.1%	0.6%
Testing	3.3%	0.7%	1.5%
Open Source management	3.3%	0.1%	0.5%
Container CI	2.3%	2.2%	5.5%
Chat	1.9%	0.3%	0.5%
Reporting	1.7%	0.1%	0.4%
Community	1.6%	0.01%	0.1%
Security	1.6%	0.1%	0.3%
<i>unspecified</i>	4.9%	0.2%	0.6%
total	95.4%	99.5%	–

etc. while the latter includes Actions to ensure CLA adherence or to welcome newcomers.

Takeaways. Using Actions in steps is a common practice (RQ5.5). A few Actions concentrate most of the reuse, and most of them are distributed by GitHub and belong to the *Utilities* or *Continuous integration* categories.

5.8. Versioning Practices Used for GitHub Actions

When a step uses a predefined Action, in addition to specifying the name of the repository hosting the Action it can optionally specify which *version* of the Action should be executed, by means of a git reference. This reference can be a commit SHA (*e.g.*, `@753c60e0`), a branch name (*e.g.*, `@main`), or a git tag (*e.g.*, `@v2.1.3`). If no reference is specified, the latest version of the Action is executed. For security and stability reasons, GitHub recommends pinning an Action to a full-length commit SHA. While this is the most secure option, specifying a tag is more convenient since GitHub’s release management support advocates the creation of a tag corresponding to the version number

of a new release. This makes it easy for a step using an Action to specify which version of the Action should be executed.

Table 5.7 shows the number and proportion of steps in function of the git reference type used for Actions, and the origin of these Actions. We distinguish between no git reference (*none*), a reference to a specific *commit SHA*, a reference to a *version tag* and a reference to a branch or another tag (*branch/tag*).

Table 5.7: Number and proportion of steps w.r.t. Action target and git reference type.

Action origin	reference type	steps		
		#	% target	% all
local path	<i>none</i>	4,397	100.0%	1.5%
	branch/tag	2	0.0%	0.0%
same repository	commit SHA	2	0.2%	0.0%
	version tag	123	12.8%	0.0%
	branch/tag	833	87.0%	0.3%
same owner	<i>none</i>	1	0.0%	0.0%
	commit SHA	123	3.0%	0.0%
	version tag	2,641	63.9%	0.9%
	branch/tag	1,368	33.1%	0.5%
other repository	<i>none</i>	2	0.0%	0.0%
	commit SHA	4,601	1.7%	1.6%
	version tag	258,647	93.0%	89.9%
	branch/tag	14,872	5.3%	5.2%

We observe major differences in the git references used to refer to an Action in function of the origin of the Action. For instance, Actions on a local path are nearly exclusively referred to without any specific reference, while the vast majority of Actions within the same repository (but referred to with the full name of the repository) are referred to with a branch or a tag name or, to a much lower extent, with a version tag. The opposite can be observed for Actions from repositories of the same owner: they are mostly referred to using a version tag and, to a lower extent, with a branch or a tag name. The situation is even more marked for Actions coming from other public repositories: 9 out of 10 steps in this large subset use a version tag to refer to an Action.

Assuming adherence to *semantic versioning*, GitHub recommends specifying the version tag by including only its major component (*e.g.*, `@v2` instead of `@v2.1.3`) in order to receive critical fixes and security patches while still maintaining compatibility. This recommendation seems to be widely followed, since 89.9% of the version tags used to refer to an Action include only a major component (*e.g.*, `@v2`), 0.9% a minor component (*e.g.*, `@v2.1`) and 9.2% a

patch component (*e.g.*, `@v2.1.3`).

Referring to a version using only its major component has clear advantages if we assume adherence to semantic versioning (Decan *et al.*, 2017). However, since versions of an Action are identified using git tags, this means that the Action maintainer must *move* some of these tags each time a new version of the Action is released (*e.g.*, moving `@v2` and `@v2.1` from `@v2.1.3` to `@v2.1.4` when version 2.1.4 is released). Unless automated, this introduces an additional burden on the maintainers. Forgetting to update these tags when a minor or a patch update is released for an Action implies that the workflows that depend on it do not automatically benefit from the bug and security fixes provided by the update.

Moreover, we found that 16.4% of the major components used in version tags to refer to a reusable Action do not target the highest major release of the corresponding Action, *i.e.*, they are relying on a lower major train. As such, dependent workflows do not benefit from the latest bug and security fixes of the Action unless these changes are backported to lower major trains as well (Decan *et al.*, 2021).

Takeaways. Versioning practices for Actions (RQ5.6) differ based on the Action origin. Around 9 out of 10 steps use a version tag when referring to an Action. Most of the version tags only specify the major component of the targeted version, and one sixth of them refer to a lower major train.

5.9. Discussion

5.9.1 The GitHub Actions Ecosystem

The information in Section 5.7 indicate that the ability to reuse Actions in GitHub Actions leads to a new emerging ecosystem that is worthy of being studied in its own right. The reuse of Actions in steps is a common practice that allows developers to easily integrate (sometimes complex) tasks without having to code them. The ability for a CI/CD tool to provide reusable components is not something specific to GitHub Actions, since many competing CI/CD tools are providing similar mechanisms. For instance, CircleCI introduced *orbs* in 2018, one year before GitHub Actions⁵; and Jenkins has been providing community-contributed plugins for years through `plugins.jenkins.io`. However, by March 2022, GitHub Actions offers more than 12,000 reusable

⁵<https://circleci.com/blog/announcing-orbs-technology-partner-program/>

components on its Marketplace, about 4 times as many as CircleCI orbs and more than 6 times higher than Jenkins plugins, and there are likely thousands more available Actions in public GitHub repositories.

Given this wide availability of reusable Actions, GitHub Actions should be studied as a “software ecosystem” that bears many similarities to ecosystems of reusable software libraries distributed by package managers, *e.g.*, npm, Cargo, RubyGems, Maven, PyPI and the like. The parallel with such packaging ecosystems is quite obvious: automated workflows, as software *clients*, frequently express *dependencies* towards reusable Actions that can exist in different *versions* or *releases*. Packaging ecosystems are known to suffer from a large number of issues in the reusable artefacts they distribute, and each of these has been an active topic of investigation. Well-known challenges include obsolescence or outdatedness (Cogo *et al.*, 2021; Decan *et al.*, 2018a), dependency issues (Decan *et al.*, 2017; Kula *et al.*, 2018; Soto-Valero *et al.*, 2021), breaking changes (Decan *et al.*, 2017; Dietrich *et al.*, 2019), security vulnerabilities (Decan *et al.*, 2018b; Zimmermann *et al.*, 2019), and so on.

The GitHub Actions ecosystem is likely to suffer from very similar issues, and these issues will continue to become more important and more impactful, as the number of reusable Actions continues to grow at a rapid pace. Therefore, there is an urgent need for further research as well as appropriate tooling to support developers of reusable Actions and workflows, especially since these issues may not only affect the workflows but also wide ranges of projects that use them.

Addressing these challenges requires further empirical research and improved tooling to support Action authors and workflow maintainers. An initial step in this direction is GitHub’s built-in Dependabot service, which has supported reusable Actions since January 2022, but existing solutions remain limited in scope. The subsequent chapters of this dissertation build on these observations by examining workflow and Action evolution, reuse practices, and maintenance in the GitHub Actions ecosystem.

5.9.2 Security Concerns

While security concerns are important to deal with for any software project, it is known that the attack surface of security issues has become several orders of magnitude higher due to the widespread dependence on reusable software libraries that can have deep transitive dependency chains (Alfadel *et al.*, 2021; Decan *et al.*, 2018b; Düsing & Hermann, 2021; Zimmermann *et al.*, 2019). The reliance on CI/CD tools to automate development activities in software projects continues to increase this attack surface considerably. This concern

was explicitly raised by developers during the interviews reported in the previous chapter, with two participants emphasizing that CI/CD automation constitutes a major security risk: *“In house we have two GitHub and GitLab CI systems that we need to maintain. It’s a major security concern because, from that automation you can basically run any code on it.”* and *“CI/CD are very important to secure the build chain, that we should focus in the future into the aspect of securing the toolchain.”*

For GitHub Actions in particular, multiple examples of security issues with potentially disastrous consequences have been reported, such as manipulating pull requests to steal arbitrary secrets,⁶ injecting arbitrary code with workflow commands⁷ or bypassing code reviews to push unreviewed code.⁸ A developer we talked to during the interviews reported in the previous chapter specifically mentioned *“You can open a pull request, build the package, and then we will deliver it. And when you do that from a pull request, there are issues with the security considerations about the credentials, because anyone could modify workflows or inject code, get access to the credentials and then access to the upload process. [...] When it’s open to everyone, you need to be careful. [...] What we do with GitHub Actions, we build in one workflow, and we have a second workflow which does the upload or the shipping or the delivery with credentials which are not exposed in the build pipeline for the pull request.”*

Relying on reusable Actions from third-party repositories or even from the Marketplace further increases the vulnerability attack surface. Since a job executes its commands within a runner shared with other jobs from the same workflow, individual jobs in a workflow can compromise other jobs they interact with. For example, a job could query the environment variables used by a later job, write files to a shared directory that a later job processes, or even more directly interact with the Docker socket and inspect other running containers and execute commands in them.⁹

Despite these risks, Section 5.7 revealed that it is common practice to rely on reusable Actions. As a general rule of thumb, GitHub recommends to only use Actions whose creator can be trusted. However, even Actions from trusted creators can be compromised. For example, an attacker having gained write access to the repository of a trusted Action can change its code and commands in order to compromise the repositories depending on this Action.

To further reduce the risks of using compromised Actions, GitHub suggests

⁶<https://blog.teddykatz.com/2021/03/17/github-actions-write-access.html>

⁷<https://packetstormsecurity.com/files/159794/GitHub-Widespread-Injection.html>

⁸<https://medium.com/cider-sec/bypassing-required-reviews-6e1b29135cc7>

⁹<https://tinyurl.com/2fwhfpx3>

referring to reusable Actions through their unique commit SHA, to avoid unintentionally using a compromised Action that may have its code changed and may be able to steal secrets: “Pinning to a particular SHA helps mitigate the risk of a bad actor adding a backdoor to the Action’s repository, as they would need to generate a SHA-1 collision for a valid Git object payload.”¹⁰ Unfortunately, we observed in Section 5.8 that this recommendation is not really followed in practice. The very large majority of steps relying on a reusable Action use a version tag (rather than a commit SHA) when referring to an Action, and most of the version tags only specify the major component of the targeted version, implying that any new compromised version of an Action will be automatically adopted by most dependent workflows.

Recent work has begun to quantitatively investigate security risks associated with reusable CI/CD components, including a study by Onsoni Delickeh *et al.* (2024) and Onsoni Delickeh and Mens (2024) that analyzes security vulnerabilities in reusable GitHub Actions Actions. This line of research provides important initial evidence that Action reuse can introduce non-trivial security risks and increase the attack surface of dependent workflows.

However, despite these efforts, the broader impact of reusable Actions on project-level security is not yet fully understood. The growing body of work in this area indicates that research in this direction has already started, but also underscores the need for continued and more comprehensive investigation.

5.10. Threats to Validity

We follow the structure recommended by Wohlin *et al.* (2012) to discuss the main threats to validity of this chapter.

Threats to *construct validity* concern the relation between the theory behind the experiment and the observed findings. They can be mainly due to imprecision in the measurements we performed. We detected the use of automated workflows in GitHub repositories on the basis of the presence of a YAML file in the `.github/workflows` folder. This approach leads to an overestimation since the presence of a YAML file does not necessarily imply that the corresponding workflow is actually being triggered and used. Another threat to construct validity stems from how we identified the git reference type used to reference public actions in steps. We relied on a heuristic to detect whether the git references correspond to a version number (via a regular expression), to

¹⁰<https://docs.github.com/en/actions/security-guides/security-hardening-for-github-actions>

a commit SHA (based on the git reference length) or to a tag or branch name (all remaining git references). This naive heuristic seemed to be effective since we did not find any false positives after having manually checked 104 distinct randomly selected cases. Another threat to validity stems from how we interpreted the identifier labels for workflows and jobs. We relied on these labels in Section 5.4 and Section 5.5 to understand the purpose of the workflows and jobs being defined. While some workflows and jobs may have a label that does not reflect their purpose, we believe these cases to be rare, as the goal of a label is to provide an indication of the nature of the workflow or job, and not to mislead practitioners or researchers.

Threats to *internal validity* concern choices and factors internal to the study that could influence the observations we made. One of such choices relates to how we mapped actions used in steps with the ones distributed on the GitHub Marketplace. We relied on the assumption that the name of the repository where an action is developed corresponds to its unique identifier on the Marketplace. This led both to false positives (e.g., action myci-actions/checkout is identified as checkout while checkout on the Marketplace is provided by actions/checkout) and false negatives (e.g., action actions/setup-node is identified as setup-node-jsenvironment on the Marketplace). False positives were addressed by comparing the repository referred from the calling step with the repository listed on the action page on the Marketplace. We were unable to address the false negatives, as this would require extracting all actions from the Marketplace in order to obtain their development repositories. Unfortunately, GitHub does not provide a complete list of actions in the Marketplace, nor an API to obtain them. We remain confident that the findings of Section 5.7 are representative, since we managed to map correctly 727 actions, accounting for 57% of the steps relying on an action.

Threats to *conclusion validity* concern the degree to which the conclusions derived from our analysis are reasonable. Since our conclusions are mostly based on quantitative observations, they are unlikely to be affected by such threats.

Threats to *external validity* concern whether the results can be generalized outside the scope of this study. One such threat was our decision to study active repositories having at least 100 stars and 100 commits, aiming at excluding abandoned, personal or experimental repositories that do not necessarily correspond to software development (Kalliamvakou *et al.*, 2014). This implies that we have no insight into the use of GitHub Actions in smaller or less active repositories, and it could be the case that GitHub Actions is used for different purposes in those repositories (e.g., to compile L^AT_EX files).

5.11. Summary and Conclusions

This chapter presented a large-scale quantitative analysis of GitHub Actions workflow usage in GitHub repositories. By examining 29,778 repositories and 70,278 workflows, the study provided the first comprehensive, ecosystem-level view of how GitHub Actions is used in practice at scale.

The findings revealed that GitHub Actions workflows are predominantly associated with highly active repositories, characterized by greater levels of contributor participation, commits, issues, and pull requests. Workflows are mainly used for development-oriented automation and are most often triggered by common events such as pushes and pull requests. At the job level, reuse plays a central role: approximately half of all workflow steps invoke reusable Actions, with a strong reliance on Actions published by GitHub itself. The widespread use of some variants of semantic versioning further suggests a relatively mature approach to reuse and dependency management within the ecosystem.

At the same time, this modular reuse introduces new security and sustainability concerns. Dependence on third-party Actions increases the attack surface and raises questions related to trust, maintenance responsibility, and long-term availability. These risks point to the need for improved tooling and best practices for vetting, versioning, and evolving reusable automation components.

While this chapter deepens our understanding of how GitHub Actions is used in practice, several open questions remain. In particular, the evolutionary dynamics of workflows, such as how they change over time and how maintainers adapt them to emerging CI/CD needs, are not yet well understood. It also remains unclear to what extent traditional ecosystem challenges, such as dependency decay, outdated components, or migration costs, manifest in this rapidly evolving automation landscape.

Finally, it is important to note that the quantitative findings reported in this chapter reflect the state of the GitHub Actions ecosystem at the time the study was conducted. Given the fast-paced evolution of CI/CD tooling, platform policies, and automation practices on GitHub, current usage patterns may differ from those observed in this analysis, underscoring the need for continued longitudinal and complementary qualitative investigations.

This chapter addresses the second thesis goal by providing a comprehensive characterization of real-world GitHub Actions usage patterns, workflow structures, and Action reuse practices. The empirical foundation established here reveals that GitHub Actions has become a central component of modern

software development workflows, with modular reuse being a dominant practice. However, this analysis also exposed critical gaps in our understanding of how these workflows evolve and are maintained over time.

The findings from this chapter raise several questions that form the basis for the subsequent chapters of this dissertation. First, the prevalence of Action reuse and the reliance on semantic versioning suggest that workflows face similar maintenance challenges as traditional software evolution, however this remains to be explored further. The next chapter addresses this gap by investigating how GitHub Actions workflows evolve over time, examining patterns of workflow modifications and the lifecycle of individual workflows.

Methodology and Tooling for Analyzing Changes in Workflow Files

“The science of today is the technology of tomorrow.”

Edward Teller

Having established the widespread adoption and practical use of GitHub Actions in the previous chapter, we now turn our attention to understanding how workflow files evolve over time within software repositories. Specifically, we aim to identify whether workflows reflect established principles of software evolution, such as those proposed by Lehman (1996), including continuous growth, continuous change, and the need for maintenance-driven modifications. To enable further analysis, we introduce and employ a dedicated differencing tool, *gawd*, designed to capture and analyze fine-grained changes in GitHub Actions workflows across their version histories.

This chapter combines insights from our findings of previous chapter with the capabilities of *gawd* to pave the way to study the nature, frequency, and scope of modifications in workflow files. In doing so, it addresses the third thesis goal by examining the evolution of workflows at scale. We analyze how often and in what ways workflows change, based on a tool that we developed to extract the contexts in which these modifications arise. We reveal broader patterns of workflow lifecycle and adaptation across repositories. This perspective provides the empirical foundation needed to understand workflows as evolving software artifacts shaped by continuous maintenance and iterative refinement.

6.1. A Preliminary Study of GitHub Actions Workflow Changes

6.1.1 Introduction

Workflow configuration files are the main components to configure GitHub Actions pipelines. Just like ordinary source code, they are developed and modified throughout the project's lifetime to meet the needs of developers.

Knowing when, why and how developers modify workflow files can be helpful to improve CI/CD practices, to detect common patterns and mistakes developers do in their workflows, and to create tools to assist them in writing and maintaining workflows.

As preliminary steps towards such a comprehension, this chapter aims to characterise the changes made to these workflow files over their lifetime. To do so, we study two research questions, based on an extracted dataset of 22,733 repositories accounting for 4,127,760 weekly snapshots of workflow files over a 34-month period from November 2019 to September 2022.

The first research question (RQ 6.1) seeks to quantify the coarse-grained changes made to GitHub Actions workflows and is further divided into three specific subquestions:

RQ 6.1.1 *When do repositories start using GitHub Actions workflows?*

RQ 6.1.2 *Which types of coarse-grained changes are workflows subject to?*

RQ 6.1.3 *When do the different types of coarse-grained changes occur in workflows?*

The second research question (RQ 6.2) is concerned with quantifying the fine-grained evolution of workflow files by examining the changes made within their contents. To guide this investigation, we focus on line-level modifications and address the following two subquestions:

RQ 6.2.1 *Which types of line-based changes are workflow files subject to?*

RQ 6.2.2 *When do different types of line-based changes occur in workflows?*

6.1.2 Motivating Example

In order to use GitHub Actions for a repository, one or more *YAML* workflow files must be created and stored within the `.github/workflows` folder. As with

any other software development component, workflow files can evolve over time to better serve their purposes.

Figure 6.1 exemplifies a visual difference of some changes made to a workflow file that automates the building, testing and code coverage analysis of some Java project. On the left is the old version, with lines highlighted in red representing removed or changed lines, and the dark red parts highlighting the parts of the line that were changed. On the right is the new version, with lines highlighted in green representing new or changed lines, and the dark green parts highlighting the changes that were made w.r.t. the previous version.

One can observe that changes may occur in different locations for different reasons. For instance, line 2 illustrates a change related to the events that trigger the workflow. This particular change can be considered a behaviour-preserving refactoring to simplify the workflow file, since declaring events without specifying the branch will use the default branch of the repository (in this case: `master`). A common change consists of adding new steps (*e.g.*, step ‘Publish Test Report’ on lines 17-19 on the right) or adding more lines to existing steps (*e.g.*, lines 12 and 14 on the right that add extra arguments for the Java setup; line 26 on the right that added an extra argument for the JaCoCo badge generator). Line 5 on the right is an example of a change to a job, by adding an extra name to it. Another common change is modifying the contents of existing lines, for example to update the version of the Action being used (*e.g.*, from `actions/checkout@v2` to `actions/checkout@v3`, from `actions/setup-java@v1` to `actions/setup-java@v3`, and so on), to edit the name of a step (line 31 on the right), to modify the Java version to use for the build (line 13 on the right), and so on.

This example shows some of the possible changes in workflow files and justifies the need for studies on when, why, and how developers modify workflow files. Such studies are likely to improve CI/CD practices, for example, by identifying common patterns and issues during workflow modifications and by providing tools to assist developers in writing and maintaining workflows.

6.1.3 Data Extraction

To study the evolution of workflow files, a large dataset of GitHub repositories relying on GitHub Actions is required. We used the SEART GitHub search engine¹ by Dabic *et al.* (2021) to select repositories. To mitigate the usual threats related to software repository mining (Kalliamvakou *et al.*, 2014), we excluded repositories that are used only for experimental or personal purposes,

¹<https://seart-ghs.si.usi.ch>

```

1 name: Java CI with Maven
2 on:
3   push:
4     branches: [ master ]
5   pull_request:
6     branches: [ master ]
7 jobs:
8   build:
9     runs-on: ubuntu-latest
10    steps:
11      - uses: actions/checkout@v2
12        name: Set up with Java 11
13      - uses: actions/setup-java@v1
14        with:
15          java-version: 11
16      - name: Build with Maven (including running of all tests)
17        run: mvn -B package --file pom.xml
18      - name: Generate JaCoCo Badge
19        id: jacoco
20        uses: cicirello/jacoco-badge-generator@v2
21        with:
22          generate-coverage-badge: true
23          generate-branches-badge: true
24      - name: Log coverage percentage
25        run: |
26          echo "coverage = ${ steps.jacoco.outputs.coverage }"
27          echo "branch coverage = ${ steps.jacoco.outputs.branches }"
28      - name: Commit and push the badge (if it changed)
29        uses: EndBug/add-and-commit@v7
30        with:
31          default_author: github_actions
32          message: 'commit badge'
33          add: '*.svg'
34      - name: Upload JaCoCo coverage report
35        uses: actions/upload-artifact@v2
36        with:
37          name: jacoco-report
38          path: target/site/jacoco/

```

```

1 name: Java CI with Maven
2 on: [push, pull_request]
3 jobs:
4   build:
5     name: build and analyse
6     runs-on: ubuntu-latest
7     steps:
8       - uses: actions/checkout@v3
9       - name: Set up with Java 17
10        uses: actions/setup-java@v3
11        with:
12          distribution: 'temurin'
13          java-version: 17
14        cache: 'maven'
15       - name: Build with Maven (including running of all tests)
16        run: mvn -B package --file pom.xml
17       - name: Publish Test Report
18        if: ${{ always() }}
19        uses: scalameta/action-surefire-report@v1
20       - name: Generate JaCoCo Badge
21        id: jacoco
22        uses: cicirello/jacoco-badge-generator@v2
23        with:
24          generate-coverage-badge: true
25          generate-branches-badge: true
26          generate-summary: true
27       - name: Log coverage percentage
28        run: |
29          echo "coverage = ${ steps.jacoco.outputs.coverage }"
30          echo "branch coverage = ${ steps.jacoco.outputs.branches }"
31       - name: Commit and push the svg badges and the json coverage summary (if it changed)
32        uses: EndBug/add-and-commit@v9
33        with:
34          default_author: github_actions
35          message: 'commit coverage badge and summary'
36          add: '*.svg *.json'
37       - name: Upload JaCoCo coverage report
38        uses: actions/upload-artifact@v3
39        with:
40          name: jacoco-report
41          path: target/site/jacoco/

```

Figure 6.1: Visual diff of some changes made to a workflow file for a Java project hosted in a GitHub repository.

or that exhibit minimal evidence of software development activity. To do so, we selected repositories created before 2022 that were still active in 2022, had at least 100 stars and 100 commits, and were not forks. Considering these constraints, the final list of repositories included 62,673 instances.

On September 2022, we locally cloned these repositories to identify the presence of GitHub Actions workflow files in the `.github/workflows` directory of their default branch (as reported by the GitHub API) and found 22,733 repositories satisfying this criterion. Since our goal is to study the evolution of the workflow files in these repositories, we relied on a combination of the `git rev-list` and `git checkout` CLI tools to materialize the content of each workflow file in each repository for every Monday between November 2019 (the official release date of GitHub Actions) and September 2022, accounting for 148 time points. By considering weekly snapshots instead of all the commits that modified the workflow files, we mitigate the usual threats related to commits performed on parallel branches that are eventually merged (Bird *et al.*, 2009). This resulted in a dataset of 4,127,760 workflow file snapshots (of which 271,422 are unique) in 22,733 GitHub repositories.

Figure 6.2 shows the evolution of the number of repositories and workflow files through time. We observe that both numbers are continuously increasing through time, indicating that more and more repositories are making use of

GitHub Actions and more and more workflow files are created in these repositories. At the end of the observation period, there are 65,067 workflow files spread in 22,733 repositories. The figure also reveals a slight increase in the evolution of both numbers in November and December 2020. This coincides with restrictions imposed by Travis CI on its free plan for public repositories, which caused many repositories to switch from Travis CI to GitHub Actions during these two months, as already observed by Golzadeh *et al.* (2021c).

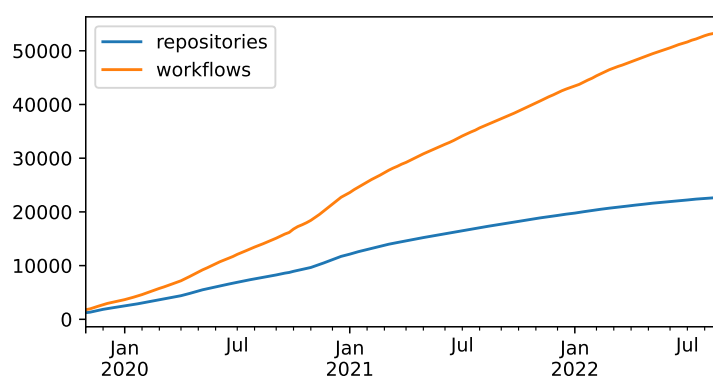


Figure 6.2: Evolution of the number of repositories and GitHub Actions workflows in our dataset.

6.1.4 GitHub Actions Adoption by Repositories

To investigate the adoption of GitHub Actions in GitHub repositories (RQ 6.1.1), we analyzed the time it took for repositories to start using it. Our findings in Chapter 4 have reported that GitHub Actions has become the dominant CI/CD tool on GitHub. This section aims to understand the time it takes for repositories to adopt GitHub Actions as their CI/CD tool.

We distinguish between the repositories that already existed when GitHub Actions was introduced on GitHub in November 2019 and those that were created after. Indeed, repositories that were created before the introduction of GitHub Actions were likely to use another CI/CD tool before migrating to GitHub Actions and such a migration does not come for free. In these cases, we are interested in the time they took to adopt GitHub Actions since its official release. On the other hand, repositories that were created after the introduction of GitHub Actions are more likely to adopt it as their CI/CD tool because of its deep integration into GitHub. In these cases, we are interested

in the time they took to adopt GitHub Actions since these repositories were created.

We therefore divided the repositories in our dataset into two categories: those that already existed before the public release of GitHub Actions, accounting for 18,805 repositories; and the remaining 3,928 that were created after.

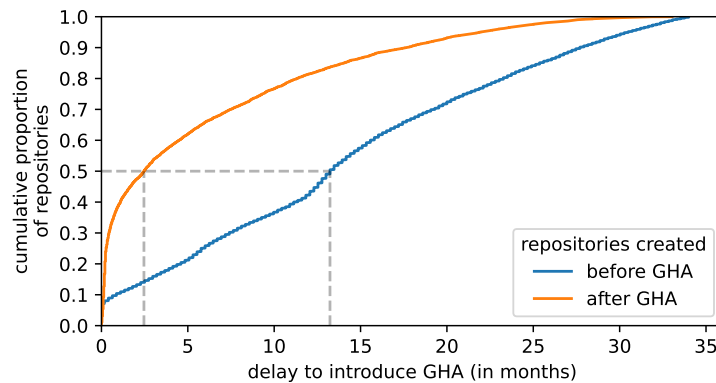


Figure 6.3: Cumulative proportion of repositories in function of the time (in months) to start using GitHub Actions workflows.

Figure 6.3 shows the cumulative proportion of repositories in function of the time they took to start using GitHub Actions workflows. It is worth recalling that our dataset only contains repositories that are still using GitHub Actions in the latest considered snapshot, explaining why the proportions reach 100%.

Focusing first on the repositories that were created after GitHub Actions public release (orange line in Figure 6.3), we observe that it takes less than 3 months for 50% of them to adopt GitHub Actions, as indicated by the left-most dotted line. After only 10 months, this proportion reaches 75%. The adoption rate for the repositories that already existed when GitHub Actions was released (blue line) is much lower: only 15% of these repositories adopted GitHub Actions after 3 months and it required 10 months to reach 50% of the repositories (as indicated by the rightmost dotted line), and even 21 months to reach 75%.

This supports our hypothesis that it takes more time for repositories already in place to start adopting GitHub Actions, likely because they were already using another CI/CD tool before GitHub Actions existed. This is implying a longer delay to start using GitHub Actions due to the technical or organizational difficulties that may come with a migration to a new CI/CD solution,

or simply due to the lack of need to carry out such a migration as reported in Chapter 4. On the other hand, adopting GitHub Actions in newer repositories is much easier thanks to the tight integration of GitHub Actions with GitHub, its ease of use, its low learning curve, and the ease of setting up new workflow files from scratch based on suggested configurable templates (Saroar & Nayebi, 2023).

Takeaways. The time to start using GitHub Actions workflows (RQ 6.1.1) depends on whether the repository was created before or after GitHub Actions’s official release. A majority of the repositories that were created after GitHub Actions adopted it within a few months. On the other hand, it took more than one year since GitHub Actions release for most of the older repositories to start using GitHub Actions workflows.

6.1.5 Types of Coarse-grained Changes Affecting Workflows

This section aims to identify the kind of coarse-grained changes workflow files are subject to (RQ 6.1.2). We distinguish between the four following types of changes: *addition*, *modification*, *renaming*, or *removal* of workflow files. To keep track of these changes, we attributed to each workflow file a unique identifier that is preserved through renamings. It is worth mentioning that some of these changes can co-occur (e.g., a file can be renamed at the same time as its contents is modified) while some changes are causally dependent (e.g., a workflow file can only be removed after having been added previously).

We detect the four different types of changes as follows. The *addition* of a workflow file is detected when the workflow file is seen for the first time in a snapshot. Similarly, the *removal* of a workflow file is detected when the file is no longer visible in a snapshot. A *modification* of a workflow file is detected in a snapshot by comparing its contents in the current snapshot with its contents in the previous snapshot. These modifications are detected by comparing the SHA-256 hashes of consecutive workflows files. Finally, the *renaming* of a workflow file is detected based on the following heuristic. The heuristic detects a renaming from file *A* to file *B* when *A* is removed at the same time that *B* is added. If *A* and *B* have exactly the same contents (i.e., they have the same SHA-256 hash), we consider that *A* was renamed to *B*. If *A* and *B* differ in their content, we check whether there is no other workflow file *C* that was added or removed at the same time. If there is no such *C*, then we consider that *A* was renamed to *B*. By doing so, we ensure we are not

exposed to false positives even if this implies we may miss some renamings, e.g., in the case multiple workflow files are renamed at the same time.

For each repository and each pair of consecutive snapshots, we relied on the above approaches to detect changes. The most frequent workflow change type is *modification*, accounting for 73% of all changes. The second most frequent change type is *addition*, accounting for 22.8% of all changes. *Removal* is uncommon, accounting for less than 3.9% of changes. *Renaming* is the least common change type, accounting for 0.1% of all changes over the entire considered period.

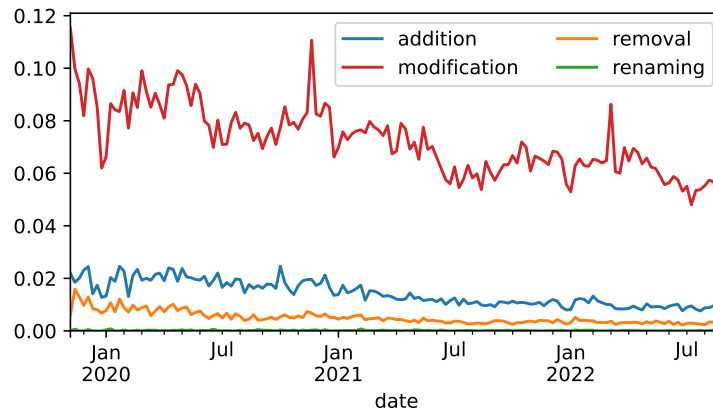


Figure 6.4: Evolution of the proportion of repositories exhibiting different workflow change types over time.

Since a repository may have many workflow files being changed at the same time, we also computed for each snapshot the proportion of repositories exhibiting the *addition*, *removal*, *modification* and *renaming* of a workflow file. Figure 6.4 shows the evolution of these proportions. We observe that the most frequent change type is *modification*, exhibited in around 9% of the repositories at the beginning of the observation period and slowly decreasing to around 6% of the repositories at the end of the observation period. Unsurprisingly, *additions* and *removals* are less frequently observed in repositories, ranging from 2.4% to 0.7% and from 1.5% to 0.2% of the repositories, respectively. Finally, *renamings* are barely never observed in the considered repositories.

Takeaways. Looking at the type of coarse-grained changes (RQ 6.1.2), each week, on average, 7.2% of the repositories modify a workflow file, while 1.4% of them add a new workflow and 0.5% remove a workflow.

6.1.6 Temporal Patterns of Coarse-grained Workflow Changes

Section 6.1.5 reported on the various types of changes that occur in workflow files. In the current section, we aim to understand when those changes occur with respect to the adoption of GitHub Actions in each repository (RQ 6.1.3). We posit that the first few weeks after introducing GitHub Actions, workflow maintainers are likely to make many changes until the workflows reach a stable state and that, afterwards, only occasional changes are being made. In order to verify this hypothesis, for each change detected in Section 6.1.5, we computed the time between the introduction of GitHub Actions in the corresponding repository and the date of the change to the workflow file.

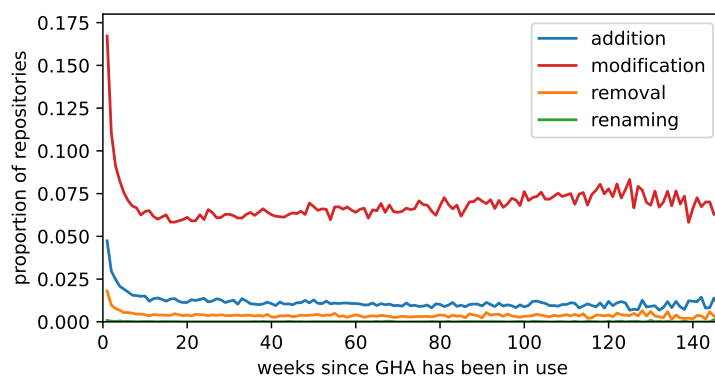


Figure 6.5: Proportion of repositories exhibiting different workflow change types as a function of time elapsed (in weeks) since adoption of GitHub Actions workflows.

Figure 6.5 shows the proportion of repositories that exhibit a change in function of the time elapsed (in weeks) since workflows were introduced in each repository. We observe that the proportion of repositories exhibiting a change, regardless of the change type, is higher during the first weeks, and that this proportion quickly decreases through time to reach a quite stable value. For instance, more than 15% of the repositories modified their workflow files during the first week, but this proportion decreases to around 6% after six weeks only. Similarly, the proportion of repositories adding workflow files decreased from around 5% in the first week to approximately 1.5% after six weeks.

The increasing variation one can observe in later weeks in Figure 6.5 is likely a side effect of the much lower number of repositories that have been using GitHub Actions for 100 weeks or more. For instance, while we have 22K

repositories for the first week, we only have 4.8K repositories at week 100, 2.4K at week 120, and 574 at week 140.

These observations suggest that workflow files follow Lehman’s evolution laws (Lehman, 1996) of *continuing change* (workflows are regularly modified through time) and *continuing growth* (at any given point in time, the proportion of workflow additions exceeds that of removals, resulting in net workflow growth).

Takeaways. Repositories are more likely to change their workflow files (RQ 6.1.3) within the first weeks after having adopted GitHub Actions. Nevertheless, we observe that each week, around 6% of the repositories modify a workflow file. This confirms that workflows are subject to the laws of *continuing change* and *continuing growth*.

Although it is not surprising that workflow files are regularly modified in order to integrate new pipelines or new functionalities, we postulate that part of the observed modifications are the consequence of the difficulty to debug, test and validate workflows. This challenge was already identified by Saroar and Nayebi (2023) and in a qualitative study conducted in Chapter 4: “*You will see that you do a lot of typos and try to run the CI/CD 20 times until it works once. You copy-paste some examples from the Internet, you adapt it, but you forget to like there’s a lot of details. It’s often YAML files that are really prone to mistakes. So you make [lots of] commits until you get to the result you want to have. And there’s no way to pre-test it on your local machine. So you just commit, push, wait for the build to run, and then look at the results.*”

6.1.7 Types of Line-based Changes Affecting Workflows

As a first step towards studying the fine-grained changes in GitHub Actions workflows, we aim to identify how frequently lines are added, removed and modified in workflow files (RQ 6.2.1). To do so, we relied on the CLOC command-line tool (Danial, 2021). CLOC is a tool that counts the number of lines in files. It has an option to compare two files, and reports on the number of lines that were *added*, *removed*, *modified* and *untouched* between the two files, distinguishing between lines of *code*, *blank* lines and *comments*.

We applied CLOC on all the workflow files that were modified, comparing the contents of each workflow file with the contents of the previous version of this workflow file. We found that the very large majority (97.6%) of the changes are related to lines of code while other changes are related to blank and commented lines.

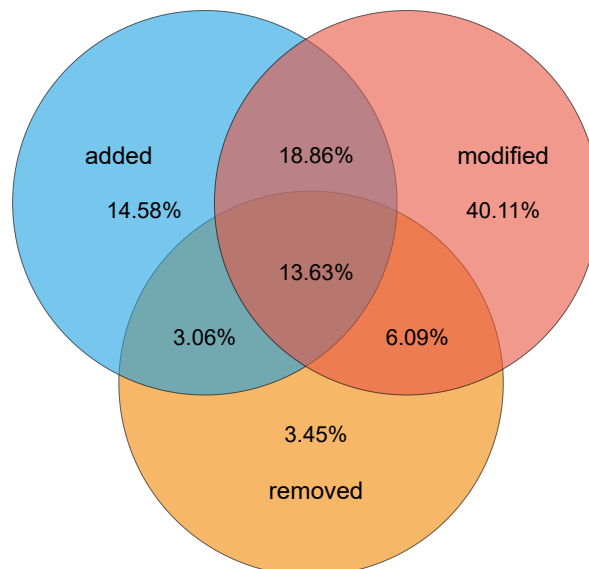


Figure 6.6: Percentage of workflow file changes containing added, removed or modified lines.

We also looked at how frequently lines are added, removed or modified. Figure 6.6 shows a Venn diagram reporting on the proportion of workflow modifications in which lines were added, modified or removed (or any combination of those). We observe that 40.11% of the changes consist of line modifications only, and lines are modified in 78.69% ($40.11\% + 18.86\% + 6.09\% + 13.63\%$) of the changes. Adding lines is the second most frequent change observed in workflow files, especially in combination with modifying lines. For instance, half of the changes ($50.13\% = 14.58\% + 18.86\% + 3.06\% + 13.63\%$) involves adding lines, while “only” 26.23% of the changes ($= 3.45\% + 3.06\% + 6.09\% + 13.63\%$) involve removing lines. It is noteworthy that removing lines alone is infrequent.

Takeaways. Our study on the type of line-based changes (RQ 6.2.1) shows that nearly all the changes made to workflow files involve lines of code accounting for almost 9 out of 10 lines added, removed or modified. Modifying and adding lines are the most frequent operations made to workflow files.

6.1.8 Temporal Patterns of Line-based Workflow Changes

We already observed in Section 6.1.6 that, while repositories are more likely to change their workflow files within the very first weeks after having adopted GitHub Actions, changes are nevertheless observed during the whole lifetime of these files (RQ 6.2.2). With this section, we aim to gain a better understanding of the type of changes that are made to lines of code in workflow files, hypothesizing that many lines of code will be added to the workflow files during their first weeks to add new functionalities until a stable point is reached, and then lines will be mostly modified for maintenance purposes.

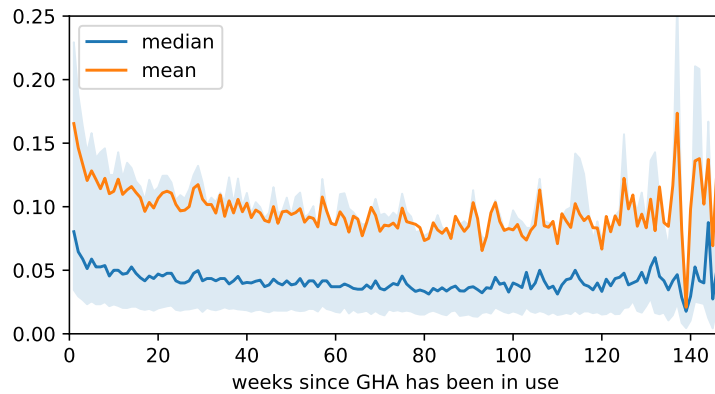


Figure 6.7: Proportion of lines of code *touched* during workflow changes. The shaded area corresponds to the interval between the 25th and 75th percentiles.

As a first step, we start by computing the proportion of lines of code that are *touched* (i.e., lines that are added, removed or modified) during workflow changes. Figure 6.7 shows the evolution of this proportion in function of the time elapsed since GitHub Actions was adapted by the corresponding repositories. The figure shows the median and mean values, as well as the 25th and 75th percentiles, represented by the shaded light blue area. As explained in Section 6.1.6, the higher variation on the rightmost part of the figure are due to the lower number of repositories that have been using GitHub Actions for more than 100 weeks.

We observe that both the median and mean values exhibit a gradual decrease during the first year. The higher mean relative to the median suggests a positively skewed (right-skewed) distribution, where extreme values in the upper tail pull the mean upward. Focusing on the mean value, the decrease in the number of lines of code touched is particularly visible in the first weeks,

going from an average of 13.6% during the first six weeks to an average of 10.2% for the next year. This indicates that more changes are applied to the lines of workflow files in the early phase of the workflows' lifetime.

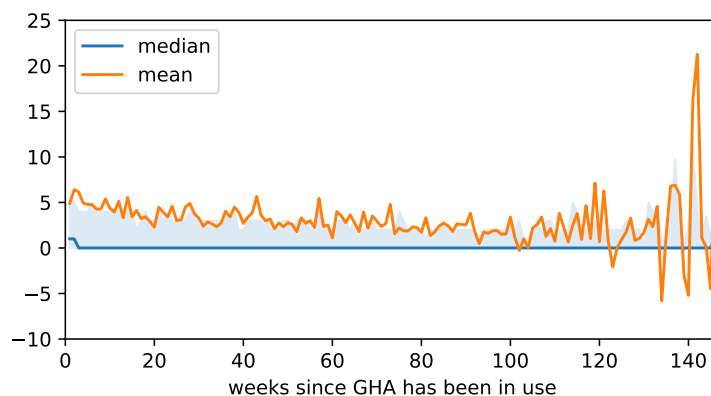


Figure 6.8: Net growth in lines of code (added minus removed) during workflow changes. The shaded area corresponds to the interval between the 25th and 75th percentiles.

Figure 6.8 shows the evolution of the distribution of net growth in workflow files, defined as the number of lines added minus the number of lines removed, as a function of the time elapsed since workflows were adopted by their corresponding repositories. The shaded area represents the interval between the 25th and 75th percentiles. As previously observed in Section 6.1.7, workflow changes often involve limited modifications, which explains why the median net growth remains close to zero throughout the workflow lifecycle. This indicates that many changes either balance additions and removals or result in only minor growth.

Focusing on the mean net growth, we observe a gradual reduction over time. In particular, the mean net growth decreases from an average of 5.3 lines during the first six weeks following adoption to approximately 3.5 lines over the subsequent year. This pattern suggests that early workflow evolution is characterized by larger structural expansions, followed by smaller and more incremental adjustments as workflows mature.

To better understand how the intensity of workflow edits evolves over time, we analyze the number of lines of code that are *modified* (i.e., changed but not added or removed) during each workflow update as a function of the time elapsed since workflow adoption. Figure 6.9 shows the evolution of the number of lines of code modified in workflow files. As for Figure 6.8, the difference

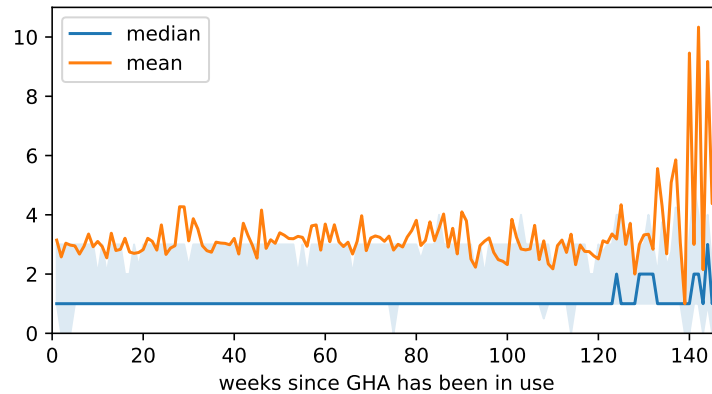


Figure 6.9: Number of lines of code *modified* during workflow changes.

between the median and mean values indicates a positively skewed distribution.

Focusing on the mean value, we observe that the number of lines of code modified is quite stable through time, only exhibiting a slight increase during the first weeks. For instance, the mean number of modified lines slightly increased from an average of 2.8 during the first six weeks to an average of 3.1 during the next year.

Takeaways. Looking at the temporal patterns of line-based changes (RQ 6.2.2), we observe that one out of ten lines of code are touched when a workflow file is modified, and this proportion is higher during the first weeks after the considered repositories started to adopt GitHub Actions. On average, each workflow file modification modifies 3.1 lines of code and produces a net addition (added minus removed) of 3.5 lines. This suggests that the contents of workflow files are subject to *continuing change* and *continuing growth*.

6.1.9 Threats to Validity

In following, we discuss the threats that may affect the validity of our findings.

Internal validity relates to the extent to which the study results are influenced by the experimental treatment or condition being studied (Ampatzoglou *et al.*, 2019). We analyzed the evolution of workflows in software development repositories by studying weekly snapshots.

Although one can study workflows commit by commit, Bird *et al.* (2009) stated that git lacks a mainline and a file can change in parallel in different

branches, which makes it difficult to track the linear history of a file. Therefore, we chose to analyze snapshots instead of checking commits directly.

External validity concerns the generalisability of the results (Ampatzoglou *et al.*, 2019). We only consider public software repositories with more than 100 stars and commits which are still under development and maintenance. These criteria are used to find projects best suited for software evolution studies in the case of GitHub Actions CI/CD tool. However, we can not generalize these findings on other GitHub repositories, including personal webpages.

Construct validity concerns the relation between the theory behind the experiment and the observed findings (Ralph & Tempero, 2018). To detect the use of workflows in GitHub repositories, we identified the presence of a *YAML* file in the `.github/workflows` folder. This approach may lead to an overestimation of the presence and correctness of a *YAML* file. It can be due to problems in the corresponding workflow files or simply not being triggered for use. However, we believe that most of these workflows are indeed used since developers are unlikely to keep GitHub Actions workflows without using them or do not solve the issues in the workflow files.

Conclusion validity threats concern the degree to which reasonable conclusions have been derived from our analysis (Maxwell, 1992). As our results only report quantitative observations, they are not exposed to such threats.

6.2. A Differencing Tool for GitHub Actions Workflows

The analyses in Section 6.1 reveal that GitHub Actions workflows evolve through a diverse set of coarse-grained and line-based changes that occur at different points in a repository’s lifecycle. Accurately identifying and classifying these changes requires a differencing approach that is both workflow-aware and capable of operating at multiple levels of granularity. Existing generic diff tools are insufficient for this purpose, as they do not capture the semantic structure of workflow files nor distinguish between different types of workflow-specific modifications. To address this methodological need, Section 6.2 introduces gawd, a differencing tool specifically designed to support the fine-grained analysis of GitHub Actions workflow changes.

6.2.1 Introduction

Prior research has focused on changes in CI/CD pipelines based on Travis (Durieux *et al.*, 2019; Gallaba & McIntosh, 2020; Zampetti *et al.*, 2021;

Zampetti *et al.*, 2020). In a similar vein, researchers have started to explore changes in GitHub Actions workflows including our research in last section (Section 6.1) and Valenzuela-Toledo and Bergel (2022). However, such workflows still lack a deep understanding on the types of changes they exhibit, and the frequency of these changes.

The typical way of identifying changes made to code files during commits is through the use of the `git diff` tool² or one of its variants. When applied to GitHub workflow files, however, the use of such a tool is hindered by its lack of precision when identifying changes. More syntax-aware diff tools exist for analysing changes in source code files for specific programming languages, but we are not aware of any diff tool that takes the specific YAML-based syntax of GitHub Actions workflow files into account.

This is why we have developed `gawd`, an acronym for **GitHub Actions Workflow Differencing** tool. The tool implements a differencing algorithm that enables the comparison of syntactic workflow differences, thereby supporting in-depth analysis of GitHub Actions workflow file changes and how such workflows evolve over time.

6.2.2 Motivating Example

Just like any other software artifact, workflow files are susceptible to change during their lifetime. Neither GitHub’s default diff tool nor any of `git diff` algorithms is capable of correctly identifying the set of changes made to these files in the different commits touching them. This is illustrated in the example of Figure 6.10, depicting the visual diff of the changes made to a workflow file in some commit.³

Lines 13-14 (and the two preceding unnumbered lines in red) show a change of job id from `linux-x64` to `linux-x64-x` and a modification of the job’s name. Lines 35-36 and 37-38 suggest that a sixth and seventh step⁴ have been added to this job, but this is not what happened in reality. Instead, the step named ‘Build Windows x64 binary’ on lines 55-58 (which is reported by the diff tool as not being part of any change) has been removed from this job, and added into a new job with id `linux-x64-w` (lines 41-58). As a consequence, the two steps on lines 35-38 have changed their relative position from 7 and 8, respectively, to 6 and 7, respectively. It is clear that the diff tool misses out on a lot of syntactic changes by considering lines 35-54 as a single composite change of 20

²<https://git-scm.com/docs/git-diff>

³Original commit: <https://github.com/d99kris/nchat/commit/0438344525f6bb36f41c0a145dfe54fad5a3f2c2>

⁴We assume a zero-based indexing for steps.

```

- linux-x64:
-   name: 'Linux Intel - OpenOCD ${ github.event.inputs.version } build'
13+ linux-x64-x:
14+   name: 'Linux Intel X - OpenOCD ${ github.event.inputs.version } build'
15   runs-on: [self-hosted, linux, x64]
16   container:
17     image: ilegeul/ubuntu:amd64-18.04-xbb-v5.0.0
18   defaults: ...
19
20   steps:
21     - name: 'Environment'
22       run: ...
23     - name: 'Clean working area'
24       run: ...
25     - name: 'Checkout project'
26       uses: ...
27     - name: 'Install xpm'
28       run: ...
29     - name: 'Install project dependencies'
30       run: ...
31     - name: 'Build Linux x64 binary'
32       run: |
33         xpm install --config linux-x64
34         xpm run build --config linux-x64
35+     - name: 'Publish pre-release'
36+       uses: ...
37+     - name: 'Rename working area'
38+       run: ...
39+
40+
41+ linux-x64-w:
42+   name: 'Linux Intel W - OpenOCD ${ github.event.inputs.version } build'
43+   runs-on: [self-hosted, linux, x64]
44+   container:
45+     image: ilegeul/ubuntu:amd64-18.04-xbb-v5.0.0
46+   defaults: ...
47+
48+   steps:
49+     - name: 'Checkout project'
50+       uses: ...
51+     - name: 'Install xpm'
52+       run: ...
53+     - name: 'Install project dependencies'
54+       run: ...
55+     - name: 'Build Windows x64 binary'
56+       run: |
57+         xpm install --config win32-x64
58+         xpm run build --config win32-x64

```

Figure 6.10: Visual difference of the changes made to some GitHub Actions workflow during a commit to a GitHub repository.

consecutive lines being added, while in fact it should be considered as multiple, more primitive changes.

Listing 6.1 provides a condensed summary of what an improved syntax-aware diff tool should produce for this specific workflow file changeset. Line 1 corresponds to the change of job id and line 2 represents the change of the job's name. Line 3 reflects the fact that the step 'Build Windows x64 binary' that used to be at position 6 is now removed from the job (because it has been inserted as part of a newly added job on line 6, of which the full contents is

Listing 6.1: Expected output.

```
1 renamed jobs.linux-x64 to jobs.linux-x64-x
2 changed jobs.linux-x64.name from "Linux Intel ..." to"..."
3 removed jobs.linux-x64.steps[6]
4 moved jobs.linux-x64.steps[7] to jobs.linux-x64-x.steps[6]
5 moved jobs.linux-x64.steps[8] to jobs.linux-x64-x.steps[7]
6 added jobs.linux-x64-w with "..."
```

not shown for the sake of brevity). Lines 4 and 5 correspond to the resulting reordering of the next two steps, that change their relative position in the job because of the removal of the step on line 3.

6.2.3 GAWD

Installation and execution

gawd has been developed as a standalone Python 3 package with a command-line interface. gawd reports on the syntactic differences between two GitHub Actions workflow files. Given two workflow files, gawd identifies the addition or removal of steps and key-value pairs, changes in values associated to a given key, and moves of steps to a different position within an existing job. gawd is publicly available on PyPI and can be installed through pip (`pip install gawd`), and its source code is available on <https://github.com/sgl-umons/gawd>.

To compute the syntactic diff between two workflow YAML files `file1.yml` and `file2.yml`, one needs to run the command `gawd file1.yml file2.yml`

The tool can be configured with a range of optional arguments. For all details, we refer the reader to the tool's built-in help (*i.e.*, `gawd --help`). For example, the following command outputs the result in JSON format (`--json`) using a more condensed form (`--short`) by not showing the full details of the values associated to each change, as values can sometimes be very long:

```
gawd file1.yml file2.yml --json --short
```

Other optional command-line arguments can be used to change the default behaviour of the tool: `threshold`, `position_weight` and `job_name_weight`.

`threshold` can be set between 0 and 1 (by default it is 0.5) to modify the sensitivity of the tool in identifying changes. Pairs whose distance score is at or below this threshold are classified as `changed`, `renamed`, or `moved`; pairs above it are treated as independent `additions` and `removals`.

`position_weight` specifies a value between 0 and 1 (by default it is 0.2) that determines the weight assigned to the relative positions of two items. Increas-

ing this value increases the importance of the relative position of two items in a sequence (*e.g.*, two steps in a job) while decreasing the influence of their content (dis)similarity.

job_name_weight specifies a value between 0 and 1 (by default it is 0.2) that quantifies the significance of differences in job names when determining the similarity of jobs. A higher value places greater importance on job name similarity and reduces the influence of their content (dis)similarity.

Implementation details

The implementation of *gawd* relies on a set of functions dedicated to finding matches, calculating distances and comparing differences.

Finding matches. Two functions *find_list_matches* and *find_job_matches* are dedicated to identifying and computing matches, whether they involve items within sequences or jobs. These functions assign distance scores to various item combinations, allowing for the generation of sorted lists of matches, while considering positions and job names. In this process, all potential matches are returned, ordered by distance, but each item is only matched once. Items whose distance is below the given threshold are categorized as *changed*. Other items are categorized as *removed* or *added* depending on their presence in the first (old) or second (new) file.

Concretely, *find_list_matches* generates all $M \times N$ possible pairs between lists a and b (of sizes M and N) using `itertools.product`. Each pair (a_i, b_j) is assigned a score that combines element similarity and positional difference: $(1 - w_p) \cdot \text{distance}(a_i, b_j) + w_p \cdot |i - j| / \max(M, N)$, with $w_p = 0.2$. The pairs are then sorted by this score, and matches are selected greedily while ensuring that each element from both lists is used at most once, tracked via sets of matched indices.

find_job_matches follows the same approach, but replaces the positional term with a name-similarity component: $(1 - w_n) \cdot \text{distance}(a, b) + w_n \cdot \text{distance}(i, j)$, where $w_n = 0.2$ and $\text{distance}(i, j)$ measures similarity between job names. Both functions have a time complexity of $O(MN \log(MN))$ due to the sorting step, and require $O(MN)$ space to store the candidate pairs.

Calculating distances. The *dict_distance* function computes a normalized distance between dictionaries, based on common, changed, added, and removed items. The *distance* function calculates distances between objects, accounting for data types such as simple values, dictionaries (by relying on *dict_distance*) and sequences. This comprehensive approach enables a nuanced understanding of similarity and difference of two data structures. Specifically, *distance* dispatches on type:

- It returns 0 for equal values and 1 for a `None` operand or mismatched types;
- For dictionaries it delegates to `dict_distance`, which partitions keys into common, added, and removed sets and returns their mean distance (added and removed keys each contribute 1);
- For lists it calls `find_list_matches` recursively;
- For strings it applies `difflib.SequenceMatcher` in $O(|s_1| \cdot |s_2|)$ time, returning $1 - \text{ratio}$.

The recursion depth is bounded by the nesting depth d of the YAML structure; in the worst case the total cost across the full recursive call tree is $O(N^2 \log N)$ where N is the number of leaf nodes in both files, with $O(N^2)$ auxiliary space for candidate lists and $O(d)$ stack space.

Comparing differences. Function `diff_workflows` takes two workflow files represented as dictionaries, performs a comparison, and generates a list of differences. These differences are categorized into additions, removals, renames, changes, and moves with associated paths and values from both the old and new workflows. Function `diff_workflow_files` operates on two workflow files specified by their file paths. It reads and converts these files into dictionaries and calls `diff_workflows` to provide a list of differences between the two workflow files. All detected changes are represented as 5-tuples (`kind`, `old_path`, `old_value`, `new_path`, `new_value`), where paths are lists of string keys and integer indices locating each change in the workflow tree.

`diff_workflows` applies specialised handling for the `on` trigger key by normalising string and list forms into a canonical dictionary prior to comparison, so that syntactically different but semantically equivalent trigger specifications are not reported as changes.

For jobs, `find_job_matches` is used instead of positional matching since jobs are run in parallel based on their names and content, and pairs whose names differ but whose content is sufficiently similar emit a `renamed` record in addition to any nested `changed` records. `diff_workflow_files` parses both YAML files via `ruamel.yaml` in safe mode before delegating to `diff_workflows`; parsing runs in $O(N)$ time proportional to file size N , after which the difference cost is dominated by the recursive distance computations described above.

6.2.4 Example

```

@@ -10,6 +10,9 @@ on:
10   paths:
11     - resources/Dockerfile
12     - rust-toolchain.toml
13 + env:
14 +   REGISTRY: ghcr.io
15 +   IMAGE_NAME: ${ github.repository }
16
17   jobs:
18     main:
@@ -28,12 +31,20 @@ jobs:
31     id: get-toolchain
32     run: echo "toolchain=`rustup show active-toolchain | cut -d ' ' -f1`" >>
33           $GITHUB_ENV
34 -   - name: Login to DockerHub
35 -     if: ${ github.repository == 'cloud-hypervisor/rust-hypervisor-firmware' &&
36 -       github.event_name == 'push' }}
37 +   - name: Login to ghcr
38 +     uses: docker/login-action@v2
39 +     with:
40 +       username: ${ secrets.DOCKERHUB_USERNAME }
41 +       password: ${ secrets.DOCKERHUB_TOKEN }
42 +     registry: ${ env.REGISTRY }
43 +     username: ${ github.actor }
44 +     password: ${ secrets.GITHUB_TOKEN }
45 +   - name: Extract metadata (tags, labels) for Docker
46 +     id: meta
47 +     uses: docker/metadata-action@v4
48 +     with:
49 +       images: ${ env.REGISTRY }/${ env.IMAGE_NAME }
50 +       flavor: |
51 +         latest=true
52 +   - name: Build
53 +     uses: docker/build-push-action@v3
@@ -44,8 +55,8 @@ jobs:
55     platforms: |
56       linux/arm64
57       linux/amd64
58 -     push: ${ github.repository == 'cloud-hypervisor/rust-hypervisor-firmware'
59 -       && github.event_name == 'push' }}
60 -     tags: rusthypervisorfirmware/dev:latest
61 +     push: ${ github.event_name == 'push' }}
62 +     tags: ${ steps.meta.outputs.tags }
63
64 -   - name: Image digest
65 -     run: echo ${ steps.docker_build.outputs.digest }

```

Figure 6.11: Real-world example of GitHub’s visual difference of workflow changes made in a commit.

Let us consider a concrete example of a difference between two versions of a workflow file associated with a specific commit.⁵ Figure 6.11 shows the difference as reported by GitHub. In contrast, Figure 6.12 illustrates gawd’s output, presenting the changes in a more fine-grained and syntax-aware manner. It highlights gawd’s proficiency in recognizing step movements (in terms of their relative position within a job) resulting from the addition or deletion

⁵<https://tinyurl.com/2hetxdue>

```

1 added env with {'REGISTRY': 'ghcr.io', 'IMAGE_NAME': '${ github.repository
  ↪ }}'
2 moved jobs.main.steps[6] to jobs.main.steps[7]
3 moved jobs.main.steps[5] to jobs.main.steps[6]
4 changed jobs.main.steps[5].with.push from "${ github.repository == ... }"
  ↪ to "${ github.event_name == 'push' }"
5 changed jobs.main.steps[5].with.tags from 'rusthypervisorfirmware/dev:latest'
  ↪ to '${ steps.meta.outputs.tags g}'
6 removed jobs.main.steps[4].if with "${ github.repository ==
  ↪ 'cloud-hypervisor/rust-hypervisor-firmware' ... }"
7 changed jobs.main.steps[4].name from 'Login to DockerHub' to 'Login to ghcr'
8 added jobs.main.steps[4].with.registry with '${ env.REGISTRY }'
9 changed jobs.main.steps[4].with.username from '${ secrets.DOCKERHUB_USERNAME
  ↪ }' to '${ github.actor }'
10 changed jobs.main.steps[4].with.password from '${ secrets.DOCKERHUB_TOKEN
  ↪ }' to '${ secrets.GITHUB_TOKEN }'
11 added jobs.main.steps[5] with {'name': 'Extract metadata (tags, labels) for
  ↪ Docker', 'id': 'meta', ... }

```

Figure 6.12: gawd output for the example of Figure 6.11.

(in this case addition) of steps preceding them in the updated workflow file. Additionally, the tool successfully captures changes in the values corresponding to each key within the workflow files. For each such change, the fully qualified path related to the key is reported. This capacity to trace changes to the exact location in the workflow’s hierarchical structure enhances the tool’s utility for software developers in effectively interpreting workflow file modifications.

Looking at Figure 6.12, gawd is navigating the hierarchical YAML structure of the workflow file and reporting changes at a specific, path-based granularity.

changed jobs.main.steps[4].name from ‘Login to DockerHub’ to ‘Login to ghcr’ (line 7, Figure 6.12): gawd identified that the fourth step (zero-indexed) within the main job had its name key’s value replaced. It matched the old and new versions of this step as the *same* step, rather than treating it as a removal and a new addition, because their overall content similarity fell below the configured distance threshold.

changed jobs.main.steps[4].with.username from ‘\${ secrets.DOCKERHUB_USERNAME }’ to ‘\${ github.actor }’ (line 9, Figure 6.12): Here, the username key existed in both versions of the step, but its associated value changed. gawd’s *distance* function compared the two string values using `difflib.SequenceMatcher` and, finding them sufficiently similar in structure (both are GitHub expression strings), reported this as a

change rather than a removal and addition.

It is worth noting that the exact output gawd produces depends on the configuration values used, in particular the *threshold* and *position_weight* parameters. Had the values of these parameters been set differently, the same commit could produce a different classification: for instance, a lower threshold might cause gawd to treat the old and new versions of step four as entirely unrelated, reporting the old one as *removed* and the new one as *added*, rather than recognizing them as a *changed* pair. The robustness of these matching decisions is something we examine further in the Section 6.2.5, where we manually verified gawd’s output against real-world commits to confirm that its default configuration yields meaningful results across a range of change scenarios.

6.2.5 Validation

We validated the correctness of gawd by manually checking the tool’s output for real-life cases of workflow file changes made in existing commits in 40 different GitHub repositories. To do so, we first used the SEART GitHub search engine (Dabic *et al.*, 2021) to create a dataset comprising GitHub repositories that (i) are not forks of existing repositories; (ii) use GitHub Actions and contain at least one workflow file in the `.github/workflow` directory; (iii) have been created after 2019-01-01 and before 2019-06-01; (iv) have at least 100 stars and 300 commits; and (v) have their most recent commit after 2023-01-01. We randomly selected 40 of these repositories and locally cloned them on 2023-05-24.

For each of these 40 repositories, we extracted all commits involving changes to GitHub Actions workflow files and randomly selected one such commit. Using the 40 selected commits we manually checked whether gawd’s output corresponded to our own interpretation of the changes being made to the workflow. In those cases where the commit contained changes to multiple workflow files, we selected the first workflow file.

The manual confirmation of the correctness of the tool’s output proceeded as follows. Three distinct researchers discussed together about the interpretation of their changes, in order to come to a consensus on the actual changes that were being observed in the workflow files. These changes were then compared against gawd’s output. The outcome of this comparison revealed that gawd successfully identifies all code-related modifications within workflow files present in our test cases.

We have integrated this manual confirmation of gawd’s output on 40 cases of workflow file changes as an automated test suite that is part of gawd’s CI process whenever we are changing its implementation and creating new versions

and releases.

6.2.6 Limitations

A limitation of gawd is its inability to properly comprehend composite “semantic” changes such as merging (or, conversely, splitting) multiple steps into a single one, moving steps between jobs, replacing a step executing some shell commands (`run: key`) by a step using a reusable Action (`uses: key`) and vice versa. Similarly, gawd ignores changes to comments, or cosmetic changes such as the addition or removal of empty lines, white spaces or indentations that do not affect the workflow syntax.

In future endeavors, we plan to enhance gawd’s functionality by implementing a three-way differencing feature, facilitating improved merging of workflow files within repositories. A three-way difference is a comparison technique that considers three versions of a file simultaneously: a common ancestor (the *base* version), and two diverging descendants (typically referred to as *ours* and *theirs*). This is in contrast to the two-way difference that gawd currently implements, which only compares two versions of a workflow file at a time.

The motivation for this feature arises naturally from collaborative software development workflows. When two developers independently modify the same workflow file starting from the same base version, a three-way difference allows a merging tool to determine which changes originated from each developer, and whether those changes conflict with one another. Specifically, if a line or structural element was modified in *ours* but left untouched in *theirs* relative to the *base*, the change can be accepted automatically. Conversely, if both versions modified the same element differently, a conflict is flagged for manual resolution.

In the context of gawd, a three-way difference would operate at the syntactic level of GitHub Actions workflow files rather than at the line level. For instance, if one developer added a new step to a job while another developer modified the trigger events of the same workflow, a three-way gawd would correctly identify these as non-conflicting changes and merge them automatically. By contrast, if both developers independently modified the same step’s `with` parameters in incompatible ways, gawd would flag this as a structural conflict, providing a more semantically meaningful conflict report than a line-based tool like `git merge` would produce.

We also aim to provide a visualization to allow users to identify the reported changesets in a familiar way, as in traditional line-based diff tools. Furthermore, similar to the Unix `patch` tool, we plan to create a tool taking as input some changes generated by gawd and to apply them to similar workflow files.

We also plan to make gawd aware of the various programming languages that can be used in workflow files (*e.g.*, shell commands or Python scripts in `run:`) when computing the distance between values.

6.3. Summary and Conclusions

This chapter investigated the evolution of GitHub Actions workflows in collaborative software development by combining large-scale empirical analysis with tool development. We conducted a quantitative study of 22,733 GitHub repositories, encompassing more than 4 million weekly snapshots of workflow files, to examine how workflow automation artifacts change over time.

At a coarse-grained level, we observed that repositories created after the introduction of GitHub Actions tend to adopt it as their primary CI/CD tool within a few months. Among observed workflow changes, modifications were the most frequent, followed by additions, indicating continuous maintenance rather than one-off configuration. A substantial portion of these changes occurred within the first six weeks after adoption. These observations align with Lehman’s laws of continuing change and continuing growth (Lehman, 1996), supporting the view of CI/CD workflows as evolving software artifacts.

At a fine-grained level, we found that almost 9 out of 10 of all workflow changes involved line-level edits, with such edits appearing in the overwhelming majority of all workflow history versions. Line additions were concentrated early in the workflow’s lifetime, while modifications increases throughout its evolution. This pattern of modifications indicates that workflows are continuously adapted to changing project requirements, dependencies, and integration contexts, further reinforcing their characterization as long-lived, maintainable artifacts rather than static configuration files.

To support systematic study of workflow evolution, we introduced gawd (GitHub Actions Workflow Differ), a syntax-aware differencing tool tailored to GitHub Actions workflows. Implemented as an open-source Python library with a command-line interface, gawd enables structured detection of additions, deletions, modifications, and code movement across workflow versions. Manual validation on 40 real-world commits confirmed its accuracy across a wide range of change scenarios.

Taken together, the empirical findings and the supporting tooling directly address the third thesis goal by establishing an empirical understanding of workflow automation as a long-lived software artifact. The results uncover recurring temporal patterns in workflow adoption, growth, and maintenance, as well as common usage conventions reflected in early configuration bursts

followed by sustained incremental modification.

In the next chapter, we plan to use gawd and our understanding of workflow evolution in large-scale studies to explore changes in workflow configuration, execution behavior, quality attributes, and security practices.

Evolution of GitHub Actions Workflows

“Problems are only opportunities with thorns on them.”

Hugh Miller

As established in the previous chapter, GitHub Actions workflows are not static configuration artifacts but rather evolving software entities that undergo continuous modification throughout their lifetime. While Chapter 6 provided a broad view of workflow evolution, it left open an important question: which workflow entities are most prone to change, and what types of changes occur most frequently?

Understanding which parts of a workflow file require the most attention during maintenance is practically relevant: it can guide tool builders toward better automated support, help developers follow best practices, and inform researchers about the structure of CI/CD maintenance work. Yet, to our knowledge, no large-scale study has specifically examined the types of syntactic changes made to GitHub Actions workflow files.

This chapter addresses that gap through a mixed-methods empirical analysis of 267K+ workflow histories from 49K+ public GitHub repositories, spanning over six years of change data. Using the custom-built differencing tool *gawd* for fine-grained, entity-level comparisons, the analysis identifies which workflow entities are changed most frequently and which tend to remain stable after initial setup.

The chapter makes three main contributions: a qualitative catalog of seven types of conceptual changes derived from the manual analysis of 439 workflow modifications; a large-scale quantitative characterization of over 7.8 million individual changes across workflow entities; and a before–after analysis examining whether the introduction of LLM-based coding tools or other technological changes related to GitHub Actions coincide with measurable changes in workflow maintenance behavior. The findings reveal that workflow evolution is dominated by incremental modifications concentrated in a small number of workflow entities, primarily within jobs and steps, and that automated dependency update tools play a significant role in driving these changes.

7.1. Introduction

Like many other CI/CD tools, GitHub Actions requires developers to define workflows in configuration files to automate a repository’s CI/CD pipeline. Following the CaC practice, these configuration files are stored in a human-readable YAML format within the repository. As such, workflow files are subject to version-controlled changes throughout the repository’s lifetime. These changes are driven by evolving software project requirements, technological advancements, and the shifting needs of developers as it was seen in Section 6.1.

Previous studies have sought to understand changes in workflow files for different CI/CD tools (Durieux *et al.*, 2019; Gallaba & McIntosh, 2020; Rostami Mazrae *et al.*, 2023; Valenzuela-Toledo & Bergel, 2022; Zampetti *et al.*, 2021; Zampetti *et al.*, 2020), with an important focus on Travis CI, as it was the dominant CI/CD tool on GitHub before GitHub Actions’s introduction (Golzadeh *et al.*, 2021c). We are not aware of any large-scale study specifically focused on the types of syntactic changes made to GitHub Actions workflow files. The analysis in this paper aims to close this gap, and its results could be used to suggest various improvements to maintaining GitHub Actions workflows, such as following best practices, increasing the awareness and take-up of existing tools (such as Dependabot for managing workflow dependencies), improving automated tool support (*e.g.*, for debugging, testing and refactoring workflows). The results can also instruct other researchers to conduct future work in this domain.

We carry out a mixed-method empirical analysis, formulated around five research questions:

RQ 7.1 *How frequently are workflow files changed?*

RQ 7.2 *Which conceptual changes are made to workflows?*

RQ 7.3 *What types of changes are made to workflows?*

RQ 7.4 *What is the impact of AI on workflow evolution?*

RQ 7.5 *Which syntactic entities are frequently changed in workflows?*

In general, this chapter has three main contributions: (i) we provide catalog of seven types of conceptual changes in GitHub Actions workflow files based on a manual *qualitative* analysis of 439 workflow file modifications; (ii) we report on our finding of a large-scale quantitative analysis over 267K+ workflow change histories from workflow files across 49K+ public GitHub repositories cover-

ing 3.4M+ workflow file versions from November 2019 to August 2025; (iii) we relate our findings to previous work in the context of CI/CD and GitHub Actions, highlighting observed changes, assessing potential improvements, and suggesting how to further advance research and practice in the GitHub workflow ecosystem.

7.2. Data Extraction

To conduct a large-scale empirical analysis of workflow changes over time, we need a large collection of GitHub Actions *workflow histories*. To do so, we rely on the 2025-10-09¹ version of a dataset obtained from public GitHub repositories that were both popular and active at the time of data collection, meaning repositories with a star count above a certain threshold, a high number of commits, and recent activity Cardoen *et al.*, 2024. The dataset contains 267,955 workflow histories obtained from 49,258 GitHub repositories, accounting for 3,418,911 workflow file snapshots. It includes all the commits that were made to these workflows, from their introduction in the repositories to their removal (if any) or up to the data collection date of 25 August 2025. Among others, the dataset provides, for each workflow history, a unique `uid` to keep track of renamed workflow files and, for each commit, its `date`, the `name` of the workflow file before and after the commit, and a `hash` value of the file contents before and after the commit. Since the dataset also contains the contents of all workflow files, this hash value can be used to compare the file contents before and after each commit.

We apply two additional filters to this dataset, motivated by the need to obtain valid workflow histories and ensure consistent temporal coverage:

1. **Ensuring valid workflow histories.** 776,339 of the 3,418,911 workflow file snapshots are flagged as invalid YAML files. These cases are problematic since they cannot be parsed in order to detect the changes made to them. Removing only these invalid files does not suffice as it would lead to incomplete workflow histories. Consider for example a workflow history of consecutive workflow files A, B and C, where B is not a valid YAML file. Removing B from this workflow history would erroneously aggregate the changes from A to B and from B to C into a single set of changes from A to C. To avoid such problems, we excluded from the dataset the 30,922 workflow histories containing an invalid YAML file.

¹<https://zenodo.org/records/17301952>

- Ensuring consistent temporal coverage.** Since the analysis is based on weekly observations, we restrict the observation period to complete weeks only, from Sundays to Saturdays. We excluded the last two days from the dataset so that the observation period covers 316 complete weeks, starting on Sunday 4 August 2019 and ending on Saturday 23 August 2025.

After these steps, the final dataset comprises 236,775 workflow histories from 47,488 repositories, accounting for 2,640,584 workflow file snapshots (abbreviated to workflow files in the remainder).

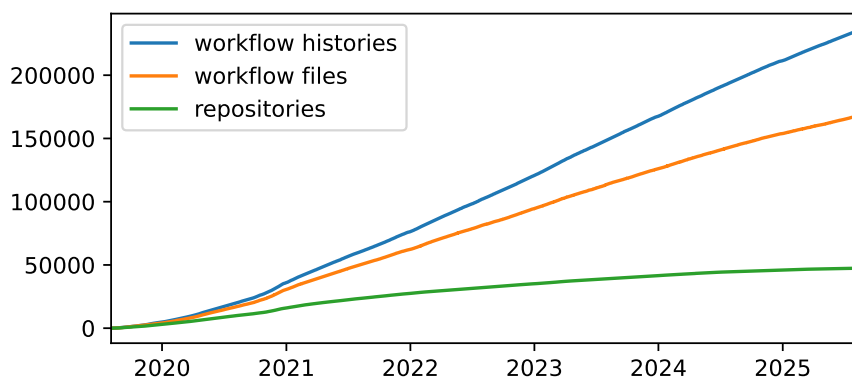


Figure 7.1: Evolution of the number of GitHub repositories, workflow files, and workflow histories in the dataset.

Figure 7.1 shows the evolution of the number of GitHub repositories, workflow files and workflow histories in the dataset. One can observe that the number of repositories using GitHub Actions is increasing through time, and the number of workflow files and workflow histories are growing at a faster pace, indicating that more and more workflows are being added to the repositories. The slight variations that can be observed in late 2020 coincides with restrictions imposed by Travis (a competing CI/CD service) on its free plan, leading many repositories to migrate from Travis to GitHub Actions (Golzadeh *et al.*, 2021c; Rostami Mazrae *et al.*, 2023). The data and code produced to replicate the analysis are available on Zenodo.²

²<https://doi.org/10.5281/zenodo.18414913>

7.3. Quantifying Workflow Change Frequency

The first section aims to quantify to which extent workflow files are subject to changes during their lifetime (RQ 7.1). To do so, we consider four different change types, namely *addition* of a new workflow file to the repository, *removal* of the workflow file from the repository, *modification* of the workflow file contents w.r.t. its predecessor in its workflow history, and *renaming* the workflow file w.r.t. the previous one in its workflow history. For each workflow file in the dataset, we compute its change type w.r.t. its predecessor in its workflow history.

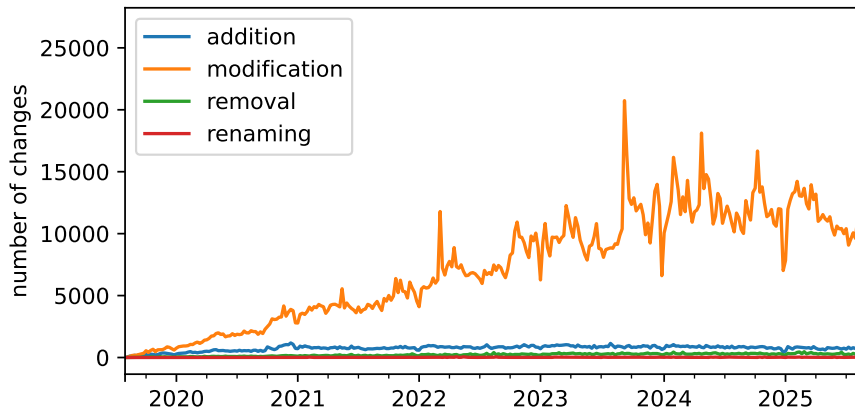


Figure 7.2: Weekly number of workflow file changes, per change type.

Overall, we found 2,330,529 modifications, 236,775 additions, 68,083 removals, and 29,159 file renamings. Figure 7.2 breaks this down into the weekly number of observed workflow file changes. The overwhelming majority of workflow file changes are modifications, more than a fivefold of all other change type occurrences combined. The number of modifications tends to increase through time, which is probably a consequence of the fact that more and more workflows are added through time (as observed in Figure 7.1). Additions and removals tend to remain quite stable through time, which is a consequence of the fact that a workflow can only be added or removed once throughout its history. We found that 24.9% of the repositories have more than one workflow file added to them, with a median of three workflow files per repository.

Looking more closely at the changes through time in Figure 7.2, we observe some peaks and troughs at specific times, regardless of the change type. The troughs coincide with the end of year holidays, during which developers are

less likely to work and change their workflows. The peaks will be examined in more detail as part of Section 7.5 and Section 7.7 that delve deeper into the changes to the workflow contents and to the values of workflow entities.

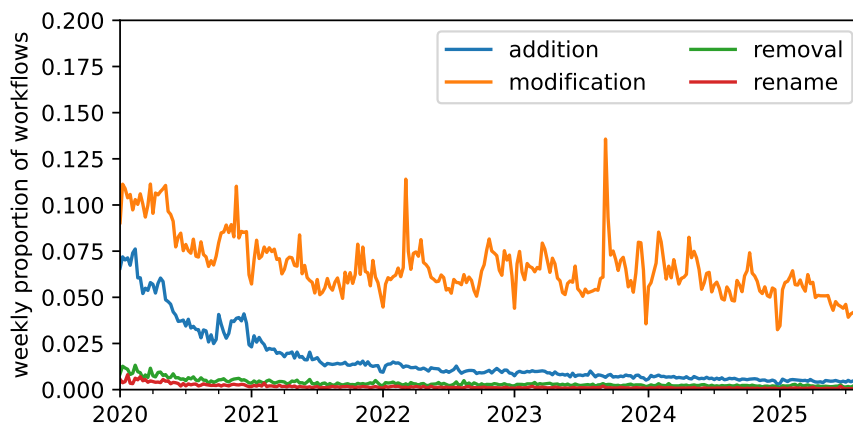


Figure 7.3: Weekly proportion of workflow files exhibiting a change.

To determine whether the observed increase in modifications is mainly due to more workflow files being added over time, or rather to workflow files being modified more frequently, we repeated the previous analysis by computing the weekly proportion of workflow files exhibiting a change of a given type. In Figure 7.3, the “weekly proportion” is calculated as the number of workflow files that experienced a specific type of event (addition, modification, removal, or renaming) during a given week, divided by the total number of workflow files that existed in that week. This normalization allows to assess the relative likelihood of workflows undergoing a change of a given type, independent of the overall growth in the number of workflows. On average, per week 7.3% of the workflow files are modified, 2.1% are added, 0.4% are removed, and 0.2% are renamed.

In this section, we examined the average change rate of workflow files. To do so, we identified all commits touching a particular workflow file, and we divided the workflow’s lifespan by the number of times it was touched. We observed that, on average, workflow files are updated every 159 days, with 25% (Q1) of workflow files being touched every 31 days, and 50% (median) every 82 days.

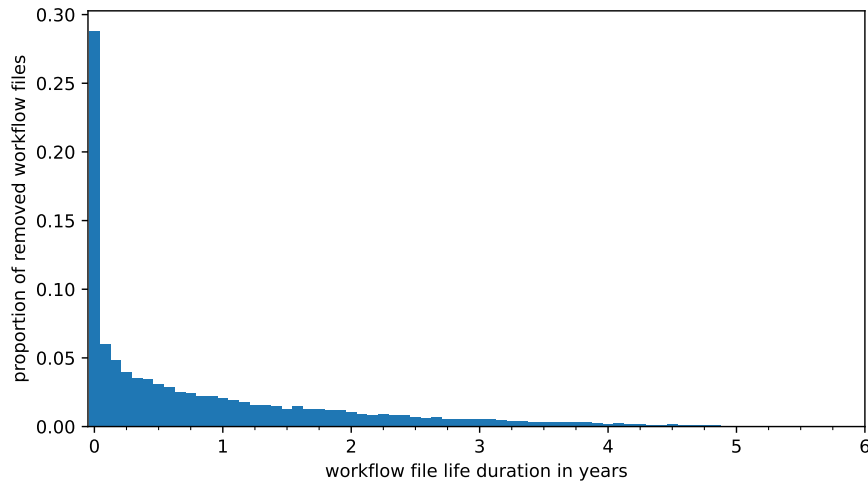


Figure 7.4: Proportion of removed workflow files with respect to their lifespan.

Takeaways. As the answer to RQ 7.1, around 25% of all repositories add more than one workflow file throughout their lifetime, with a median of three workflow files per repository. Workflow files are updated on average every 159 days, with 7.3% of workflow files being modified each week. While workflow files can be added, modified, renamed or removed, modifications clearly dominate, whereas renaming and removal are uncommon.

Short-lived workflow files While the overall frequency of workflow file removals is low, examining when these removals occur provides additional insight into workflow maintenance behavior. In particular, we aimed to understand whether workflow files that are removed tend to be short-lived or long-standing configurations files. To this end, Figure 7.4 shows the distribution of removed workflow files relative to their lifespan (*i.e.*, the duration between the commit that introduced the workflow file and the commit that removed it). We grouped these durations into monthly intervals and computed their relative proportions (*i.e.*, normalized by the total number of removed workflow files). The figure allows to examine not just the absolute number of removals, but their relative prevalence across different lifespan ranges. One can observe that a substantial fraction of workflow files are removed shortly after their creation. Specifically, 15.2% of all removed workflow files were deleted within the first day, 20.6% within the first week, and 29.0% within the first month. In total, 68,083 work-

Table 7.1: Workflow burst statistics under varying commit time intervals, showing prevalence, burst count, and commit density.

time interval	# workflow histories with bursts	% workflow histories with bursts	mean #bursts per workflow	mean #commits in bursts
15 mins	20,277	15.4%	1.52	3.90
30 mins	25,488	19.3%	1.57	4.05
60 mins	29,932	22.7%	1.59	4.12

flow files were removed at some point in their lifetime, a significant fraction of which were removed soon after their creation. Given the short lifespan and limited impact of short-lived workflows on long-term workflow maintenance efforts, we recommend that future longitudinal studies aiming to characterize sustained maintenance practices consider excluding such ephemeral workflow files. However, we also recognize the value of studying short-lived workflows in their own right, for example, to understand exploration, onboarding patterns, or rapid prototyping activities on GitHub software projects and their associated workflows.

Takeaways. Removing workflow files is uncommon, and such removals tend to occur shortly after the workflows' creation.

Commit bursts While we have showed that workflow files are frequently modified throughout their lifetime, these aggregate statistics do not reveal *how* such changes are distributed over time. To better understand whether workflow maintenance occurs as steady, continuous activity or rather as short, intense editing sessions, we examined the temporal clustering of commits touching the same workflow file.

Nagappan *et al.* (2010) studied commit “bursts” in software components as defect predictors. We investigate whether similar short bursts of rapid commits also occur for workflow files, signaling trial-and-error debugging attempts to fix failing workflows. To do so, we apply a temporal clustering algorithm on the dataset to identify workflow burst episodes. We define a burst as a chain of at least three (not necessarily consecutive) commits belonging to the same workflow history, in which each commit occurs within a fixed short time interval of the previous one. Table 7.1 reports the results using three thresholds for this time interval: 15, 30, and 60 minutes. For the 60-minute interval we

observed bursts in 22.68% (29,932) of all workflow histories, with a relative commit density of 4.13 commits on average per burst (median of 3). For the shortest interval of 15 minutes, there were still 15.37% (20,277) of all workflow histories with burst activity. Across all three time intervals, the mean number of bursts per workflow history remained around 1.55, with each burst typically comprising 3 (50th percentile) to 4 commits (75th percentile). We also observed a non-negligible number of longer and more intense bursts. For example, for the 15, 30, and 60 minute intervals, we identified respectively 1,273, 2,120, and 2,818 bursts containing at least 8 commits. Some more extreme cases involve up to 43 commits³ for the 15-minute interval and up to 50 commits⁴ for the 60-minute interval. These findings confirm that bursty, high-frequency workflow commits are not uncommon. A substantial number of workflow histories, up to one in ten, show signs of clustered change sequences that may reflect iterative testing and debugging.

Taken together, this temporal analysis supports our hypothesis that workflows are frequently changed through short, iterative editing sessions, likely to fix bugs in response to failing workflows. Further analysis of workflow execution logs would be necessary to fully validate this trial-and-error behavior, but is considered out of scope of the current paper.

Takeaways. Bursts of rapid successions of workflow changes are not uncommon, suggesting the need for improved debugging and testing tools.

7.4. Classification of workflow changes

Section 7.3 revealed that a significant proportion of workflow files undergo frequent modifications throughout their lifetime. In this section, we aim to qualitatively analyse the nature and location of the changes made to the workflow contents stored in these files (RQ 7.2). This will allow to assess whether specific kinds of changes are more frequent than others, thus providing the basis for a large-scale quantitative analysis in the next sections.

We conducted a manual qualitative investigation of the changes made to the contents of a small yet statistically representative sample of workflow files. Four researchers were actively involved in this process, with the author of

³First commit of the burst: <https://github.com/FreeTubeApp/FreeTube/commit/1c9959ccfc020475caf1e632c30149cb30cd9b78>

⁴First commit of the burst: <https://github.com/iptv-org/iptv/commit/21c58cccae1a4c570e40e271cdd88054741bdd57>

this dissertation leading the design, coordination, and execution of the main tasks. We followed the *framework method* (Spencer *et al.*, 2004) to identify and categorize conceptual changes to GitHub Actions workflows.

We began by *familiarizing* ourselves with the syntax and structure of workflows to understand the possible changes that can be made to them. During this stage of *framework identification*, we realized that most workflow file modifications could be mapped to changes of specific syntactic *entities* (such as triggers, environment variables, matrix strategies, jobs, and steps).

During a subsequent *indexing* phase, we qualitatively examined a statistically representative number of modified workflow files. We aimed for a 95% confidence level with a 5% margin of error (Burmeister & Aitken, 2012). This required randomly sampling 389 distinct workflow histories from our dataset (Section 7.2). Each workflow history was chosen from a different repository to ensure diversity. From each workflow history, we arbitrarily selected one workflow file for manual analysis. Its changes relative to its predecessor were obtained from the git commit information.

The sample of 389 cases was divided into three batches, each randomly assigned to two researchers for independent classification. Each researcher identified all observed changes to the contents of the assigned workflow file, as well as their change type (*i.e.*, addition, modification, or removal), providing clarifying comments when necessary. At the end of this process, the two researchers compared and merged their lists of identified changes and resolved disagreements. In the few cases where consensus could not be reached, a third researcher intervened to come to an agreement. At the end of this step, for each workflow file, we have set of changes like ‘change an instruction in run: step’ or ‘add job permission’.

To ensure saturation, we continued the above process by iteratively analysing ten additional workflow file modifications at a time. After processing 50 more cases in this way, we observed that the last batch did not yield noteworthy new information, concluding that saturation was reached. Considering all 439 analysed cases in total (389 + 50), we reduced the margin of error to 4.67% while keeping the 95% confidence level.

We then *charted* the documented changes by systematically mapping all changed workflow *entities* into a provisional classification scheme of change types (addition, modification, removal). For instance, we mapped the observed workflow modification ‘change an instruction in run: step’ to a ‘modification’ at ‘step’ level for a ‘command’. Similarly, we mapped the observed workflow modification ‘add job permission’ to an ‘addition’ at ‘job’ level for ‘permissions’. This charting allowed us to refine and consolidate the initial categories

Table 7.2: Conceptual changes being made to workflows, together with the number and proportion of occurrences observed in the considered sample of 439 modified workflow files.

concept	all changes		change type		
	#	%	modification	addition	removal
Task Specification (TS)	441	39.8%	196	145	100
Task Configuration (TC)	265	23.9%	226	21	18
Documentation (D)	201	18.1%	64	87	50
Runtime Configuration (RC)	85	7.6%	50	26	9
Execution Triggers (ET)	46	4.2%	23	19	4
Execution Rules (ER)	43	3.9%	14	16	13
Formatting (F)	28	2.5%	19	2	7
total	1,109	100%	53.4%	28.5%	18.1%

that had emerged during the indexing phase.

The final step consisted of *interpretation*, *i.e.*, abstracting the concrete change types towards higher-level conceptual categories. We employed an open card-sorting approach, allowing broader concepts to emerge from the classification of the charting phase, without imposing any pre-defined concepts. The coding during this step was performed by the author of this thesis, followed by a review and discussion with the other researchers until a negotiated agreement was reached (Garrison *et al.*, 2006).

The outcome of this step was a set of seven concepts of identified workflow changes. Table 7.2 reports on each of these concepts, providing the number of changes overall and per change type (*i.e.*, modification, addition, and removal) in the considered sample of 439 cases. In total, 1,109 change types were observed, each associated with one specific concept. 53.4% of these changes were *modifications*, 28.5% were *additions*, and 18.1% were *removals*. Below we discuss each of the seven concepts in detail. The first three concepts constitute the large majority (81.8%) of all observed changes:

Task Specification (TS) is the concept referring to the structure and logic that defines what tasks are performed in the workflow, and how they are executed. Entities belonging to this concept are changed the most frequently (39.8% of all observed changes). They also correspond to the highest number of additions and removals. Changes belonging to this concept include the introduction, removal and reordering of jobs and steps, as well as changing the `uses:` and `run:` specifications of steps. Another frequent category of changes is adding or modifying a strategy matrix (`matrix:`, `strategy:`, `include:` and `exclude:` keywords) to allow jobs to run multiple times based on the combination of

variables used in the matrix definition. Moreover, changes of the command of the steps for `run:` were also observed frequently.

Task Configuration (TC) is the concept covering the detailed settings of how individual steps are configured and executed within a workflow job. Entities belonging to this concept are changed quite frequently (23.9% of all observed changes) and correspond to the highest number of modifications observed during the qualitative analysis. Most of these modifications relate to the configuration of reusable Actions such as changing their versions (`uses:` keyword) and their parameters. Other entities that belong to this concept are defining working directories (`working-directory:`), declaring and using jobs and step identifiers (`id:`), and managing outputs of steps to be used by other jobs (`outputs:`).

Documentation (D) is the concept referring to the practice of providing descriptive information that explains or clarifies various parts of the workflow. Such documentation increases the readability and understandability of the purpose, structure, and behavior of the workflow. This can be achieved by providing human-readable names (`name:`) for workflows, jobs, or steps, or by including comments in the YAML file. Entities belonging to this concept are changed quite frequently (18.1% of all observed changes). Most of the changes are related to adding or modifying names in different parts of the workflow file.

The next four concepts are considerably less subject to changes, accounting for only 18.2% of all observed changes.

Runtime Configuration (RC) refers to the settings that control *how* and *where* the workflow needs to run. Proper runtime configuration ensures that workflows are efficient, secure, and adaptable to different project needs or environments. This includes defining the environment in which jobs will operate, ensuring it behaves as intended when it is triggered (`env:`), setting global defaults for jobs (`defaults:`), specifying the operating system or platform (`run-on:`), and managing the security permissions required for the workflow to perform specific actions (`permissions:`). These settings are crucial for ensuring consistency across environments, and controlling the access the workflow has during execution. Despite their importance, entities belonging to this concept are changed less frequently (7.6% of all observed changes).

Execution Triggers (ET) refer to the mechanisms that determine *when* a workflow is activated and executed. These triggers define the specific events or conditions that prompt the workflow to run (`on:`), allowing for automated responses to various activities within a repository. By configuring execution triggers, developers can specify the types of events, such as code pushes (`push:`), pull requests (`pull_request:`), or issue comments, that will initiate the workflow.

Additionally, these configurations may include details about the data associated with these events, enabling the workflow to respond appropriately based on the context of the trigger. Understanding and managing execution triggers is essential for creating efficient and responsive workflows that align with the development lifecycle and project requirements. However, once declared in the workflow, these entities are changed infrequently (4.2% of all observed changes), as they are typically set up at the beginning of the workflow development process and remain stable throughout the workflow's lifecycle.

Execution Rules (ER) establish the conditions under which workflow jobs are executed, ensuring that jobs behave predictably and efficiently. They enable workflow maintainers to control the flow of the workflow by specifying prerequisites that must be met before a job runs (*if:*), determining the sequence in which jobs are executed (*needs:*), and managing error handling strategies (*continue-on-error:*). Execution rules also allow for time constraints on jobs or steps, which can prevent indefinite runs and ensure that resources are managed effectively (*timeout-minutes:*). Only 3.9% of all observed changes were related to entities in this concept. The low frequency of changes can be attributed to the reduced need for such conditions in simpler scenarios.

Formatting (F) corresponds to cosmetic changes that do not affect a workflow's execution. This includes cases like changing the list representation of a workflow trigger, changes to delimiters, and quotation styles (*e.g.*, replacing single by double quotes). Changes in formatting were observed very infrequently (2.5% of all observed changes).

Takeaways. We qualitatively identified 1,109 individual changes across 439 commits that we classified into 7 categories (RQ 7.2), with modifications of workflow entities being the most common (53.4%), followed by additions (28.5%) and removals (18.1%). The large majority of workflow changes (81.8%) fall into three concepts: task specification (39.8%), task configuration (23.9%), and documentation (18.1%). These concepts primarily concern the contents of workflow steps, defining what to execute, how to configure it, and how to document it. This suggests that maintainers mainly adjust workflow code to refine or extend its functionality. Less frequent changes involve runtime configuration (7.6%), execution triggers (4.2%), execution rules (3.9%), and formatting (2.5%), which tend to be defined early in a workflow's lifecycle and remain stable over time.

The previous analysis focused on changes to individual workflow files. However, some GitHub repositories contain multiple workflow files that may be changed within the same commit. In our sample of 439 commits, 95 of them

(21.64%) touched multiple workflow files. We performed an in-depth analysis of those 95 multi-workflow commits to identify the nature of their changes. We found 23 cases of *unrelated changes* where no relation could be discerned between the changes to each workflow, 68 commits with *co-changes* where the same kind of change was made to each workflow, and 4 commits with *dependent changes* where a change to some workflow was the natural consequence of a different change to another. The commits with *dependent changes* involved moving a job or steps from one workflow to another, splitting a workflow into two, and refactoring jobs into a reusable workflow. 31 of the 68 commits with *co-changes* involved updating the version of reused components, either manually (13 commits) or automatically (18 commits) via tools such as Dependabot or Renovate. The 37 remaining *co-change* commits applied the same change to multiple workflow files (changing branch name, updating `run:` or updating the strategy matrix).

Takeaways. Most of the multi-workflow changes (68 out of 95) apply similar changes to multiple workflows. As a consequence, the quantitative analysis in the remaining sections will focus on changes to individual workflows only.

7.5. Qualitative analysis of fine-grained workflow change types

Section 7.4 reported on a manual **qualitative** classification of workflow changes into seven different concepts. Each concept encompasses multiple syntactic entities, corresponding to specific workflow keys. Section 7.5 shifts the focus toward a **quantitative** perspective aimed at analysing the number and type of changes made inside workflows (*i.e.*, addition, modification and removal), whereas Section 7.7 will focus on the frequency of changing specific syntactic entities. In Section 7.5 (RQ 7.3), we are particularly interested in understanding whether developers tend to create small changes (e.g., modifying a single step) or larger ones (e.g., modifying multiple steps, adding new steps, etc.). This distinction provides insights in how developers approach workflow maintenance, particularly whether their changes are focused on small refinements or involve more complex updates that may require significant restructuring.

Answering both Section 7.5 and Section 7.7 requires a tool to compute the changeset of syntactic differences between a modified workflow file and its immediate predecessor. To do so, we rely on gawd (GitHub Actions Workflow

Differ), a Python-based tool that identifies changes made to workflow files which was introduced in Section 6.2.

We executed gawd on all modified workflow file snapshots and their immediate predecessors in our dataset. This resulted in 2,261,806 changesets containing a total of 7,811,892 changes. The number of changesets is lower than the number of modified workflow files (2,330,529 cases, see Section 7.3), since gawd excludes changes that are irrelevant for our analysis, such as adding or removing whitespaces, comments, etc. The dataset of changes and changesets created in this process is available through our replication package.

The 7,811,892 changes can be broken down into 3,551,083 modifications, 1,513,588 additions, 887,961 removals, 1,811,068 moves, and 48,192 renames. We ignore the move and rename change types because they are of little relevance for our analysis and because gawd only provides a partial view of them. For instance, gawd only detects renames of jobs while moves are only detected inside the list of steps.

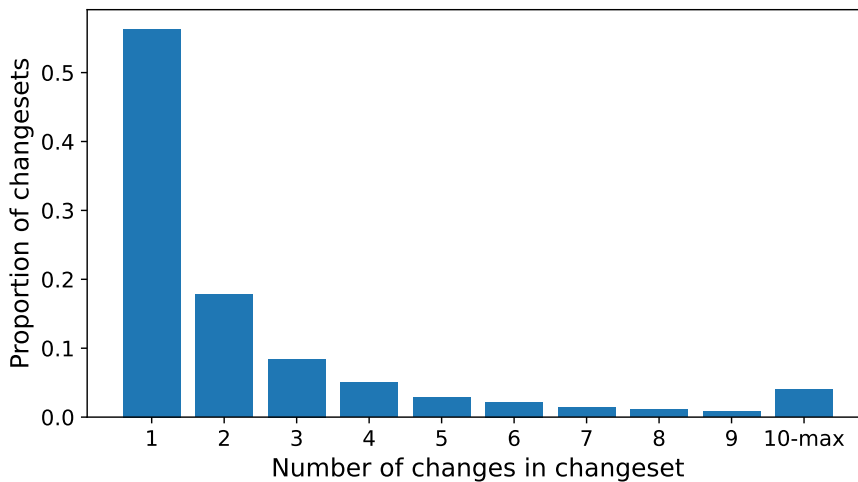


Figure 7.5: Histogram of the proportion of changesets in function of number of changes.

The number of changes (5,952,632 after ignoring renames and moves) is 2.67 times higher than the number of changesets (2,226,452) because many changesets contain more than a single change. Still, the median number of changes per changeset is one, implying that at least half of all changesets involve only a single change. Figure 7.5 visualises the distribution of changes per changeset. We observe that most changesets are small and focused, with

a steep decline in the frequency of larger changesets. We also observe a group of changesets that contain disproportionately many changes. Indeed, 10% of changesets include more than five changes, and nearly 4% contain ten or more changes, with a maximum of 1,596 changes observed in some changeset. Such outliers are typically the result of workflow files being modified by automated tools.⁵

Further looking at the change types in these changesets, we found that 78.98% of all the changesets contain only one change type, with **modifications** being the most common (78.81%), followed by **additions** (15.79%) and **removals** (5.40%). The remaining 21.02% of changesets involve multiple change types. The most frequent combination of change types is **addition and modification** (37.43% of multi-type changesets). Other common combinations include all three change types together (28.40%), **addition and removal** (20.10%), and **modification and removal** (14.05%).

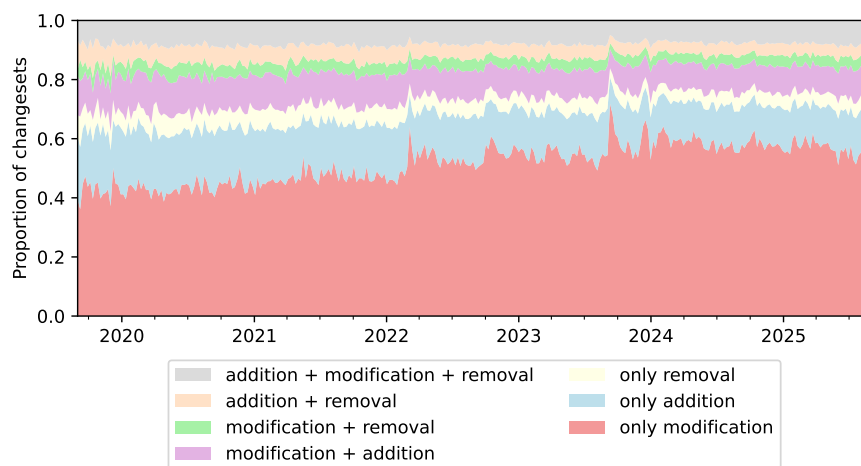


Figure 7.6: Weekly evolution of the proportion of changesets in function of the change types.

To analyze how these trends evolve over time, we examined the proportion of changesets containing at least one instance of each change type on a weekly basis. Figure 7.6 reveals a growing dominance of changesets consisting solely of **modifications**, indicating that as workflows mature, developers focus more on refining configurations than on adding or removing elements. At the end of the observation period, **modifications** account for 51.29% of monthly changesets on

⁵Example: <https://tinyurl.com/3xcjtude>

average, compared to 15.6% for additions and 4.89% for removals. Multi-type changesets continue to remain relevant as well. This suggests that, while most changes are focused on refining workflows, a significant portion still involves more complex changes combining multiple types, reflecting an ongoing balance between workflow maintenance activities and functional enhancements.

We also observe two relative spikes in modification activity, one in the first quarter of 2022 and another in the third quarter of 2023, aligned with major events in the GitHub Actions ecosystem. The 2022 spike corresponds to the migration from `node12` to `node16`, coinciding with the release of version 3 of several widely used Actions, including `checkout`,⁶ `setup-node`,⁷ `setup-python`,⁸ and `cache`.⁹ These Actions are among the most frequently used in workflows, as previously reported by Decan *et al.* (2022). Similarly, the spike in 2023 can be attributed to planned deprecation, such as the phase-out of Ubuntu 18.04 and the removal of Python 2.x support in version 3 of `setup-python`, which prompted widespread updates to existing workflows.

Takeaways. We analyzed 2,261,806 workflow file changesets, comprising a total of 7,811,892 individual changes obtained through gawd, composed of additions, removals and modifications of workflow entities (RQ 7.3). We observed that nearly four out of five changesets (78.98%) belong to a single change type, and in nearly four out of five cases (78.81%) these are modifications. The proportion of changesets consisting exclusively of modifications tends to grow over time, suggesting an increased focus on refining existing workflows.

7.6. Impact of AI and other technological changes on workflow evolution

Since 2021, LLM-powered coding tools such as GitHub Copilot (GitHub, 2021), ChatGPT (OpenAI, 2022), and Claude Code (Anthropic, 2023) have become increasingly integrated into software development practice. Prior work suggests that these tools may affect developer productivity, task allocation, and coding behaviour (Bird *et al.*, 2023). Their rapid adoption raises the question of whether they have also influenced the way CI/CD workflows are created and

⁶<https://github.com/actions/checkout/releases/tag/v3.0.0>

⁷<https://github.com/actions/setup-node/releases/tag/v3.0.0>

⁸<https://github.com/actions/setup-python/releases/tag/v3.0.0>

⁹<https://github.com/actions/cache/releases/tag/v3.0.0>

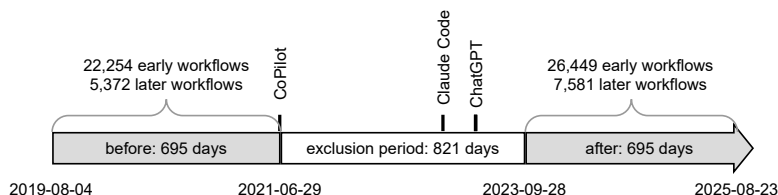


Figure 7.7: Comparison of early and later-phase workflow histories before and after introduction of LLM-powered coding tools and other major technological changes to GitHub Actions.

maintained over time. To explicitly address this question, we introduce as a dedicated before–after analysis. The goal of this section is to assess whether the emergence of LLM-based coding tools together with other major technological changes during the same period coincides with measurable differences in workflow creation and maintenance activity (RQ 7.4).

To this end, we perform a time-based comparison of workflow evolution before and after the introduction of LLM-powered coding tools. As shown in Figure 7.7, the dataset is divided into two observation periods of equal length (695 days). The *before* period captures GitHub Actions workflow usage prior to the availability of LLM-based coding tools, whereas the *after* period reflects workflow usage once such tools had become available and increasingly adopted. The two periods are separated by an exclusion window of 821 days, starting at the point when GitHub Copilot became part of the GitHub platform. This exclusion window accounts for the gradual diffusion of AI-based coding tools as well as other overlapping technological developments, thereby reducing contamination between the two observation windows.

Within each period, workflows are analysed at two distinct stages of their lifecycle in order to distinguish potential effects on workflow creation from those on longer-term maintenance. Specifically, we consider *early-phase* workflow histories, defined as the first three months of file changes following workflow creation, and *later-phase* workflow histories, defined as the three-month interval spanning months nine to twelve of the workflow’s lifetime, considering only workflows with at least one year of activity within the corresponding period.

The analysis considers a broad set of metrics capturing workflow evolution at multiple levels of granularity. At the coarsest level, we measure workflow-level change frequency as the number of commits touching a workflow file

Table 7.3: Descriptive statistics of workflow burst activity before and after exclusion period, stratified by workflow phase and burst time interval.

Burst interval	Workflow phase	Period	#workflow histories	Mean bursts per workflow	Mean commits per burst
15 mins	Early	Before	565	1.19	3.84
	Early	After	921	1.42	4.02
	Later	Before	89	1.11	3.47
	Later	After	165	1.10	3.53
30 mins	Early	Before	773	1.22	3.95
	Early	After	1,242	1.40	4.23
	Later	Before	123	1.12	3.57
	Later	After	213	1.15	3.63
60 mins	Early	Before	945	1.23	4.06
	Early	After	1,530	1.37	4.34
	Later	Before	147	1.12	3.62
	Later	After	265	1.16	3.69

during the considered three-month window (similar to reported findings on Section 7.3). At a finer granularity, we examine change frequency for individual workflow entity change types, namely modification, addition, removal, move, and renaming (similar to reported findings on Section 7.5). Finally, we analyse burst-based maintenance activity by considering both the number of bursts observed per workflow history and the number of commits per burst. Burst-related metrics are computed using three different commit time intervals (15, 30, and 60 minutes) in order to assess the robustness of the results to the burst definition (similar to reported findings on Section 7.3).

For each metric and workflow phase, we apply non-parametric Mann–Whitney U tests to assess whether the distributions observed in the before and after periods differ significantly. Overall, this results in 18 independent hypothesis tests: two at the workflow level, three related to burst characteristics per workflow, three examining commits per burst, and ten at the content level comparing before and after periods. To control the family-wise error rate induced by multiple comparisons, a Bonferroni correction is applied (Halperin *et al.*, 1988). Specifically, each reported p -value is multiplied by 18 (i.e., the total number of tests), and statistical significance is assessed against a Bonferroni-adjusted significance threshold of $\alpha = 0.01$. Whenever the null hypothesis (H_0) is rejected under this correction, the magnitude of the observed difference is quantified using Cliff’s δ effect size measure and interpreted following the guidelines of Romano *et al.* (2006).

Table 7.4: Before–after comparison of the number of bursts per workflow using the Mann–Whitney U test and Cliff’s δ , stratified by workflow phase and burst time interval.

Interval	Workflow phase	p -value	Adjusted p -value	Interpretation
15 mins	Early	1.44×10^{-5}	2.592×10^{-4}	Negligible (-0.091) H_0 not rejected
	Later	0.750	13.5	
30 mins	Early	8.76×10^{-4}	0.0158	H_0 not rejected
	Later	0.199	3.582	H_0 not rejected
60 mins	Early	0.00317	0.05706	H_0 not rejected
	Later	0.215	3.87	H_0 not rejected

Table 7.5: Before–after comparison of the number of commits per burst using the Mann–Whitney U test, stratified by workflow phase and burst time interval.

Interval	Workflow phase	p -value	Adjusted p -value	Interpretation
15 mins	Early	0.283	5.094	H_0 not rejected
	Later	0.091	1.638	H_0 not rejected
30 mins	Early	0.018	0.324	H_0 not rejected
	Later	0.133	2.394	H_0 not rejected
60 mins	Early	0.122	2.196	H_0 not rejected
	Later	0.106	1.908	H_0 not rejected

For the change frequency metric, H_0 is rejected for both early- and later-phase workflows (early: $p = 7.622 \times 10^{-117}$; later: $p = 8.485 \times 10^{-37}$), reflecting a decreased change frequency in the after period. However, the effect size is *negligible* for both groups (early: $\delta = -0.117$; later: $\delta = -0.125$, both $|\delta| < 0.147$).

A similar pattern is observed for burst-based maintenance activity. Descriptive statistics for burst-related metrics are reported in Table 7.3, while the results of the statistical comparisons are summarised in Table 7.4 and Table 7.5. H_0 is only rejected for the number of bursts per workflow when using a 15-minute burst interval for early-phase workflows. Even in this case, the effect size remains *negligible*. For later-phase workflows, and for the number of commits per burst under all burst intervals, no statistically significant differences are observed between the before and after periods.

At the level of workflow entity change types, the before–after comparison

Table 7.6: Before–after comparison of entity-level workflow change frequency using the Mann–Whitney U test and Cliff’s δ , stratified by workflow phase.

Change type	Workflow phase	p -value	Adjusted p -value	Interpretation
Modification	Early	3.819×10^{-33}	6.874×10^{-32}	Negligible ($\delta = 0.074$)
	Later	7.181×10^{-23}	1.293×10^{-21}	Negligible ($\delta = 0.116$)
Addition	Early	0.576	10.368	H_0 not rejected
	Later	0.555	9.99	H_0 not rejected
Removal	Early	0.914	18.452	H_0 not rejected
	Later	0.610	10.98	H_0 not rejected
Move	Early	0.542	9.756	H_0 not rejected
	Later	0.524	9.432	H_0 not rejected
Renaming	Early	0.943	16.974	H_0 not rejected
	Later	0.670	12.06	H_0 not rejected

reveals statistically significant differences only for modifications, for both early- and later-phase workflows, and for additions in early-phase workflows. These results are reported in Table 7.6. As with higher-level metrics, all corresponding effect sizes are *negligible*, reflecting small decreases in change frequency during the after period without substantive practical impact.

Taken together, the results of this section provide no conclusive evidence that the introduction of LLM-based coding tools, or other major technological changes occurring during the exclusion period, have substantially altered how GitHub Actions workflows are created or maintained. Across workflow-level, entity-level, and burst-based metrics, observed differences between the before and after periods are either statistically non-significant or associated with negligible effect sizes.

Takeaways. To answer RQ 7.4, we found no evidence that the introduction of LLM-based coding tools or other major technological changes have had a substantive impact on the creation or maintenance of GitHub Actions workflows.

7.7. Quantitative analysis of changes to workflow entities

Continuing the quantitative analysis of Section 7.5, Section 7.7 takes a finer-grained perspective by focusing on the syntactic changes made to workflows.

By doing so, we aim to uncover which workflow entities are most frequently changed (RQ 7.5). Understanding these syntactic hotspots can guide future improvements to the workflow language and its supporting tools. Frequently changed entities are prime candidates for enhanced tool support, such as automated bug detection, refactoring, and security audits, while infrequently changed entities may pinpoint low usage, limited usefulness, or lack of awareness, pointing to opportunities for clearer language documentation or redesign.

To identify the workflow entities that are most frequently changed, we rely on the workflow paths extracted using `gawd` (cf. Section 7.5). These paths represent the hierarchical structure of the modified workflow entities (e.g., `jobs.build.steps.3.if:always(:)`). To uncover common patterns across workflows, we normalize these paths as follows: (1) we replace user-defined identifiers (such as job names) with generic placeholders (e.g., replacing `jobs.build` by `jobs.<id>`); and (2) we mask the position of elements within arrays or lists (e.g., replacing `steps.3:` by `steps.<nr>`). For example, the path `jobs.build.steps.3.if:` is normalized to `jobs.<id>.steps.<nr>.if:`. These normalisations enables the abstraction of change paths that are syntactically different but semantically equivalent across workflows. This abstraction allows to group changes by their structural role within the workflow, such as jobs or steps, even when they differ in names or positions, enabling a better comparison across workflows.

Applying this normalization process to the changed workflows in our dataset, we obtain 198 unique normalized paths, reflecting all syntactic workflow entities that have been subject to changes. We organize these paths into a tree structure that mirrors the nested key organization of YAML syntax: each tree node represents an entity (e.g., `jobs:`, `steps:`, `permissions:`), and child nodes represent increasingly specific sub-entities. The root of the tree represents the conceptual structure of a complete workflow file, aggregated across all workflows. For each tree node, we compute the proportion of changes it accounts for by summing the counts from all corresponding subpaths. This hierarchical view enables a structured analysis of how changes are distributed across different workflow entities.

Table 7.7 presents the most frequently changed paths within this hierarchy. For clarity and interpretability, paths are grouped based on shared structural prefixes. Each table entry thus represents a category of syntactic entities sharing a common prefix (e.g., `jobs.<id>.steps.<nr>.if:`), capturing all changes affecting any sub-element within that subtree of the configuration. We only include the paths that cumulatively account for at least 0.2% of all observed changes.

For each path, we report the proportion of changes associated with it relative to the total number of changes in the dataset. This metric illustrates how frequently modifications occur at that specific location in the workflow structure. Additionally, the table indicates the most common change types (*i.e.*, Modification, Addition, or Removal) observed at each path. If a single change type constitutes more than 50% of the changes at that path, it is labeled as the primary change type. Otherwise, the two most frequent change types are reported. This breakdown reveals which parts of the workflow tree structure are more actively changed and in which way.

Change concepts. Below, we report on the most frequent path changes in workflows, grouped by the workflow *concepts* of Section 7.4 (cf. Table 7.2).

Task Configuration (TC). The most frequently changed paths belong to the workflow **jobs:** (91.05%), and more specifically their **steps:** (70.10%). Within steps, the most common changes involve the **run:** execution commands, and the **uses:** and **with:** entities that reference Actions and their parameters. Notably, nearly all observed changes to **run:** and **uses:** are *modifications* (over 99%). For **uses:**, these modifications correspond to Action version updates to maintain workflow stability by keeping dependencies up to date. Similarly, **with:** is frequently *modified* (64.8%), reflecting the need to adjust Action parameters as workflows evolve.

Task Specification (TS). Frequently changed paths belonging to this concept relate to the **strategy:** (9.89%), primarily for the **matrix:** and **include:** entities that tend to be changed through a mix of additions and modifications, whereas for **exclude:** removals are more common than modifications. This pattern indicates that while matrix configurations are frequently expanded or adjusted, exclusions are more often cleaned up or removed entirely. Changes to the **uses:** entity that involve replacing a referenced Action (rather than updating its version) are also classified under the concept of *Task Specification*, as they correspond to changing the workflow logic by altering what task is being executed rather than how it is configured.

Execution Triggers (ET). The most frequently changed workflow triggers are **push:** and **pull_request:**. They are used to initiate workflows in response to making a pull request or commit to the repository. For execution triggers, the primary change type is addition, followed by a smaller number of modifications.

Runtime Configuration (RC). Environment variables (**env:**) in workflows, jobs or steps are changed relatively frequently, mostly through addition or modification. The **permissions:**, which are used to give fine-grained control over what workflows and jobs can access, are changed less frequently and mostly involved adding more fine-grained permissions.

Table 7.7: Relative frequency of changed workflow paths, alongside their primary change type (Addition, Modification, Removal) and concept (using the shortcuts introduced in Table 7.2.)

entity	% changes	primary change types	concept
jobs/<id>	91.05	M (62.0%)	TS
steps[<nr>]	70.10	M (69.2%)	TS
uses	22.40	M (99.8%)	TC, TS
run	12.99	M (99.7%)	TS
with	12.65	M (64.8%)	TC
name	3.27	M (87.5%)	D
env	2.29	A (48.4%), R (26.7%)	RC
if	2.09	M (55.2%)	ER
id	0.34	M (43.2%), A (40.7%)	TC
working-directory	0.28	M (50.2%)	TC
strategy	9.89	A (37.6%), M (34.6%)	TS
matrix	9.52	A (36.3%), M (35.8%)	TS
include	2.88	M (48.3%), A (31.4%)	TS
exclude	0.47	A (44.7%), R (29.4%)	TS
runs-on	1.47	A (2.2%), M (95.9%)	RC
needs	1.08	A (46.9%), R (28.1%)	ER
env	1.05	A (41.5%), M (33.1%)	RC
if	0.95	M (47.8%), A (38.7%)	ER
name	0.79	M (79.9%)	D
with	0.72	M (50.3%)	TC
container	0.43	M (82.2%)	TS
uses	0.36	M (99.5%)	EC
permissions	0.35	A (81.6%)	RC
timeout-minutes	0.25	A (58.2%)	ER
outputs	0.24	A (54.9%)	TC
on	5.86	A (47.0%), M (28.1%)	ET
push	2.22	A (45.2%), R (29.2%)	ET
pull_request	1.62	A (49.6%), R (25.6%)	ET
schedule	0.54	M (73.8%)	ET
workflow_call	0.31	A (54.8%)	ET
env	1.48	M (47.9%), A (33.5%)	RC
name	0.81	M (96.8%)	D
permissions	0.45	A (81.6%)	RC
concurrency	0.29	A (68.7%)	ER

Documentation (D). Relatively frequent modifications are observed to the **name**: of workflows, jobs or steps, indicating that developers often change the names of these entities to maintain workflow readability.

Execution Rules (ER). The most frequently changed entity for this concept is **if**: (2.09% at step level and 0.95% at job level), used for the conditional

execution of jobs or steps. Its primary change type is modification, reflecting adjustments made to match evolving execution conditions.

Takeaways. The top ten modified workflow paths account for nearly 80% of all changes (RQ 7.5), mostly in the `jobs:`, particularly within `steps:`. The most frequently changed workflow entities (`run:`, `used:` and `with:`) map closely to the *Task Configuration* concept.

The remainder of this section dives deeper into the observed evolution of specific workflow entities, and discusses the impact of such changes on the security and reusability of workflows.

Usage of automated tools for workflow maintenance. Table 7.7 revealed that `uses:` is the most frequently modified workflow entity. The manual analysis of Section 7.4 revealed that many of these changes are in fact version updates of reusable Actions or other versioned workflow entities. We verified this on the full dataset of Section 7.7, observing a frequent use of automated dependency update tools, identified based on commit authors associated with bots. GitHub's built-in Dependabot was found to be used in 69.2% of all workflow histories, and the third-party tool Renovate in 21.0% of them. They are primarily updating Action versions in the `uses:` entity, accounting for 562,093 of the 587,624 modifications involving version updates.

We also dived deeper in the kind of Action version updates proposed by these tools. GitHub Actions supports four version formats: fully specified version tags (*e.g.*, `v5.0.1`), major-only version tags (*e.g.*, `v5`), commit hashes (*e.g.*, `8f4b7f8...`), and branches (*e.g.*, `main`). GitHub recommends commit hashes as the most secure option, since they allow for immutable version pinning. For version tags, GitHub recommends fully specified versions over major-only versions. Our workflow history analysis revealed an encouraging trend towards more secure version usage, with a decrease in version tag usage by 7,112 cases, and an increase in commit hashes by 7,235, nearly catching up in absolute numbers (239K *vs.* 242K). The version tags also shifted towards the more secure option of fully specified versions with an increase of 1,434 cases, compared to a decrease of 667 cases for major-only tags. Fully specified versions have now become as common as major-only tags (120K *vs.* 121K). These results highlight the wide adoption of dependency automation tools, contributing to more secure workflow practices.

Modifying workflow permissions. As another way to increase workflow security, GitHub Actions provides the `permissions:` mechanism to give fine-grained control over what workflows and jobs can access. Introduced in April

2021, they allow to enforce the principle of least privilege. While changes to `permissions:` represent a small fraction of the observed workflow changes (cf. Table 7.7), we observed a notable increase in the number of additions of permissions at workflow level (26,410) as well as at job level (25,708). The most frequently added permission type at workflow level was `contents:read`, whereas a wider variety of permissions were observed at job level (*e.g.*, `contents:write`, `id-token:write`, `pull-requests:write`). These additive permission changes suggest that workflows start with default permissions that get refined gradually as the workflow evolves.

Tradeoff between control and reusability. The analysis in this section revealed that the most frequently changed workflow entities are `steps:`, mostly within the `uses:` and `run:` specifications. During the manual analysis of Section 7.4 we observed multiple replacements of `run:` by `uses:` or vice versa. Analysing the full dataset of Section 7.7, we observed 9,827 cases of replacing `run:` by `uses:`, reflecting a shift toward exploiting reusable Actions. Doing so promotes workflow modularity and maintainability, but can come with increased security risks (Onsori Delicheh *et al.*, 2024). We also observed 7,230 cases of replacing `uses:` by `run:`, reflecting a preference for using custom scripts to enable more fine-grained control, performance optimisation, and avoidance of untrusted dependencies.

Takeaways. An analysis of the syntactic change patterns in workflows reveals that: (i) automated tools play a major role in maintaining workflows, especially to keep versions up to date; (ii) there is a trend towards more precise version pinning, enhancing workflow reliability and reproducibility. (iii) workflows tend to improve upon security best practices (such as respecting the principle of least privilege and the use of commit hashes for Action versioning) even though there is still significant room for improvement; (iv) workflow maintainers make trade-offs between modularity (though the use of reusable Actions) and fine-grained control (through the reliance on custom scripts).

7.8. Discussion

In this section, we compare our findings with existing research on the evolution and maintenance of CI/CD configuration files. Each subsection corresponds to one or more of this chapter's sections, providing a focused discussion of how our results align with, extend, or differ from prior work. Section 7.8.1

discusses workflow file evolution patterns (Section 7.3), Section 7.8.2 addresses conceptual changes observed in workflows (Section 7.4), Section 7.8.3 examines the changes on the content of workflow files (Section 7.5 and Section 7.7) and impact of AI tools and other technological changes (Section 7.6), and compare those findings with prior studies.

7.8.1 CI/CD Configuration File Evolution Patterns

This subsection compares the results of Section 7.3 to prior research. Our analysis over 267K+ workflow change histories revealed that roughly 7% of workflow files are modified each week, with a median of 82 days between successive updates. Modifications to workflow files clearly dominate over file additions, removals and renamings. One out of four repositories add more than one workflow file (with a median of three) during their lifetime, and removals tend to occur shortly after their creation. These results indicate that GitHub Actions workflows are undergoing a continued maintenance practice rather than a one-time setup.

Prior studies of CI/CD configuration evolution reported similar yet less pronounced trends. Hilton *et al.* (2016) investigated how often developers evolve their CI configurations by analysing the full history of `.travis.yml` files across 34,544 GitHub repositories. They reported a median of 12 configuration changes per repository, with 25% of them making five or fewer changes, suggesting that most teams follow a “set up once and adjust occasionally” pattern, while a minority frequently revise their CI setups. In comparison, our analysis of GitHub Actions workflows revealed a more active evolution pattern, with a median of 22 workflow file changes per repository, with 14% of them making five or fewer changes.

Zampetti *et al.* (2021) studied the evolution of 4,644 GitHub repositories, showing that Jenkins CI/CD configuration files typically change less often than production or test code but still undergo several updates throughout a repository’s lifetime. The median number of commits impacting Jenkins configuration files is 20, and 10% of the studied repositories have more than 100 such commits. In comparison, we found a median of 22 commits impacting GitHub Actions configuration files, with 10% of the repositories having more than 121 commits.

The above observations suggest a slightly higher modification frequency for GitHub Actions workflows than for other types of CI/CD configuration files.

We are not aware of any prior research having statistically studied the potential impact of LLM-based coding agents (such as Copilot) on the evolution patterns of CI/CD configuration files. This is unsurprising, given their relative

recency and the many perils that need to be overcome to reliably detect their presence, use and evolution in GitHub repositories (Robbes *et al.*, 2026). The peril of *partial observability* states that coding agents only leave partial traces of their activity (if at all). The perils of *agent diversity and multiplicity* state that different agents may work quite differently and hence may leave traces in very different ways. Finally, the peril of *high velocity* implies that the way agents function and are being used changes very rapidly. Because of this, quantifying the impact of LLM-based agents on specific development practices is very challenging. In this paper, we conducted a before-after analysis to try to detect whether their introduction had any significant impact on the change frequency of GitHub Actions workflow commits, but we observed only negligible effects at best. Further work at a more fine-grained level, based on the mitigation heuristics proposed in Robbes *et al.* (2026), would be needed to study any positive or negative impact of the introduction and use of specific LLM-based tools, either during the early phase or sustained lifetime of workflow configuration files.

7.8.2 Conceptual Changes in Workflows

In Section 7.4 we qualitatively analysed and classified 439 distinct workflow changes into seven conceptual categories (*e.g.*, modifications, additions or removals of workflow content changes made to the categories of task specification, task configuration, documentation, etcetera). The results indicated that changes to GitHub Actions workflows predominantly focus on *task specification* and *task configuration*, mirroring developers' emphasis on refining execution logic and adjusting the parameters of reusable Action components.

Earlier studies on the evolution of other CI/CD services such as Travis and Jenkins (Beller *et al.*, 2017; Zampetti *et al.*, 2021; Zampetti *et al.*, 2020) reported similar high-level categories. For instance, the “build logic” category of (Beller *et al.*, 2017; Zampetti *et al.*, 2021; Zampetti *et al.*, 2020) corresponds to our *task specification* concept, their “environment configuration” to our *runtime configuration* concept, and their “documentation” category to our *documentation* concept. This overlap demonstrates that GitHub Actions workflow maintenance is similar in nature to maintaining workflows or pipelines for other CI/CD services, though expressed through different syntactic constructs.

We also analysed to what extent commits modify multiple workflow files simultaneously and what was the nature of these changes. The majority of the identified cases (68 out of 95) consisted of the same kind of change being made to multiple workflows, mostly for shallow routine maintenance-oriented activities. While some of these activities (*e.g.*, dependency updates) were

automated via tools, we suspect that many other co-changes were actually performed manually. This provides an opportunity for tool builders to better support cross-workflow maintenance beyond dependency automation. For researchers, the presence of cross-workflow changes provides an opportunity to shift attention from studying isolated workflows to more ecosystem-wide studies of workflow changes, not only within individual commits, but even across different repositories. As an illustration of such research, Cardoen *et al.* (2026) empirically analysed the prevalence of duplication within workflow files, but also across workflow files belonging to different repositories.

7.8.3 Evolution of Workflow Contents

Section 7.5 and Section 7.7 jointly investigated how the contents of individual GitHub Actions workflow files evolve and, more specifically, what kinds of changes are made (additions, removals, or modifications) and which workflow entities are most frequently affected. Section 7.5 identified 7,811,892 individual changes across all workflow histories, including 3.5M+ modifications, 1.5M+ additions, 887K+ removals, and 48K+ renames. Nearly four out of five changesets (78.98%) consist of a single change type, most often modifications (78.81%). This share grows over time, suggesting that workflow evolution is dominated by incremental refinements rather than major restructuring. These quantitative results indicate that most workflows evolve through continuous adjustments rather than complete redesigns, mirroring earlier findings for other CI/CDs such as Travis and Jenkins (Ghaleb *et al.*, 2024; Hilton *et al.*, 2016; Zampetti *et al.*, 2021). We also statistically analysed possible effects of evolving technology (such as the introduction and use of LLM-based coding tools) in Section 7.6 on the frequency of changes to workflow entities, but we observed only *negligible* differences in change frequency for *modifications* to workflow entities. As already mentioned in Section 7.8.1, more fine-grained future research would be needed to study other possible effects, taking into account the many perils and pitfalls that come with such analysis (Robbes *et al.*, 2026).

Section 7.7 examined the distribution of changes across workflow entities, observing that nearly 80% of all changes occur in the `jobs:` section, particularly within `steps:`. Most of these correspond to *task configuration* activities (`uses:`, `run:`, and `with:`), while changes to `name:`, `matrix:`, and `permissions:` reflect adjustments to *documentation*, *task specification*, and *runtime configuration*, respectively. Taken together, these trends show that workflow maintenance is primarily about fine-tuning rather than structural reorganisation.

A deeper examination of the `uses:` entity revealed distinct maintenance

strategies. Among 1.3M+ changes to this entity, 85.2% correspond to Action version updates, while 14.8% involve replacing the Action entirely. Such Action replacements arise when existing Actions become outdated, deprecated, or unmaintained, consistent with prior reports on ecosystem health and abandonment (Avelino *et al.*, 2019; Constantinou & Mens, 2017; Kaur & Kaur, 2022). Some of these Action replacements may also aim to improve performance or optimise execution, similar to the optimisation opportunities identified by Bouzenia and Pradel (2024). For example, we found cases of workflows replacing actions/download-artifact with actions/cache to improve efficiency, or migrate from deprecated Actions (such as ghaction-docker-buildx which has been created and maintained by community) to officially supported ones by docker organisation. Another type of change relates to replacing the uses: entity by the run: entity to gain finer control and reduce dependency risks, or vice versa to increase workflow modularity. This duality reflects broader maintenance trade-offs between reusability and autonomy.

Our findings revealed a positive trend towards improved workflow security to reduce the GitHub Actions attack surface, through an increase in the use of permissions to enforce the principle of least privilege, the extensive use of automated tools (such as Dependabot and Renovate) to keep versions up to date, and the increased use of more secure Action versioning through commit-hash version pinning. While prior research reported limited use of Action version pinning (only 1.6% reported by Decan *et al.* (2022)), our findings show that nearly half of all Action version updates employ commit hashes, indicating a growing tendency toward secure versioning practices. However, this observation is based on changes rather than on the overall population of workflows, thus, the increased proportion of commit-hash updates could reflect more frequent updates within a subset of workflows rather than widespread adoption across all repositories. Our observations about increased use of dependency automation tools for workflows aligns with prior findings that projects relying on such tools reduce their exposure to security issues and technical debt (Mohayjeji *et al.*, 2023). With respect to the use of permissions, automated tools such as StepSecurity¹⁰ could help to further enforce the principle of least privilege, but remain underexploited. These results align with broader observations on the evolving maturity of the GitHub Actions ecosystem (Decan *et al.*, 2023), in which maintainers progressively balance reuse, security, and organizational consistency.

In summary, the analyses of Section 7.5 and Section 7.7 portray the continuous evolution of GitHub Actions workflows to satisfy diverse needs. Work-

¹⁰<https://www.stepsecurity.io>

flows evolve through incremental modifications, focused on dependency management, configuration refinement, and security improvements, rather than large-scale redesign. Future work could further contextualize these findings by linking workflow change histories with execution logs, such as those proposed by Moriconi *et al.* (2025), and enriched metadata such as commit messages or issue references, enabling a more comprehensive understanding of how workflows evolve in response to runtime outcomes, errors, and developer intent.

7.9. Threats to Validity

We follow the structure recommended by Wohlin *et al.* (2012) to discuss possible threats to validity of our research.

Construct validity concerns the relation between the theory behind the experiment and the observed findings. The dataset of workflow files we used identified the presence of workflow files by detecting (valid) YAML files within the designated `.github/workflows` directory. As a consequence, it may be the case that some workflows in this dataset were not actively used (i.e., never triggered for execution) or were not even intended to be used. Moreover, since changes to workflow files reflect developer effort and intent, the effort of maintaining workflow files remain valuable independently of whether the workflow is actively being used or not.

A second threat arises from our decision to exclude workflow histories containing invalid workflow files, as explained in Section 7.2. We did so in order to ensure quality and syntactic correctness of the data under scrutiny. This led to the exclusion of 11.54% of all workflow files histories. Those workflow histories may exhibit a different change patterns compared to valid workflow files, particularly in terms of debugging. However, there is no evidence that this would be the case.

A third threat relates to the fact that we only considered the primary branch of each repository when detecting workflow file changes, excluding changes occurring in parallel branches. While this prevented us from studying fine-grained changes in parallel branches, we argue that this is not a significant threat to our study. Indeed, the primary branch is typically the most active one, reflecting the development state of the repository. Additionally, changes made in parallel branches are usually merged back into the primary branch or discarded, meaning that the primary branch will eventually reflect all changes made in the repository. The main bias related to this threat stems in how changes are merged from parallel branches to the primary branch: changes from parallel branches can be merged with a squashing strategy, implying that

multiple distinct changes are combined into one large changeset, potentially altering the observed evolution patterns. Unfortunately, there is no way to detect whether a squashing strategy was used or not, as the git history does not retain information about the merging strategy (Bird *et al.*, 2009).

Internal validity relates to the extent to which the study results are influenced by the experimental treatment or condition being studied. One such treat is related to the gawd tool we used for analyzing workflow file changes. The tool can be parameterised in many ways to detect different types of changes in workflow files. We relied on gawd’s default configuration for our analysis, but other settings might produce different results. To mitigate this threat, we manually checked the results of gawd on a small sample of workflow file changes to ensure that the default configuration was appropriate for our study.

Conclusion validity concerns the degree to which reasonable conclusions have been derived from our analysis. The qualitative analysis in Section 7.4 was based on a manual analysis of a limited number of workflow files, potentially leading to an incomplete classification of change concepts. We mitigated this threat by analysing workflow file changes until saturation was reached, ensuring no new change types were emerging. Another potential threat to conclusion validity arises from the subjective interpretation of changes in workflow files. This subjectivity could lead to misclassification or biased analysis of the observed changes, potentially impacting the reliability of our findings. To mitigate this risk, we followed the well-established framework method (Spencer *et al.*, 2004) to guarantee a structured and systematic qualitative analysis. Additionally, we involved multiple researchers in the process to cross-verify interpretations, thereby reducing the likelihood of individual biases and ensuring a more robust understanding of the data. Finally, the qualitative findings from Section 7.4 were triangulated with quantitative results in Section 7.7, further supporting the validity of our conclusions.

External validity concerns the generalisability of the results beyond the specific data being analysed. The dataset we relied on to conduct our study is based on active GitHub repositories (i.e., those that had at least one commit since October 2024), that have at least 300 stars and 300 commits. The rationale behind these criteria is to exclude repositories that are not representative of typical software development practices, such as abandoned, personal, or experimental repositories (Kalliamvakou *et al.*, 2014). While we share the rationale behind these criteria, it is important to note that our findings may not be generalisable to smaller or less active repositories that may employ workflows for different purposes, such as publishing GitHub Pages or managing personal projects. Consequently, the applicability of our conclusions to

these contexts remains uncertain.

7.10. Summary and Conclusions

This chapter presented a mixed-methods empirical study of how GitHub Actions workflows evolve over time, examining which workflow components are most frequently modified and how changes manifest at scale.

The qualitative analysis manually examined 1,109 changes across 439 workflows, clustering them into three change types (modification, addition, removal) across seven conceptual categories. Modifications were the most common, primarily affecting task specification and configuration. Commits changing multiple workflows were not uncommon but generally reflected routine, maintenance-oriented updates.

The quantitative analysis covered 267,955 workflow histories, encompassing 3,418,911 workflow files across 49,258 repositories over six years. Around one in four repositories added multiple workflows, with files updated on average every 159 days and roughly 7% modified weekly. Modifications dominated the change types, often occurring in bursts that indicate iterative debugging and trial-and-error maintenance. Automated analysis showed that most changesets consisted solely of modifications, which increasingly dominate over time. Nearly 80% of modifications were concentrated in ten workflow paths, mainly affecting jobs and steps, corresponding to task specification and configuration. Automated tools, particularly dependency update utilities, accounted for about one in five changesets, with trends toward safer versioning and finer-grained permission control.

Importantly, the analysis examined whether the emergence of AI-assisted coding tools, such as LLMs, as well as other technological innovations in development environments or workflow management, influenced the frequency or burstiness of workflow changes. Overall, the results did not provide conclusive evidence that the introduction of these tools or technologies had a significant effect on either the frequency or the burst behavior of changes.

These findings demonstrate that GitHub Actions workflows are dynamically maintained, continuously evolving configuration components. Iterative, fine-grained modifications reflect ongoing adaptation to performance, reliability, dependency, and functional requirements.

Conclusion

“Avoid the temptation to work so hard
that there is no time for serious thinking.”

Francis Crick

This chapter concludes the dissertation by consolidating its empirical insights on the adoption, large-scale usage, evolution, and maintenance of GitHub Actions workflows. The results collectively demonstrate that workflows exhibit software-like evolutionary behavior, are subject to continuous fine-grained maintenance, and are deeply embedded in collaborative development practices and platform ecosystems.

Beyond summarizing contributions, the chapter reflects on methodological and data-related limitations and identifies key research opportunities arising from emerging trends such as AI-assisted workflow authoring, agentic automation, and increasing security and sustainability concerns. These perspectives underscore the need to rethink CI/CD workflows as first-class software artifacts whose long-term quality, resilience, and governance are essential to the sustainability of modern software ecosystems.

8.1. Research Goals and Contributions

This dissertation investigated the adoption, usage, and evolution of GitHub Actions workflows in collaborative software development. Specifically, it aimed to examine how developers integrate and maintain workflow automation, analyze the evolutionary dynamics of workflows over time, including which components are most frequently modified, and explore what these dynamics reveal about maintenance and adaptation practices within this GitHub Actions ecosystem.

Through a mixed-method empirical research design structured around four research goals, this dissertation provided systematic evidence supporting this central claim. Each research goal addressed a complementary dimension of the dissertation statement, collectively contributing to a comprehensive understanding of how GitHub Actions workflows are adopted, evolve, and are maintained in practice.

8.1.1 Adoption and Tool Selection of CI/CD Workflows

The first research goal examined how CI/CD tools are adopted and used in contemporary software projects, with a focus on understanding the factors that influence tool selection and long-term reliance. Chapter 4 addressed this goal through a qualitative interview study involving 22 software practitioners from diverse organizational and technical contexts.

The study revealed that CI/CD tool adoption is shaped by a combination of technical, and organizational considerations. Participants identified platform integration, configuration flexibility, ecosystem maturity, availability of reusable components, and anticipated maintenance effort as central decision criteria. Organizational constraints, such as existing infrastructure, security policies, and team expertise, also played a significant role in shaping adoption choices. Within this landscape, GitHub Actions consistently emerged as a dominant solution, largely due to its tight integration with the GitHub platform and its extensive marketplace of reusable Actions.

Beyond identifying adoption drivers, this chapter established the broader context in which workflow automation tools operate. CI/CD tools were not perceived as isolated utilities, but as infrastructure embedded within collaborative development practices, organizational processes, and evolving ecosystems.

8.1.2 Large-Scale Usage of GitHub Actions in Practice

Building on the qualitative insights into adoption, the second research goal investigated how GitHub Actions is used in practice across a large and diverse

set of real-world repositories. Chapter 5 addressed this goal through a large-scale quantitative analysis of nearly 30,000 repositories and more than 70,000 workflow files.

The results showed that GitHub Actions has achieved widespread and steadily increasing adoption, with workflows playing a central role in automating build, test, and deployment activities. Usage patterns were characterized by recurring structural motifs, standardized triggering mechanisms, and extensive reuse of third-party Actions sourced from the GitHub Marketplace. Cross-project reuse was particularly prevalent, suggesting that workflows are commonly shared across repositories and contribute to a broader automation ecosystem.

At the same time, the analysis uncovered emerging challenges associated with these practices. Heavy reliance on third-party Actions introduces implicit dependencies, versioning risks, and maintenance burdens that are not always visible at the repository level. While standardized patterns support consistency and reuse, they can also propagate outdated or suboptimal configurations at scale. Collectively, these findings highlight both the productivity benefits and the potential maintenance and quality concerns associated with large-scale workflow reuse.

8.1.3 Evolutionary Characteristics of CI/CD Workflows

The third research goal examined whether GitHub Actions workflows exhibit evolutionary behavior comparable to that of traditional software artifacts. Chapter 6 addressed this question by analyzing workflow changes over time at both file-level and line-level granularity.

The findings indicate that workflows evolve continuously through frequent, small-scale modifications rather than sporadic, large restructurings. Most changes occur at the line level and reflect incremental refinements to existing configurations, such as adjusting parameters, updating commands, or refining execution conditions. This evolutionary behavior aligns with classical software evolution laws, suggesting that workflows should be understood as actively maintained software artifacts rather than static configuration files.

To support this analysis, this chapter introduced *gawd*, a syntax-aware differencing tool specifically designed for GitHub Actions workflows. By operating at the level of workflow syntax rather than raw text, *gawd* enabled fine-grained analyses of how specific workflow components evolve over time. This methodological contribution strengthened the empirical findings of the dissertation and provides a basis for future research on workflow evolution.

8.1.4 Maintenance Hotspots and Change-Prone Workflow Components

The fourth research goal focused on identifying and characterizing those workflow components that are most prone to change, thereby shedding light on the maintenance dynamics of GitHub Actions workflows. Chapter 7 addressed this goal through a mixed-methods study combining qualitative and large-scale quantitative analyses.

The qualitative analysis examined 1,109 workflow changes across 439 modified files and identified seven conceptual categories of workflow change spanning three primary change types: modification, addition, and removal. Modifications overwhelmingly dominate workflow evolution, particularly within task specifications and job configurations. Multi-workflow commits were also common but typically reflected routine and systematic maintenance activities rather than coordinated architectural redesign.

These insights were confirmed quantitatively through an analysis of 267,955 workflow histories encompassing more than 3.4 million workflow versions from over 49,000 repositories and spanning a period of more than six years. Repositories contained a median of three workflow files, approximately 7% of which were modified weekly. Most changesets were small and incremental, often consisting of a single modification. Over time, modification-only changesets became increasingly prevalent, reinforcing the view that workflow evolution is driven primarily by continuous fine-grained maintenance rather than structural change.

At a finer granularity, nearly 80% of modifications are concentrated in a limited set of workflow paths, primarily job definitions and individual steps. Bursty commit patterns were also observed, which may reflect iterative debugging and trial-and-error in workflow maintenance, though further analysis of execution logs would be needed to confirm this interpretation. A before–after analysis examining the introduction of LLM-based coding tools and other technological changes do not reveal substantial changes in workflow modification frequency or burst behavior; however, no conclusive claims about the overall stability of workflow evolution patterns can be made based on this analysis alone.

Finally, the chapter highlights the substantial influence of automated maintenance tools. Dependency update bots such as Dependabot and Renovate were found in a large fraction of workflow histories and were associated with safer versioning practices and more fine-grained permission management. Taken together, these results position GitHub Actions workflows as actively maintained software configuration components and point to opportu-

nities for improved tooling to support debugging, quality assurance, security, and context-aware maintenance.

8.2. Limitations

Despite the breadth of data and the use of multiple complementary methods, this dissertation is subject to several limitations that should be considered when interpreting its findings. These limitations relate to potential threats of construct, internal, external, and conclusion validity, and they also highlight directions for future research.

8.2.1 Dataset Representativeness

All quantitative analyses in this dissertation are based on public GitHub repositories that satisfy minimum thresholds of popularity and activity (*e.g.*, at least 100 or 300 stars and commits, depending on the chapter; see Chapters 5, 6, and 7). While these criteria improve data quality and reduce noise from inactive or toy projects, they limit the representativeness of the dataset. In particular, smaller, personal, educational, or niche repositories are underrepresented. As a result, the observed practices may reflect the behavior of relatively mature or visible projects rather than the broader population of GitHub users.

In the qualitative sample, some geographic regions were underrepresented. Moreover, developers across regions do not have equal access to CI/CD platforms, cloud infrastructure, or paid services due to geopolitical, regulatory, or financial constraints, which may influence tool adoption, usage patterns, and reported experiences.

Future work could improve generalizability by using stratified or weighted sampling across repository activity levels and by recruiting developers from underrepresented regions and project types through targeted outreach to project maintainers and developer communities. External validity could be further strengthened by incorporating industrial datasets obtained via research collaborations or data-sharing agreements.

8.2.2 Workflow Execution and Activity Uncertainty

Throughout the dissertation, the presence of YAML files in the *.github/workflows* directory was used as a proxy for CI/CD adoption. This operationalization may overestimate actual CI/CD usage, as some workflows may be inactive, deprecated, or rarely triggered (Chapters 5 and 7). Recently, GitHub has introduced mechanisms that automatically disable workflows that have not been

triggered or modified for extended periods. While this platform-level behavior partially mitigates the risk of counting long-abandoned workflows as active CI/CD usage, it does not fully eliminate the limitation, as disabled workflows may remain present in repositories and are not distinguishable from active ones in repository snapshots.

Furthermore, the analyses focus on workflow definitions rather than execution behavior, as execution logs and runtime telemetry were not available at scale. Consequently, changes to workflow files cannot be directly linked to execution outcomes such as failures, performance regressions, or reliability improvements. Integrating execution traces, logs, and failure reports would enable stronger causal reasoning about the relationship between workflow evolution and developer intent, and represents an important avenue for future research.

8.2.3 Branch and History Scope

Workflow evolution was analyzed either through weekly snapshots of repository states (Chapter 6) or by focusing on primary branches only (Chapter 7). While primary branches typically capture the consolidated state of development, this approach may miss changes introduced and reverted in feature branches, as well as fine-grained evolution that is flattened by squash merges or rebasing. This may have affected the granularity and accuracy of our evolution analysis.

In addition, invalid YAML workflow files were excluded from some of our analysis to ensure data quality, potentially omitting real-world debugging activities and correction attempts. Future studies could analyze full commit graphs, pull request histories, and merge strategies to provide a more complete view of workflow evolution. Moreover, invalid YAML files can be studied in a separate research on the mistakes that maintainers make when working on workflow files.

8.2.4 Tooling Constraints

The use of the gawd differencing tool enabled entity-aware comparison of workflow files and supported large-scale analysis, but it also introduced limitations. In particular, the tool ignores cosmetic changes such as comments and whitespace and does not explicitly detect higher-level refactorings, including step reordering, job decomposition, or structural reorganization (Chapter 7). Moreover, default configuration parameters were used throughout the analysis; although validated on a sample, they may not capture all meaningful variations.

Future tooling could combine syntactic and semantic differencing, incorpo-

rate refactoring detection, and operate on higher-level workflow abstractions to better reflect developer intent and maintenance effort.

8.2.5 Subjectivity in Qualitative Analysis

Chapters 4 and 7 relied on qualitative methods, including semi-structured interviews and manual coding of workflow changes. Despite following strategies such as coder triangulation, saturation checks, and the use of structured coding frameworks (*e.g.*, the Framework Method by Spencer *et al.* (2004)), some degree of subjectivity and interpretation bias is unavoidable.

Interview results may also be influenced by participant self-selection, demographics (*e.g.*, regions) and by the predominance of experienced developers, potentially underrepresenting the perspectives of novices or less confident users, or developers from specific regions. Replication studies and complementary automated analyses could help reduce subjectivity. Including less experienced developers, developers who rely heavily on AI-based tools, and automated agents as study subjects would further broaden the empirical perspective by capturing alternative development practices, decision-making strategies, and maintenance behaviors that may differ substantially from those of experienced, tool-centric practitioners.

8.2.6 Generalizability of Findings

The conclusions of this dissertation are shaped by both the characteristics of the dataset (*e.g.*, active and popular repositories) and the methodological choices (*e.g.*, weekly snapshots and bounded history windows). As a result, certain phenomena such as very long-term deprecation cycles, rare usage patterns, or highly bursty activity may be underrepresented or only partially captured.

Moreover, the findings may not generalize to other forms of automation, such as documentation pipelines, or CI/CD ecosystems outside of GitHub Actions. Future research could improve generalizability by conducting longer-term longitudinal studies and by examining domain-specific automation scenarios across a wider range of platforms.

8.3. Future Research Perspectives

The results of this dissertation open several promising directions for future research on workflow automation.

8.3.1 Linking Workflow Changes to Execution Behavior

Although this dissertation primarily examines syntactic and structural changes in workflow files, many of these modifications are ultimately driven by execution-time behavior, such as improving performance, mitigating flakiness, or addressing failures caused by external dependencies, platform changes, or environmental inconsistencies. Prior work has demonstrated that GitHub Actions is widely used to automate a variety of development activities, and that execution outcomes play a central role in shaping workflow usage and evolution (Kinsman *et al.*, 2021). An important next step is to systematically connect workflow change histories with their execution logs, an opportunity that is enabled by the availability of large-scale datasets of workflow runs, such as GHALogs (Moriconi *et al.*, 2025).

By combining workflow diffs with data such as build duration, failure probabilities, caching effectiveness, and resource utilization, future research can determine which types of workflow changes (i) result in measurable improvements, (ii) indicate recurring runtime failures, and (iii) constitute reactive responses to ecosystem-level changes. Such insights would enhance our understanding of maintenance motivations and facilitate the development of evidence-based best practices for workflow optimization. This direction is motivated by the findings regarding the commit burst patterns identified in Chapter 7. Our results show that up to 22% of workflow histories exhibit bursts of rapid successive commits, with some cases reaching up to 50 commits within a short period. This interpretation is also qualitatively supported by Chapter 4, where practitioners reported making repeated commits to observe CI outcomes in the absence of local testing support; however, without access to execution logs, this could not be directly confirmed. Linking change histories to execution traces would enable future studies to examine whether burst-heavy workflow histories are associated with higher failure rates, and whether specific entity-level changes, such as modifications to `run:` steps or `uses:` version pins, consistently precede or follow build failures, thereby transforming descriptive observations of change frequency into actionable, empirically grounded insights into maintenance triggers.

8.3.2 Detecting Workflow Smells and Anti-Patterns

Just as code smells guide refactoring activities in traditional software, CI/CD workflows exhibit recurring workflow smells that negatively affect maintainability, security, or correctness. Examples include duplicated logic, excessively long jobs, unsafe permission settings, redundant triggers, and misconfigured

dependencies. Recent work has begun to systematically identify such issues in GitHub Actions workflows, including GASH (Freitas & Rocha, 2024), which introduces an initial catalog and detection tool for workflow smells, as well as empirical studies of security misconfigurations (Riggio & Pautasso, 2025), broader analyses of workflow quality issues (Khatami *et al.*, 2024), and recent work on code clones in workflow evolution (Cardoen *et al.*, 2026).

Future work could build on these initial efforts by designing and validating (i) rule-based smell detectors, (ii) machine learning models trained on historical evolution patterns, and (iii) automated refactoring suggestions aimed at removing detected smells and improving reuse. Subsequent evaluations should measure detection precision, false positive rates, and practitioner acceptance. This line of work could ultimately result in tools that provide proactive, real-time feedback to developers as they edit workflow files.

Our findings offer a direct empirical basis for this direction. The seven-category taxonomy of conceptual workflow changes developed in Chapter 7 can serve as a structured foundation for defining smell types and linking them to specific workflow entities. For example, the observed concentration of nearly 80% of all changes within just ten workflow paths, primarily in `steps:`, `uses:`, `run:`, and `with:`, indicates that smell detectors should prioritize these entities. Moreover, the 11.54% of workflow histories excluded from our analysis due to invalid YAML files represent a largely unexplored set of potentially broken or incorrectly authored workflows. Examining these excluded histories in particular could uncover recurring authoring errors, structural anti-patterns, and error-prone configurations that remain undetected in evolution analyses restricted to valid workflow files.

8.3.3 Context-Aware Recommendation Systems for Workflow Design

Given the diversity of real-world workflows analyzed in this dissertation, there is considerable potential to develop context-aware recommendation systems that assist developers in designing or updating GitHub Actions workflows. Such recommenders could suggest safer or more up-to-date Action versions, alternative job structures, security hardening patterns, domain-specific configuration idioms, dependency update strategies tailored to project characteristics, or improvements in documentation quality based on workflow context. Recent work such as CIGAR by Huang and Lin (2023), which leverages step-level context to recommend meaningful step names, as well as approaches by Mastropaolo *et al.* (2024) and Maddila (2022) that aim to automatically complete workflows, demonstrate the feasibility of such intelligent assistance.

At the same time, GitHub has recently introduced enhanced native support for authoring workflows, both within the GitHub web interface and through editor integrations, and developers can increasingly rely on general-purpose AI assistants such as GitHub Copilot for workflow auto-completion and guidance (GitHub, 2026). While these tools reduce the barrier to writing workflows, they primarily offer syntactic assistance or generic code generation and may not fully consider project-specific constraints, long-term maintainability, security policies, or ecosystem-level best practices.

This raises two complementary research needs. First, there remains an opportunity for context-aware recommendation systems that extend beyond generic completion by explicitly leveraging repository history, workflow evolution patterns, dependency ecosystems, and organizational conventions. Second, there is a clear need for empirical studies that evaluate how the increasing use of generative AI-based tools influences workflow maintenance in practice. Such studies could investigate whether AI-assisted authoring alters the frequency, nature, or quality of workflow modifications, introduces new maintenance risks (*e.g.*, configuration bloat, security misconfigurations, or reduced developer understanding), or improves consistency and robustness over time.

Beyond recommendation, emerging agentic workflow frameworks suggest a further shift toward semi-autonomous or fully autonomous workflow management systems (GitHub Next, 2025). Future research should examine when recommendation is sufficient and when agentic delegation becomes appropriate, as well as how control, override mechanisms, and human-in-the-loop safeguards should be designed.

Future work could address these questions through longitudinal analyses of workflow evolution before and after the adoption of AI-assisted tools, controlled user studies comparing AI-supported and manual workflow maintenance, and mixed-methods investigations that combine repository mining with developer interviews. Evaluating both the benefits and unintended consequences of AI-assisted workflow authoring is essential for understanding how these tools reshape CI/CD practices and for informing the design of next-generation, workflow-aware recommendation systems.

Findings of this dissertation provide inputs for the design of such systems. The analysis of uses: changes in Chapter 7 shows that 85.2% of modifications to this entity correspond to Action version updates, with an increasing trend toward commit-hash pinning and fully specified version tags. This indicates that a substantial portion of routine workflow maintenance is both predictable and amenable to automation. A recommendation system grounded in our evolution data could proactively identify outdated Action references, suggest

more secure version formats, or warn against Actions from providers exhibiting signs of abandonment. Similarly, the 9,827 observed instances of replacing `run:` with `uses:` and the 7,230 reverse replacements highlight a recurring trade-off between reuse and control. Recommendation systems capable of characterizing when each pattern is more appropriate, based on security context, team conventions, would address a concrete maintenance challenge identified in our data. The high frequency of manual co-changes across multiple workflow files (observed in 21% of multi-workflow commits) further points to an opportunity for cross-workflow refactoring recommendations that are not supported by current tools.

8.3.4 Cross-Ecosystem and Longitudinal CI/CD Analyses

Although this dissertation focuses exclusively on GitHub Actions within the GitHub ecosystem, the broader landscape of collaborative software development and CI/CD automation encompasses multiple platforms and tooling paradigms. Future research should explicitly separate these two dimensions, namely *collaborative development platforms* and *CI/CD automation tools*, as each presents distinct research opportunities and methodological challenges.

Comparative Analyses Across Collaborative Development Platforms. From the platform perspective, GitHub represents only one of several environments that support large-scale collaborative software development. Other major platforms include GitLab, BitBucket, Azure DevOps, Codeberg/-Forgejo, and cloud-centric environments such as GCP, AWS, Kubernetes-based ecosystems, and Azure services. While these platforms differ in terms of openness, governance models, and intended use, they share core features such as repository hosting, issue tracking, access control, and CI/CD integration. Prior work has shown that platform design, governance, and openness significantly influence developer collaboration and project evolution (M. L. Gupta *et al.*, 2024; Kalliamvakou *et al.*, 2014; Storey *et al.*, 2010).

Comparative analyses across these platforms could examine how variations in platform design, governance, and default automation tooling influence CI/CD adoption, workflow complexity, and maintenance practices. However, such studies face practical limitations: many enterprise-oriented platforms predominantly host private repositories, which restricts the availability of large-scale public data. Consequently, future research may need to complement large-scale mining of public repositories (*e.g.*, GitHub, GitLab, Codeberg) with qualitative approaches such as in-depth case studies, industrial collaborations,

or in person observations within organizations using platforms like Azure DevOps, AWS, or GCP (M. L. Gupta *et al.*, 2024; Shahin *et al.*, 2017).

Comparative Analyses of CI/CD Automation Tools. Many platforms provide tightly integrated, built-in CI/CD solutions (*e.g.*, GitHub Actions, GitLab CI/CD, Google Cloud Build, Azure Pipelines, AWS CodeBuild), often based on declarative configuration-as-code paradigms. These built-in tools are typically optimized for their host platforms, offering deep integration but limited portability as prior studies have shown that such tight integration lowers adoption barriers but may also increase ecosystem lock-in (Hilton *et al.*, 2016; Zampetti *et al.*, 2021).

In contrast, independent CI/CD solutions such as Jenkins, CircleCI, TeamCity, and Travis aim to operate across multiple platforms and infrastructures. Comparing these tools could reveal how portability, extensibility, and ecosystem independence affect workflow design, evolution, and maintenance. For example, future research could examine whether cross-platform tools encourage more abstract or reusable workflow designs, or whether platform-specific tools lead to tighter coupling and faster ecosystem-driven evolution.

Longitudinal and Socio-Technical Perspectives. Longitudinal analyses spanning multiple years and ecosystems could further clarify how CI/CD practices evolve in response to external factors such as changes in pricing models, security incidents, policy updates, ecosystem consolidation, or shifts in development culture. Such studies could reveal whether the workflow evolution patterns observed in GitHub Actions, such as incremental changes, dependency-driven maintenance, and automation by bots, are universal phenomena or artifacts of a particular platform-tool pairing. Prior work on software ecosystems and build systems suggests that these external pressures play a decisive role in shaping tooling evolution and maintenance practices (Mokhov *et al.*, 2018; Rahman *et al.*, 2018). Furthermore, there are opportunities to examine how socio-technical factors such as team size, organizational structure, and developer experience influence CI/CD adoption and workflow evolution across different platforms and tools (M. L. Gupta *et al.*, 2024).

This broader perspective is essential for evaluating the generalizability of findings from GitHub-centric studies and for informing the design of future automation platforms that better support sustainable and portable CI/CD practices.

This dissertation establishes a concrete baseline for such comparative analyses. The specific evolution metrics we characterized for GitHub Actions,

including a median update interval of 82 days, a weekly modification rate of approximately 7%, the predominance of `uses`: version updates as the most frequent entity-level change, and the strong reliance on a limited set of Action providers, could be directly replicated on other collaborative platforms such as GitLab CI/CD or Bitbucket Pipelines to determine whether these patterns are platform-specific or ecosystem-independent. The `gawd` differencing tool introduced in Chapter 6, which operates at the level of syntactic entities rather than raw text, could in principle be adapted to other YAML-based CI/CD formats, thereby providing a shared methodological foundation for cross-ecosystem comparisons.

8.3.5 Automation Through AI Agents

AI-driven tools such as GitHub Copilot, autonomous coding agents, and agentic frameworks are increasingly being integrated into the authoring, maintenance, and execution of CI/CD workflows. Recent developments, including Microsoft’s enterprise-ready Copilot coding agent embedded within GitHub Actions, reflect a broader shift toward agentic, AI-mediated automation (GitHub, 2025i). Built on LLMs, these systems enable agents to reason about code and configuration, collaborate with other agents, and dynamically adapt their behavior through frameworks such as AutoGen, CrewAI, LangChain, and Semantic Kernel, supported by emerging interaction standards such as Agent2Agent protocol (A2A) and Model Context Protocol (MCP) (Acharya *et al.*, 2025; Melo, 2025; Schneider *et al.*, 2024; Treude & Poskitt, 2025). At the same time, the GitHub ecosystem is evolving toward more advanced, configuration-aware tooling for CI/CD, with proposals and early developments pointing to future editor support for `action.yml` files, including schema-aware autocompletion, validation, and scaffolding within the GitHub UI. Such tooling would further reduce the barrier to authoring and maintaining reusable automation components and complements the rise of AI-assisted development.

In February 2026, GitHub introduced the concept of Agentic Workflows, known as *gh-aw*, which embed autonomous AI agents directly within GitHub Actions pipelines (GitHub Next, 2025). Unlike traditional AI-assisted authoring tools that provide suggestions during editing, agentic workflows enable runtime reasoning, multi-step task execution, and natural-language-defined automation compiled into executable CI/CD configurations. This paradigm shifts CI/CD from static configuration-as-code toward AI-mediated orchestration embedded directly within the execution layer.

While such tools promise productivity gains and reduced manual effort, empirical evidence presents a more nuanced picture. Prior studies report modest

productivity improvements, but also observe longer integration times and limited or inconsistent effects on code quality (Song *et al.*, 2024). Our large-scale longitudinal analysis of CI/CD workflows similarly found no conclusive evidence that the introduction of LLM-based coding tools or other technological changes resulted in systematic changes in workflow creation frequency, maintenance frequency, or commit burst behavior at the population level. Despite the increasing availability and adoption of AI assistance, workflow commit patterns remained largely stable before and after major technological milestones, suggesting that AI tools have not (yet) fundamentally altered how frequently or intensively workflows are modified in practice.

At the same time, generative AI is likely to reshape collaboration dynamics more than overall activity levels. Developers may increasingly depend on AI assistance and future configuration-aware editor support instead of manual reasoning or peer interaction, potentially influencing knowledge sharing and collective understanding over time (Ulfsnes *et al.*, 2024). Recurring bug patterns in LLM-generated code highlight the need for systematic validation and quality assurance, particularly when AI-generated artifacts are incorporated into CI/CD workflows, where failures can propagate rapidly and affect entire development pipelines (Tambon *et al.*, 2025). Even with improved tooling, ensuring semantic correctness, design intent, and operational suitability remains challenging.

Beyond productivity and quality considerations, AI-assisted workflows introduce deeper challenges related to transparency, reproducibility, and long-term maintainability. AI-generated or AI-assisted configurations may be difficult to audit or fully understand, contributing to knowledge erosion and reduced accountability. The probabilistic nature of LLMs also introduces risks of non-deterministic behavior, flaky pipelines, and configuration drift, particularly as agentic systems evolve toward greater autonomy and may eventually modify workflows dynamically. Prior research on bots in collaborative software development has shown that poorly integrated automation can disrupt developer workflows (Wessel *et al.*, 2021), even as experienced developers increasingly rely on bots as primary interaction interfaces (Wang *et al.*, 2023). As agentic systems mature, human contributors may increasingly shift toward supervisory and orchestration roles rather than direct implementation roles (Melo, 2025; Treude & Poskitt, 2025), raising open questions about trust, responsibility, skill retention, and sustainability.

Future research should focus not only on whether AI agents increase activity or productivity, but also on how they, together with emerging configuration-aware tools, influence workflow quality, resilience, and developer cognition over

time. Longitudinal, mixed-method studies are needed to examine how AI-driven automation reshapes workflow evolution, maintenance strategies, and team dynamics beyond surface-level metrics such as commit frequency. Combining large-scale repository mining with execution data, surveys, interviews, and editor interaction data will be essential to capture both observable behavioral patterns and developer perceptions. Such evidence is crucial for grounding the integration of AI agents in CI/CD pipelines within empirically validated, transparent, and socially responsible engineering practices.

8.3.6 Security of CI/CD Workflows in an Agentic Era

Security remains a critical yet still insufficiently explored dimension of CI/CD workflow automation. GitHub Actions workflows are already exposed to a broad range of security risks, including dependency poisoning, excessive permissions, secret leakage, and supply-chain attacks (OWASP Foundation, 2023). These risks have been systematically documented by initiatives such as the Open Web Application Security Project (OWASP) Top 10 CI/CD Security Risks (GenAI, 2024), many of which directly apply to GitHub Actions workflows and reusable components.

The emergence of AI-assisted and agentic automation further amplifies these concerns. AI agents may autonomously modify workflows, select dependencies, or interact with external services, thereby expanding the attack surface and complicating threat modeling. Risks such as prompt injection, malicious manipulation of agent behavior, insecure dependency resolution, and unintended privilege escalation become particularly salient when LLMs are embedded in CI/CD pipelines or granted write access to repositories and configuration files (GenAI, 2024, 2025; Williams *et al.*, 2025). These threats extend previously identified vulnerabilities in GitHub Actions workflows and reusable Actions (Onsori Delicheh *et al.*, 2024) and align with broader concerns in software ecosystem security (Zerouali *et al.*, 2022), while introducing additional challenges related to agent autonomy, opacity, and non-deterministic behavior.

Agentic workflows, in which AI agents execute tasks within CI/CD pipelines with repository-scoped permissions (GitHub Next, 2025), introduce further layers of attack surface. Natural-language-defined automation and runtime model reasoning complicate traditional static analysis and threat modeling approaches, as execution logic may not be fully represented in declarative configuration files. Ensuring transparency, traceability, and deterministic behavior in such systems remains a critical open challenge.

Future research should focus on automated detection of insecure workflow configurations, principled permission minimization, continuous auditing

of workflow changes, and safeguards for AI-generated modifications. Empirical studies that combine workflow evolution data, security incidents, and AI adoption signals would be particularly valuable for quantifying risk and identifying failure modes.

Reported empirical results highlight both progress and remaining gaps in workflow security practices. We observed a notable increase in the adoption of **permissions: settings** over time, with 26,410 additions at the workflow level and 25,708 at the job level, as well as a growing trend toward commit-hash version pinning for reusable Actions. These trends indicate increasing security awareness. However, our data also show that tools such as StepSecurity, which can automatically enforce least-privilege permissions, remain underutilized in practice. This gap between available tooling and actual adoption represents a concrete research opportunity: future studies could investigate barriers to adopting security-hardening tools, measure residual risk exposure in repositories that do not use them, and develop lightweight automated interventions that reduce adoption effort. Studying the resilience of the GitHub Actions ecosystem to such concentrated dependency risks, and designing early-warning mechanisms for maintainers, is a research direction that follows directly from our usage and evolution findings.

8.3.7 Improving Energy Efficiency and Workflow Performance

With the growing emphasis on sustainable software engineering, a promising research direction is the analysis of energy consumption and computational footprint of GitHub Actions workflows. Workflow configurations—including job parallelism, dependency installation strategies, caching policies, and runner selection—directly affect both execution time and energy consumption. Empirical studies illustrate the scale and optimization potential of these costs: Bouzenia and Pradel (2024) found that CI/CD workflows can impose substantial computational and financial overhead, with testing and building accounting for more than 90% of runtime in paid-tier repositories. Even simple interventions, such as disabling scheduled workflows during inactive periods, can reduce workflow execution by up to 30%. Similarly, Ghaleb *et al.* (2024) observed that CI/CD maintenance occurs frequently, often on a bi-monthly basis, with a large proportion of changes focused on build improvements, dependency updates, and workflow corrections. Desai *et al.* (2025) demonstrated that prioritizing test execution based on commit-to-commit failure history can significantly reduce runtime, achieving average savings exceeding 80%, thereby highlighting the potential of intelligent, resource-aware workflow management.

Future studies could integrate historical workflow data with energy mea-

surement tools, telemetry, or runner-level monitoring to quantify environmental costs and identify performance–energy trade-offs. Such analyses would support energy-aware scheduling, more efficient dependency management, and recommendations for lower-impact workflow design. Additionally, combining empirical insights on commit frequency and maintenance patterns with optimization techniques such as selective test execution, caching, and workflow pruning can further improve sustainability without compromising developer productivity or workflow correctness. Research in this direction aligns with broader trends in green computing and offers tangible benefits for both open-source projects and commercial CI/CD pipelines, providing a foundation for environmentally conscious workflow engineering.

Our findings on workflow change frequency and lifecycle patterns provide a direct entry point for this research direction. The lifecycle analysis in Chapter 6 shows that workflow files are modified most intensively during the first weeks after adoption, with the proportion of lines of code modified decreasing from an average of 13.6% in the first six weeks to 10.2% over the following year. This early-stage instability, during which workflows are repeatedly revised and re-executed to reach a stable configuration, is likely also the phase with the highest computational waste, as practitioners reported pushing repeated commits simply to observe CI outcomes. Quantifying the energy and resource costs associated specifically with this early-stage trial-and-error behavior, and evaluating whether improved local testing tools or enhanced authoring support can mitigate it, would directly connect our lifecycle findings to sustainability concerns. Similarly, the burst patterns identified in Chapter 7 represent dense execution episodes that are well suited for targeted efficiency interventions.

8.3.8 Supporting Cross-Workflow Maintenance and Reuse

One underexplored aspect highlighted by our findings is the challenge of maintaining consistency across multiple workflow files within the same repository. Our manual analysis in Chapter 7 shows that 21.64% of commits in our sample modified multiple workflow files simultaneously. Among these, 68 out of 95 cases involved applying the same type of change across multiple workflows, most commonly version updates, branch renaming, or `run:` command modifications. While some of these changes were performed by automated tools such as Dependabot, the majority appear to have been carried out manually. This indicates that developers encounter a cross-workflow maintenance burden that is not adequately supported by existing tooling.

Future research could examine this cross-workflow maintenance problem more systematically, for example by mining the full dataset to determine

how frequently co-changes occur, which entity types are most commonly co-changed, and how the delay between when a change is required and when it is applied across all affected files relates to repository size or workflow complexity. From a tooling perspective, there is an opportunity to design workflow-aware refactoring tools that detect configuration duplication across workflow files within a repository and propose unified or parameterized alternatives, extending existing dependency automation such as Dependabot toward structural and semantic co-maintenance. The conceptual taxonomy developed in Chapter 7 could inform the design of such tools by providing a structured classification of change types that are suitable for cross-workflow synchronization.

By addressing these research directions, future work can deepen our understanding of how GitHub Actions workflows are authored, maintained, and automated in practice, and how they interact with the broader ecosystem of reusable Actions, dependency bots, and emerging AI-assisted development tools. Such research will be essential for helping developers and tool builders make informed decisions about workflow design and maintenance, and for ensuring that CI/CD practices remain reliable, secure, and understandable as the degree of automation within software development continues to grow.

Bibliography

- Abbott, M. L., & Fisher, M. T. (2015). *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise* (2nd edition). Addison-Wesley Professional.
- Abdalkareem, R., Mujahid, S., & Shihab, E. (2020). A Machine Learning Approach to Improve the Detection of CI Skip Commits. *Transactions on Software Engineering (TSE)*. <https://doi.org/10.1109/TSE.2020.2967380>
- Acharya, D. B., Kuppan, K., & Divya, B. (2025). Agentic AI: Autonomous Intelligence for Complex Goals—A Comprehensive Survey. *IEEE Access*, *13*, 18912–18936. <https://doi.org/10.1109/ACCESS.2025.3532853>
- Alfadel, M., Costa, D. E., Shihab, E., & Shihab, E. (2021). Empirical analysis of security vulnerabilities in Python packages. *Software Analysis, Evolution and Reengineering (SANER)*. <https://doi.org/10.1109/saner50967.2021.00048>
- Allam, H. (2025). Code Meets Intelligence: AI-Augmented CI/CD Systems for DevOps at Scale. *Artificial Intelligence, Data Science, and Machine Learning*, *6*(1), 137–146. <https://doi.org/10.63282/3050-9262.IJAIDS ML-V6I1P115>
- Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., Nagappan, N., Nushi, B., & Zimmermann, T. (2019). Software Engineering for Machine Learning: A Case Study. *International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 291–300. <https://doi.org/10.1109/ICSE-SEIP.2019.00042>
- Ampatzoglou, A., Bibi, S., Avgeriou, P., Verbeek, M., & Chatzigeorgiou, A. (2019). Identifying, Categorizing and Mitigating Threats to Validity in Software Engineering Secondary Studies. *Information and Software*

- Technology (IST)*, 106, 201–230. <https://doi.org/10.1016/j.infsof.2018.10.006>
- Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., & Zaharia, M. (2010). A View of Cloud Computing. *Communications of the ACM*, 53(4), 50–58. <https://doi.org/10.1145/1721654.1721672>
- Arntzenius, R., Liu, X., & Zaidman, A. (2026). On The Energy Consumption of Continuous Integration in Open-Source Java Projects. *Green Software Evolution Workshop (GreenVolve), co-located with Software Analysis, Evolution and Reengineering (SANER)*. <https://azaidman.github.io/publications/liuGreenvolve2026.pdf>
- Avelino, G., Constantinou, E., Valente, M. T., & Serebrenik, A. (2019). On the Abandonment and Survival of Open Source Projects: An Empirical Investigation. *Empirical Software Engineering and Measurement (ESEM)*, 1–12. <https://doi.org/10.1109/ESEM.2019.8870181>
- Bacchelli, A., & Bird, C. (2013). Expectations, Outcomes, and Challenges of Modern Code Review. *International Conference on Software Engineering (ICSE)*, 712–721. <https://doi.org/10.1109/ICSE.2013.6606617>
- Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., *et al.* (2001). *Manifesto for Agile Software Development* (tech. rep.). Snowbird, UT.
- Beller, M., Gousios, G., & Zaidman, A. (2017). Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub. *Mining Software Repositories (MSR)*, 356–367. <https://doi.org/10.1109/MSR.2017.62>
- Berliner, B. (1990). CVS II: Parallelizing Software Development. *USENIX Winter Technical Conference*, 341, 352.
- Bernard, H. R., Wutich, A., & Ryan, G. W. (2016). *Analyzing Qualitative Data: Systematic Approaches* (2nd edition). SAGE publications.
- Bernardo, J. H., da Costa, D. A., & Kulesza, U. (2018). Studying the Impact of Adopting Continuous Integration on the Delivery Time of Pull Requests. *Mining Software Repositories (MSR)*, 131–141. <https://doi.org/10.1145/3196398.3196421>
- Bertram, D., Volda, A., Greenberg, S., & Walker, R. (2010). Communication, Collaboration, and Bugs: the Social Nature of Issue Tracking in Small, Collocated Teams. *conference on Computer supported cooperative work*, 291–300. <https://doi.org/10.1145/1718918.1718972>

-
- Betz, R. M., & Walker, R. C. (2013). Implementing Continuous Integration Software in an Established Computational Chemistry Software Package. *Software Engineering for Computational Science and Engineering (SE-CSE)*, 68–74. <https://doi.org/10.1109/SECSE.2013.6615101>
- Bird, C., Ford, D., Zimmermann, T., Forsgren, N., Eirini, K., Lowdermilk, T., & Gazit, I. (2023). Taking Flight with Copilot: Early Insights and Opportunities of AI-powered Pair-programming Tools. *ACM Queue*, 20(6). <https://doi.org/10.1145/3582502>
- Bird, C., Rigby, P. C., Barr, E. T., Hamilton, D. J., German, D. M., & Devanbu, P. (2009). The Promises and Perils of Mining Git. *Mining Software Repositories (MSR)*, 1–10. <https://doi.org/10.1109/MSR.2009.5069475>
- Bouzenia, I., & Pradel, M. (2024). Resource Usage and Optimization Opportunities in Workflows of GitHub Actions. *International Conference on Software Engineering (ICSE)*. <https://doi.org/10.1145/3597503.3623303>
- Burmeister, E., & Aitken, L. M. (2012). Sample Size: How Many is Enough? *Australian Critical Care*, 25(4), 271–274. <https://doi.org/10.1016/j.aucc.2012.07.002>
- Cardoen, G., Mens, T., & Decan, A. (2024). A Dataset of GitHub Actions Workflow Histories. *Mining Software Repositories (MSR)*, 677–681. <https://doi.org/10.1145/3643991.3644867>
- Cardoen, G., Mens, T., & Decan, A. An Empirical Analysis of Code Clones in GitHub Actions Workflows. In: *Software analysis, evolution and reengineering (SANER)*. 2026. <https://benevol2025.github.io/pre/paper06.pdf>
- Catolino, G., Palomba, F., Tamburri, D. A., & Serebrenik, A. (2021). Understanding Community Smells Variability: A Statistical Approach. *International Conference on Software Engineering (ICSE)*, 77–86. <https://doi.org/10.1109/ICSE-SEIS52602.2021.00017>
- Chen, L. (2015). Continuous Delivery: Huge Benefits, but Challenges too. *IEEE Software*, 32(2), 50–54. <https://doi.org/10.1109/MS.2015.27>
- Chen, L. (2017). Continuous Delivery: Overcoming Adoption Challenges. *Journal of Systems and Software (JSS)*, 128, 72–86. <https://doi.org/10.1016/j.jss.2017.02.013>
- Chen, T., Zhang, Y., Chen, S., Wang, T., & Wu, Y. (2021). Let’s Supercharge the Workflows: An Empirical Study of GitHub Actions. *Software Quality, Reliability and Security Companion (QRS-C)*. <https://doi.org/10.1109/QRS-C55045.2021.00163>

- Chidambaram, N., Decan, A., & Mens, T. (2023). A Dataset of Bot and Human Activities in GitHub. *Mining Software Repositories (MSR)*, 465–469. <https://doi.org/10.1109/MSR59073.2023.00070>
- Chidambaram, N., Decan, A., & Mens, T. (2025). A Bot Identification Model and Tool Based on GitHub Activity Sequences. *Journal of Systems and Software*, 221, 112287. <https://doi.org/10.1016/j.jss.2024.112287>
- Cliff, N. (1993). Dominance Statistics: Ordinal Analyses to Answer Ordinal Questions. *Psychological bulletin*, 114(3), 494. <https://doi.org/10.1037/0033-2909.114.3.494>
- Cogo, F. R., Oliva, G. A., & Hassan, A. E. (2021). Deprecation of Packages and Releases in Software Ecosystems: A Case Study on npm. *Transactions on Software Engineering (TSE)*. <https://doi.org/10.1109/TSE.2021.3055123>
- Conradi, R., & Westfechtel, B. (1998). Version Models for Software Configuration Management. *ACM Computing Surveys (CSUR)*, 30(2), 232–282. <https://doi.org/10.1145/280277.280280>
- Constantinou, E., & Mens, T. (2017). An Empirical Comparison of Developer Retention in the RubyGems and npm Software Ecosystems. *Innovations in Systems and Software Engineering*, 13(2), 101–115. <https://doi.org/10.1007/s11334-017-0303-4>
- Costa, J. M., Cataldo, M., & de Souza, C. R. (2011). The Scale and Evolution of Coordination Needs in Large-scale Distributed Projects: Implications for the Future Generation of Collaborative Tools. *SIGCHI Conference on Human Factors in Computing Systems*, 3151–3160. <https://doi.org/10.1145/1978942.1979409>
- Dabbish, L., Stuart, C., Tsay, J., & Herbsleb, J. (2012). Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository. *Computer Supported Cooperative Work (CSCW)*, 1277–1286. <https://doi.org/10.1145/2145204.2145396>
- Dabic, O., Aghajani, E., & Bavota, G. (2021). Sampling Projects in GitHub for MSR Studies. *Mining Software Repositories (MSR)*, 560–564. <https://doi.org/10.1109/MSR52588.2021.00074>
- Danial, A. (2021). *CLOC* (Version v1.96). Zenodo. <https://doi.org/10.5281/zenodo.7455676>
- Decan, A., Mens, T., & Claes, M. (2017). An Empirical Comparison of Dependency Issues in OSS Packaging Ecosystems. *Software Analysis, Evolution and Reengineering (SANER)*. <https://doi.org/10.1109/SANER.2017.7884604>

- Decan, A., Mens, T., & Constantinou, E. (2018a). On the Evolution of Technical Lag in the npm Package Dependency Network. *International Conference on Software Maintenance and Evolution (ICSME)*, 404–414. <https://doi.org/10.1109/ICSME.2018.00050>
- Decan, A., Mens, T., & Constantinou, E. (2018b). On the Impact of Security Vulnerabilities in the npm Package Dependency Network. *Mining Software Repositories (MSR)*, 181–191. <https://doi.org/10.1145/3196398.3196401>
- Decan, A., Mens, T., & Onori Delicheh, H. (2023). On the Outdatedness of Workflows in the GitHub Actions Ecosystem. *Journal of Systems and Software (JSS)*, 206. <https://doi.org/10.1016/j.jss.2023.111827>
- Decan, A., Mens, T., Rostami Mazrae, P., & Golzadeh, M. (2022). On the Use of GitHub Actions in Software Development Repositories. *International Conference on Software Maintenance and Evolution (ICSME)*. <https://doi.org/10.1109/ICSME55016.2022.00029>
- Decan, A., Mens, T., Zerouali, A., & De Roover, C. (2021). Back to the Past—Analysing Backporting Practices in Package Dependency Networks. *Transactions on Software Engineering (TSE)*. <https://doi.org/10.1109/TSE.2021.3112204>
- Desai, S. V., Bhide, S., Serbout, S., Marchezan, L., & Assunção, W. K. G. (2025). PrioTestCI: Efficient Test Case Prioritization in GitHub Workflows for CI Optimization. *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 4021–4024. <https://doi.org/10.1109/ASE63991.2025.00365>
- Dietrich, J., Pearce, D., Stringer, J., Tahir, A., & Blincoe, K. (2019). Dependency Versioning in the Wild. *Mining Software Repositories (MSR)*, 349–359. <https://doi.org/10.1109/MSR.2019.00061>
- Durieux, T., Abreu, R., Monperrus, M., Bissyandé, T. F., & Cruz, L. (2019). An Analysis of 35+ Million Jobs of Travis CI. *International Conference on Software Maintenance and Evolution (ICSME)*, 291–295. <https://doi.org/10.1109/ICSME.2019.00044>
- Düsing, J., & Hermann, B. (2021). Analyzing the direct and transitive impact of vulnerabilities onto different artifact repositories. <https://doi.org/10.1145/3472811>
- Duvall, P. M., Matyas, S., & Glover, A. (2007). *Continuous Integration: Improving Software Quality and Reducing Risk*. Pearson Education.
- Elazhary, O., Storey, M.-A., Ernst, N., & Zaidman, A. (2019). Do as I Do, Not as I Say: Do Contribution Guidelines Match the GitHub Contribution

- Process? *International Conference on Software Maintenance and Evolution (ICSME)*, 286–290. <https://doi.org/10.1109/ICSME.2019.00043>
- Elazhary, O., Werner, C., Li, Z. S., Lowlind, D., Ernst, N. A., & Storey, M.-A. (2022). Uncovering the Benefits and Challenges of Continuous Integration Practices. *Transactions on Software Engineering (TSE)*, 48(7), 2570–2583. <https://doi.org/10.1109/TSE.2021.3064953>
- Erlenhov, L., de Oliveira Neto, F. G., Scandariato, R., & Leitner, P. (2019). Current and Future Bots in Software Development. *Bots in Software Engineering (BotSE)*, 7–11. <https://doi.org/10.1109/BotSE.2019.00009>
- Evtikhiev, M., Koshchenko, E., & Kovalenko, V. (2025). What Could Possibly Go Wrong: Undesirable Patterns in Collective Development. *Transactions on Software Engineering and Methodology*, 34(3). <https://doi.org/10.1145/3707451>
- Foundjem, A., Constantinou, E., Mens, T., & Adams, B. (2022). A Mixed-Methods Analysis of Micro-Collaborative Coding Practices in Open-Stack. *Empirical Software Engineering (EMSE)*, 27(5), 120. <https://doi.org/10.1007/s10664-022-10167-w>
- Freitas, M. B., & Rocha, L. S. (2024). GASH—The GitHub Actions Smell Hunter. *Workshop de Visualização, Evolução e Manutenção de Software (VEM)*, 103–112. <https://doi.org/10.5753/vem.2024.3911>
- Fusch, P. I., & Ness, L. R. (2015). Are we There Yet? Data Saturation in Qualitative Research. *The qualitative report*, 20(9).
- Gallaba, K., Junqueira, Y., Ewart, J., & McIntosh, S. (2022a). Accelerating Continuous Integration by Caching Environments and Inferring Dependencies. *Transactions on Software Engineering (TSE)*, 48(6), 2040–2052. <https://doi.org/10.1109/TSE.2020.3048335>
- Gallaba, K., Lamothe, M., & McIntosh, S. (2022b). Lessons from Eight Years of Operational Data from a Continuous Integration Service: An Exploratory Case Study of CircleCI. *International Conference on Software Engineering (ICSE)*, 1330–1342. <https://doi.org/10.1145/3510003.3510211>
- Gallaba, K., & McIntosh, S. (2020). Use and Misuse of Continuous Integration Features: An Empirical Study of Projects That (Mis)Use Travis CI. *Transactions on Software Engineering (TSE)*, 46(1), 33–50. <https://doi.org/10.1109/TSE.2018.2838131>
- Garrison, D. R., Cleveland-Innes, M., Koole, M., & Kappelman, J. (2006). Revisiting Methodological Issues in Transcript Analysis: Negotiated

- Coding and Reliability. *The internet and higher education*, 9(1), 1–8. <https://doi.org/10.1016/j.iheduc.2005.11.001>
- Ghaleb, T., Abduljalil, O., & Hassan, S. (2024). CI/CD Configuration Practices in Open-Source Android Apps: An Empirical Study. *Transactions on Software Engineering and Methodology (TOSEM)*. <https://doi.org/10.1145/3736758>
- Gmeiner, J., Ramler, R., & Haslinger, J. (2015). Automated Testing in the Continuous Delivery Pipeline: A Case Study of an Online Company. *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 1–6. <https://doi.org/10.1109/ICSTW.2015.7107423>
- Golzadeh, M., Decan, A., Legay, D., & Mens, T. (2021a). A Ground-Truth Dataset and Classification Model for Detecting Bots in GitHub Issue and PR Comments. *Journal of Systems and Software (JSS)*, 175. <https://doi.org/10.1016/j.jss.2021.110911>
- Golzadeh, M., Decan, A., & Mens, T. (2021b). Evaluating a Bot Detection Model on Git Commit Messages. *CEUR Workshop Proceedings*, 2912. <https://doi.org/10.48550/arXiv.2103.11779>
- Golzadeh, M., Decan, A., & Mens, T. (2021c). On the Rise and Fall of CI Services in GitHub. *Software Analysis, Evolution and Reengineering (SANER)*. <https://doi.org/10.1109/SANER53432.2022.00084>
- Gousios, G., Pinzger, M., & van Deursen, A. (2014). An Exploratory Study of the Pull-Based Software Development Model. *International Conference on Software Engineering (ICSE)*, 345–355. <https://doi.org/10.1145/2568225.2568260>
- Guest, G., Bunce, A., & Johnson, L. (2006). How Many Interviews are Enough? An Experiment with Data Saturation and Variability. *Field Methods*, 18(1), 59–82. <https://doi.org/10.1177/1525822X05279903>
- Gupta, M. L., Puppala, R., Vadapalli, V. V., Gundu, H., & Karthikeyan, C. V. S. S. (2024). Continuous Integration, Delivery and Deployment: A Systematic Review of Approaches, Tools, Challenges and Practices. *Recent Trends in AI Enabled Technologies*, 76–89. https://doi.org/10.1007/978-3-031-59114-3_7
- Gupta, Y., Khan, Y., Gallaba, K., & McIntosh, S. (2017). The Impact of the Adoption of Continuous Integration on Developer Attraction and Retention. *Mining Software Repositories (MSR)*, 491–494. <https://doi.org/10.1109/MSR.2017.37>

- Halperin, M., Lan, K. G., & Hamdy, M. I. (1988). Some Implications of an Alternative Definition of the Multiple Comparison Problem. *Biometrika*, 75(4), 773–778. <https://doi.org/10.1093/biomet/75.4.773>
- Herbsleb, J. D. (2007). Global Software Engineering: The Future of Socio-technical Coordination. *International Conference on Software Engineering (ISCE)*, 188–198. <https://doi.org/10.1109/FOSE.2007.11>
- Hilton, M., Nelson, N., Tunnell, T., Marinov, D., & Dig, D. (2017). Trade-offs in Continuous Integration: Assurance, Security, and Flexibility. *European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE)*, 197–207. <https://doi.org/10.1145/3106237.3106270>
- Hilton, M., Tunnell, T., Huang, K., Marinov, D., & Dig, D. (2016). Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects. *Automated Software Engineering (ASE)*, 426–437. <https://doi.org/10.1145/2970276.2970358>
- Holm, S. (1979). A Simple Sequentially Rejective Multiple Test Procedure. *Scandinavian Journal of Statistics*, 6(2), 65–70. <http://www.jstor.org/stable/4615733>
- Holmström, H., Conchúir, E. Ó., Ågerfalk, P. J., & Fitzgerald, B. (2006). Global Software Development Challenges: A Case Study on Temporal, Geographical and Socio-Cultural Distance. *International Conference on Global Software Engineering (ICGSE)*, 3–11. <https://doi.org/10.1109/ICGSE.2006.261210>
- Huang, J., & Lin, B. (2023). CIGAR: Contrastive Learning for GitHub Action Recommendation. *Conference on Source Code Analysis and Manipulation (SCAM)*, 61–71. <https://doi.org/10.1109/SCAM59687.2023.00017>
- Huang, J., & Lin, B. (2026). On the Reruns of GitHub Actions Workflows [Just Accepted]. *Transactions on Software Engineering and Methodology (TOSEM)*. <https://doi.org/10.1145/3795771>
- Jin, X., Feng, Y., Wang, C., Liu, Y., Hu, Y., Gao, Y., Xia, K., & Guo, L. (2024). PIPELINEASCODE: A CI/CD Workflow Management System through Configuration Files at ByteDance. *Software Analysis, Evolution and Reengineering (SANER)*, 1011–1022. <https://doi.org/10.1109/SANER60148.2024.00109>
- Jin, X., & Servant, F. (2021). What Helped, and What Did Not? An Evaluation of the Strategies to Improve Continuous Integration. *International Conference on Software Engineering (ICSE)*, 213–225. <https://doi.org/10.1109/ICSE43902.2021.00031>

-
- Jin, X., & Servant, F. (2022). Which Builds are Really Safe to Skip? Maximizing Failure Observation for Build Selection in Continuous Integration. *Journal of Systems and Software (JSS)*, 188. <https://doi.org/10.1016/j.jss.2022.111292>
- Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., & Damian, D. (2014). The Promises and Perils of Mining GitHub. *Mining Software Repositories (MSR)*, 92–101. <https://doi.org/10.1145/2597073.2597074>
- Kaur, R., & Kaur, K. (2022). Insights into Developers' Abandonment in FLOSS Projects. *Intelligent Sustainable Systems*, 333. https://doi.org/10.1007/978-981-16-6309-3_69
- Khatami, A., Brandt, C., & Zaidman, A. (2026). Beyond the YAML File: Understanding Real-World GitHub Actions Workflow Adoption. *Evaluation and Assessment in Software Engineering (EASE)*. <https://doi.org/10.5281/zenodo.18258226>
- Khatami, A., Willekens, C., & Zaidman, A. (2024). Catching Smells in the Act: A GitHub Actions Workflow Investigation. *Source Code Analysis and Manipulation (SCAM)*. <https://doi.org/10.1109/SCAM63643.2024.00015>
- Khelifi, J., Benzina, Y., Chouchen, M., Ouni, A., Sayagh, M., & Bouktif, S. (2025). GHAMiner: An Open Source Tool to Extract GitHub Actions Build Metrics. *Software Analysis, Evolution and Reengineering (SANER)*, 834–838. <https://doi.org/10.1109/SANER64311.2025.00087>
- Kim, M., Zimmermann, T., DeLine, R., & A., B. (2016). The Emerging Role of Data Scientists on Software Development Teams. *International conference on software engineering (ICSE)*, 96–107. <https://doi.org/10.1145/2884781.2884783>
- Kinsman, T., Wessel, M., Gerosa, M. A., & Treude, C. (2021). How do Software Developers Use GitHub Actions to Automate their Workflows? *Mining Software Repositories (MSR)*, 420–431. <https://doi.org/10.1109/MSR52588.2021.00054>
- Koc, A., & Tansel, A. U. (2011). A Survey of Version Control Systems. *ICEME*. https://www.iis.org/cds2011/cd2011imc/iceme_2011/paperspdf/fb394vz.pdf
- Kula, R. G., German, D. M., Ouni, A., Ishio, T., & Inoue, K. (2018). Do Developers Update Their Library Dependencies? *Empirical Software Engineering (EMSE)*, 23(1), 384–417. <https://doi.org/10.1007/s10664-017-9521-5>

- Kulas, M., Borelli, J. L., Gässler, W., Peter, D., Rabien, S., de Xivry, G. O., Busoni, L., Bonaglia, M., Mazzoni, T., & Rahmer, G. (2014). Practical Experience with Test-Driven Development during Commissioning of the Multi-Star AO System ARGOS. *Software and Cyberinfrastructure for Astronomy III*, 9152, 110–119. <https://doi.org/10.1117/12.2056218>
- Kurose, J. F., & Ross, K. W. (2013). *Computer Networking: A Top-Down Approach* (6th). Pearson.
- Lehman, M. M. (1996). Laws of Software Evolution Revisited. *European Workshop on Software Process Technology (EWPST)*, 108–124. <https://doi.org/10.1007/BFb0017737>
- Leppänen, M., Mäkinen, S., Pagels, M., Eloranta, V.-P., Itkonen, J., Mäntylä, M. V., & Männistö, T. (2015). The Highways and Country Roads to Continuous Deployment. *IEEE Software*, 32(2), 64–72. <https://doi.org/10.1109/MS.2015.50>
- Li, Z., Liu, W., Chen, H., Wang, X., Liao, X., Xing, L., Zha, M., Jin, H., & Zou, D. (2022). Robbery on DevOps: Understanding and Mitigating Illicit Cryptomining on Continuous Integration Service Platforms. *Symposium on Security and Privacy (SP)*, 2397–2412. <https://doi.org/10.1109/SP46214.2022.9833803>
- Lu, J., Yang, Z., & Qian, J. (2014). Implementation of Continuous Integration and Automated Testing in Software Development of Smart Grid Scheduling Support System. *International Conference on Power System Technology*, 2441–2446. <https://doi.org/10.1109/POWERCON.2014.6993503>
- Maddila, C. (2022). *Recommender Systems for DevOps* (Doctoral dissertation). University of Delft. <https://doi.org/10.4233/uuid:769d3d81-8a84-4f59-80a6-2d237aa878a4>
- Mann, H. B., & Whitney, D. R. (1947). On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *Annals Mathematical Statistics*, 18(1), 50–60. <https://doi.org/10.1214/aoms/1177730491>
- Mastropaolo, A., Zampetti, F., Bavota, G., & Di Penta, M. (2024). Toward Automatically Completing GitHub Workflows. *International Conference on Software Engineering (ICSE)*. <https://doi.org/10.1145/3597503.3623351>
- Maxwell, J. (1992). Understanding and Validity in Qualitative Research. *Harvard educational review*, 62(3), 279–301. <https://doi.org/10.17763/haer.62.3.8323320856251826>

-
- Melo, G. (2025). Opportunities and Challenges of Software Engineering Bots: A Forward-Looking Analysis. *Bots in Software Engineering (BotSE)*, 28–32. <https://doi.org/10.1109/BotSE67031.2025.00014>
- Meyer, A., Barr, E., Bird, C., & Zimmermann, T. . (2019). Today was a Good Day: The Daily Life of Software Developers. *Transactions on Software Engineering (TSE)*, 47(5), 863–880. <https://doi.org/10.1109/TSE.2019.2904957>
- Mohayjeji, H., Agaronian, A., Constantinou, E., Zannone, N., & Serebrenik, A. (2023). Investigating the Resolution of Vulnerable Dependencies with Dependabot Security Updates. *Mining Software Repositories (MSR)*, 234–246. <https://doi.org/10.1109/MSR59073.2023.00042>
- Mokhov, A., Mitchell, N., & Peyton Jones, S. (2018). Build Systems à La Carte. *Proceedings of the ACM on Programming Languages*, 2(ICFP). <https://doi.org/10.1145/3236774>
- Moriconi, F., Durieux, T., Falleri, J.-R., Troncy, R., & Francillon, A. (2025). GHALogs: Large-Scale Dataset of GitHub Actions Runs. *Mining Software Repositories (MSR)*, 669–673. <https://doi.org/10.1109/MSR66628.2025.00104>
- Nagappan, N., Zeller, A., Zimmermann, T., Herzig, K., & Murphy, B. (2010). Change Bursts as Defect Predictors. *International Symposium on Software Reliability Engineering (ISSRE)*, 309–318. <https://doi.org/10.1109/ISSRE.2010.25>
- Onsori Delicheh, H., Decan, A., & Mens, T. (2024). Quantifying Security Issues in Reusable JavaScript Actions in GitHub Workflows. *Mining Software Repositories (MSR)*, 692–703. <https://doi.org/10.1145/3643991.3644899>
- Onsori Delicheh, H., & Mens, T. (2024). Mitigating Security Issues in GitHub Actions. *Engineering and Cybersecurity of Critical Systems (EnCy-CriS)*, 6–11. <https://doi.org/10.1145/3643662.3643961>
- O’Sullivan, B. (2009). *Mercurial: The Definitive Guide: The Definitive Guide*. " O’Reilly Media, Inc.". <https://svn.0x00ff00ff.com/mirror/http/bitbucket.org/tortoisehg/thg/downloads/tortoisehg-2.3.1-hg-2.1.1-x64.msi/program/doc/hgbook.pdf>
- Pilato, C. M., Collins-Sussman, B., & Fitzpatrick, B. W. (2008). *Version Control with Subversion* (2nd edition). O’Reilly Media. <https://ceres.larc.nasa.gov/documents/subversion-book.pdf>
- Rahman, A., Agrawal, A., Krishna, R., & Sobran, A. (2018). Characterizing the Influence of Continuous Integration: Empirical Results from 250+ Open Source and Proprietary Projects. *SIGSOFT International Work-*

- shop on Software Analytics*, 8–14. <https://doi.org/10.1145/3278142.3278149>
- Ralph, P., & Tempero, E. (2018). Construct Validity in Software Engineering Research and Software Metrics. *Evaluation and Assessment in Software Engineering (EASE)*, 13–23. <https://doi.org/10.1145/3210459.3210461>
- Riggio, E., & Pautasso, C. (2025). Pipelines Under Pressure: An Empirical Study of Security Misconfigurations of GitHub Workflows. *Product-Focused Software Process Improvement*, 220–236. https://doi.org/10.1007/978-3-032-12089-2_14
- Robbes, R., Matricon, T., Degueule, T., Hora, A., & Zacchiroli, S. Promises, Perils, and (Timely) Heuristics for Mining Coding Agent Activity. In: *Mining software repositories (MSR)*. 2026. <https://doi.org/10.48550/arXiv.2601.18345>.
- Rochkind, M. J. (1975). The Source Code Control System. *Transactions on Software Engineering (TSE)*, SE-1(4), 364–370. <https://doi.org/10.1109/TSE.1975.6312866>
- Rochkind, M. J. (2025). A Retrospective on the Source Code Control System. *Transactions on Software Engineering (TSE)*, 51(3), 695–699. <https://doi.org/10.1109/TSE.2024.3524947>
- Romano, J., Kromrey, J. D., Coraggio, J., Skowronek, J., & Devine, L. (2006). Exploring Methods for Evaluating Group Differences on the NSSE and Other Surveys: Are the t-Test and Cohen’s d Indices the Most Appropriate Choices? *Annual Meeting of the Southern Association for Institutional Research*.
- Rostami Mazrae, P., Decan, A., Mens, T., & Wessel, M. (2023). A Preliminary Study of GitHub Actions Workflow Changes. *Proceedings of the Belgium-Netherlands Software Evolution Workshop (BENEVOL)*. <https://ceur-ws.org/Vol-3483/paper8.pdf>
- Roundy, D. (2005). Darcs: Distributed Version Management in Haskell. *ACM SIGPLAN Workshop on Haskell*, 1–4. <https://doi.org/10.1145/1088348.1088349>
- Saavedra, N., Silva, A., & Monperrus, M. (2024). GitBug-Actions: Building Reproducible Bug-Fix Benchmarks with GitHub Actions. *International Conference on Software Engineering (ICSE)*. <https://doi.org/10.1145/3639478.3640023>
- Saidani, I., Ouni, A., Mkaouer, M. W., & Palomba, F. (2021). On the Impact of Continuous Integration on Refactoring Practice: An Exploratory Study on TravisTorrent. *Information and Software Technology (IST)*, 138, 106618. <https://doi.org/10.1016/j.infsof.2021.106618>

-
- Santos, J., da Costa, D. A., & Kulesza, U. (2022). Investigating the Impact of Continuous Integration Practices on the Productivity and Quality of Open-Source Projects. *Empirical Software Engineering and Measurement (ESEM)*, 137–147. <https://doi.org/10.1145/3544902.3546244>
- Santos, J., da Costa, D. A., & Kulesza, U. (2025a). Monitoring Continuous Integration Practices in Industry: A Case Study. *International Conference on Software Maintenance and Evolution (ICSME)*, 721–731. <https://doi.org/10.1109/ICSME64153.2025.00072>
- Santos, J., da Costa, D. A., McIntosh, S., & Kulesza, U. (2025b). On the Need to Monitor Continuous Integration Practices. *Empirical Software Engineering (EMSE)*, 30(5), 125. <https://doi.org/10.1007/s10664-025-10682-6>
- Saroar, S. G., & Nayebi, M. (2023). Developers' Perception of GitHub Actions: A Survey Analysis. *Evaluation and Assessment in Software Engineering (EASE)*. <https://doi.org/10.1145/3593434.3593475>
- Savor, T., Douglas, M., Gentili, M., Williams, L., Beck, K., & Stumm, M. (2016). Continuous Deployment at Facebook and OANDA. *International Conference on Software Engineering (ICSE)*, 21–30. <https://doi.org/10.1145/2889160.2889223>
- Schneider, J., Meske, C., & Kuss, P. (2024). Foundation Models: A New Paradigm for Artificial Intelligence. *Business & Information Systems Engineering*, 66(2), 221–231. <https://doi.org/10.1007/s12599-024-00851-0>
- Shahin, A., Babar, M., & Zhu, L. (2017). Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. *IEEE Access*, 5, 3909–3943. <https://doi.org/10.1109/ACCESS.2017.2685629>
- Soares, E., Sizilio, G., Santos, J., da Costa, D. A., & Kulesza, U. (2022). The Effects of Continuous Integration on Software Development: a Systematic Literature Review. *Empirical Software Engineering (EMSE)*, 27(3), 1–61. <https://doi.org/10.1007/s10664-021-10114-1>
- Song, F., Agarwal, A., & Wen, W. (2024). The Impact of Generative AI on Collaborative Open-Source Software Development: Evidence from GitHub Copilot. *arXiv*. <https://doi.org/10.48550/arXiv.2410.02091>
- Soto-Valero, C., Harrand, N., Monperrus, M., & Baudry, B. (2021). A comprehensive study of bloated dependencies in the Maven ecosystem. *Empirical Software Engineering (EMSE)*, 26(3), 45. <https://doi.org/10.1007/s10664-020-09914-8>

- Spencer, L., Ritchie, J., Lewis, J., Dillon, L., *et al.* (2004). Quality in Qualitative Evaluation: A Framework for Assessing Research Evidence.
- Spinellis, D. (2012). Git. *IEEE Software*, 29(3), 100–101. <https://doi.org/10.1109/MS.2012.61>
- Ståhl, D., & Bosch, J. (2013). Experienced Benefits of Continuous Integration in Industry Software Product Development: A Case Study. *IASTED International Conference on Software Engineering*, 736–743.
- Storey, M.-A., Treude, C., Van Deursen, A., & Cheng, L.-T. (2010). The Impact of Social Media on Software Engineering Practices and Tools. *FSE/SDP workshop on Future of software engineering research (FoSER)*, 359–364. <https://doi.org/10.1145/1882362.1882435>
- Suárez-Brieva, E., Pardo Calvache, C. J., & Pérez-Castillo, R. (2026). Social Debt in Software Development Environments: A Systematic Literature Review. *Science of Computer Programming*, 249, 103396. <https://doi.org/10.1016/j.scico.2025.103396>
- Tambon, F., Moradi-Dakhel, A., Nikanjam, A., Khomh, F., Desmarais, M. C., & Antoniol, G. (2025). Bugs in Large Language Models Generated Code: An Empirical Study. *Empirical Software Engineering (EMSE)*, 30(3), 65. <https://doi.org/10.1007/s10664-025-10614-4>
- Tichy, W. F. (1985). RCS — a system for version control. *Software: Practice and Experience*, 15(7), 637–654. <https://doi.org/10.1002/spe.4380150703>
- Treude, C., & Poskitt, C. M. (2025). Bot-Driven Development: From Simple Automation to Autonomous Software Development Bots. *Bots in Software Engineering (BotSE)*, 18–22. <https://doi.org/10.1109/BotSE67031.2025.00012>
- Tsay, J., Dabbish, L., & Herbsleb, J. (2014). Influence of Social and Technical Factors for Evaluating Contribution in GitHub. *International Conference on Software Engineering (ICSE)*, 356–366. <https://doi.org/10.1145/2568225.2568315>
- Ulfesnes, R., Moe, N. B., Stray, V., & Skarpen, M. (2024). *Transforming Software Development with Generative AI: Empirical Insights on Collaboration and Workflow*. Springer. https://doi.org/10.1007/978-3-031-55642-5_10
- Valenzuela-Toledo, P., & Bergel, A. (2022). Evolution of GitHub Action Workflows. *Software Analysis, Evolution and Reengineering (SANER)*, 123–127. <https://doi.org/10.1109/SANER53432.2022.00026>
- Valenzuela-Toledo, P., Bergel, A., Kehrer, T., & Nierstrasz, O. The Hidden Costs of Automation: An Empirical Study on GitHub Actions Workflow

-
- Maintenance. In: *Source code analysis and manipulation (scam)*. IEEE. 2024. <https://doi.org/10.1109/SCAM63643.2024.00029>.
- Vasilescu, B., Yu, Y., Wang, H., Devanbu, P., & Filkov, V. (2015). Quality and Productivity Outcomes Relating to Continuous Integration in GitHub. *European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE)*, 805–816. <https://doi.org/10.1145/2786805.2786850>
- Vassallo, C., Palomba, F., & Gall, H. C. (2018). Continuous Refactoring in CI: A Preliminary Study on the Perceived Advantages and Barriers. *International Conference on Software Maintenance and Evolution (ICSME)*, 564–568. <https://doi.org/10.1109/ICSME.2018.00068>
- Vassallo, C., Proksch, S., Gall, H. C., & Di Penta, M. (2019). Automated Reporting of Anti-patterns and Decay in Continuous Integration. *International Conference on Software Engineering (ICSE)*, 105–115. <https://doi.org/10.1109/ICSE.2019.00028>
- Vassallo, C., Proksch, S., Jancso, A., Gall, H. C., & Di Penta, M. (2020). Configuration smells in continuous delivery pipelines: a linter and a six-month study on GitLab. *European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE)*, 327–337. <https://doi.org/10.1145/3368089.3409709>
- Vemuri, N., Thaneeru, N., & Tatikonda, V. M. (2024). AI-optimized DevOps for streamlined cloud CI/CD. *Innovative Science and Research Technology (ISRT)*, 9(7), 10–5281. <https://doi.org/10.5281/zenodo.10673085>
- Verdecchia, R., Cruciani, E., Bertolino, A., & Miranda, B. (2025). Energy-Aware Software Testing. *International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, 101–105. <https://doi.org/10.1109/ICSE-NIER66352.2025.00026>
- Wang, Z., Wang, Y., & Redmiles, D. (2022). From Specialized Mechanics to Project Butlers: the Usage of Bots in OSS Development. *IEEE Software*. <https://doi.org/10.1109/MS.2022.3180297>
- Wang, Z., Wang, Y., & Redmiles, D. (2023). Optimizing Workflow for Elite Developers: Perspectives on Leveraging SE Bots. *Bots in Software Engineering (BotSE)*, 23–27. <https://doi.org/10.1109/BotSE59190.2023.00013>
- Wessel, M., Wiese, I., Steinmacher, I., & Gerosa, M. A. (2021). Don't Disturb Me: Challenges of Interacting with Software Bots on Open Source Software Projects. *Human-Computer Interaction (HCI)*. <https://doi.org/10.1145/3476042>

- Widder, D., Hilton, M., Kästner, C., & Vasilescu, B. (2019). A Conceptual Replication of Continuous Integration Pain Points in the Context of Travis CI. *European Software Engineering Conference and Foundations of Software Engineering (ESEC/FSE)*, 647–658. <https://doi.org/10.1145/3338906.3338922>
- Widder, D., Vasilescu, B., Hilton, M., & Kästner, C. (2018). I’m Leaving You, Travis: a Continuous Integration Breakup Story. *Mining Software Repositories (MSR)*, 165–169. <https://doi.org/10.1145/3196398.3196422>
- Williams, L., Benedetti, G., Hamer, S., Paramitha, R., Rahman, I., Tamanna, M., Tystahl, G., Zahan, N., Morrison, P., Acar, Y., *et al.* (2025). Research Directions in Software Supply Chain Security. *Transactions on Software Engineering and Methodology (TOSEM)*, 34(5), 1–38. <https://doi.org/10.1145/3714464>
- Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., & Wesslén, A. (2012). *Experimentation in software engineering*. Springer.
- Wróbel, M. R., Szymukowicz, J., & Weichbroth, P. (2023). Using Continuous Integration Techniques in Open Source Projects—An Exploratory Study. *IEEE Access*, 11, 113848–113863. <https://doi.org/10.1109/ACCESS.2023.3324536>
- Zaidman, A. (2024). An Inconvenient Truth in Software Engineering? The Environmental Impact of Testing Open Source Java Projects. *Automation of Software Test (AST)*, 214–218. <https://doi.org/10.1145/3644032.3644461>
- Zampetti, F., Geremia, S., Bavota, G., & Di Penta, M. (2021). CI/CD Pipelines Evolution and Restructuring: A Qualitative and Quantitative Study. *International Conference on Software Maintenance and Evolution (IC-SME)*. <https://doi.org/10.1109/ICSME52107.2021.00048>
- Zampetti, F., Scalabrino, S., Oliveto, R., Canfora, G., & Di Penta, M. (2017). How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines. *Mining Software Repositories (MSR)*, 334–344. <https://doi.org/10.1109/MSR.2017.2>
- Zampetti, F., Vassallo, C., Panichella, S., Canfora, G., Gall, H., & Di Penta, M. (2020). An Empirical Characterization of Bad Practices in Continuous Integration. *Empirical Software Engineering (EMSE)*, 25, 1095–1135. <https://doi.org/10.1007/s10664-019-09785-8>
- Zerouali, A., Mens, T., Decan, A., & De Roover, C. (2022). On the Impact of Security Vulnerabilities in the npm and RubyGems Dependency Net-

-
- works. *Empirical Software Engineering (EMSE)*, 27(5), 1–45. <https://doi.org/10.1007/s10664-022-10154-1>
- Zhang, X., Muralee, S., Cherupattamoolayil, S., & Machiry, A. (2024). On the Effectiveness of Large Language Models for GitHub Workflows. *Availability, Reliability and Security (ARES)*. <https://doi.org/10.1145/3664476.3664497>
- Zheng, L., Li, S., Huang, X., Huang, J., Lin, B., Chen, J., & Xuan, J. (2025). Why Do GitHub Actions Workflows Fail? An Empirical Study. *Transactions on Software Engineering and Methodology (TOSEM)*. <https://doi.org/10.1145/3749371>
- Zhu, H.-N., Guan, K. Z., Furth, R. M., & Rubio-González, C. (2023). Action-remaker: Reproducing GitHub Actions. *International Conference on Software Engineering (ICSE)*, 11–15. <https://doi.org/10.1109/ICSE-Companion58688.2023.00015>
- Zimmermann, M., Staicu, C.-A., Tenny, C., & Pradel, M. (2019). Small World with High Risks: A Study of Security Threats in the npm Ecosystem. *USENIX Security Symposium*, 995–1010. <https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman>

Webography

- Anthropic. (2023). *Introducing Claude* [Accessed February 2026]. <https://www.anthropic.com/news/introducing-claude>
- Atlassian. (2023). *Jenkins vs Travis CI* [Accessed January 2026]. <https://www.atlassian.com/continuous-delivery/ci-vs-cd/tools/jenkins-vs-travis>
- Atlassian. (2025a). *Bamboo Continuous Integration and Delivery* [Accessed January 2026]. <https://www.atlassian.com/software/bamboo/features>
- Atlassian. (2025b). *Bitbucket Pipelines* [Accessed January 2026]. <https://www.atlassian.com/software/bitbucket/features/pipelines>
- Atlassian. (2025c). *Introducing Dynamic Pipelines: A New Standard in CI/CD Flexibility* [Accessed January 2026]. Retrieved April 23, 2024, from <https://www.atlassian.com/bitbucket/blog/introducing-dynamic-pipelines-a-new-standard-in-ci-cd-flexibility>
- Atlassian. (2025d). *Sharing CI/CD Configuration in Bitbucket Pipelines* [Accessed January 2026]. Retrieved August 24, 2023, from <https://www.atlassian.com/blog/bitbucket/share-ci-config>
- BitMover. (2000). *BitKeeper: Distributed Revision Control*. <https://www.bitkeeper.org>
- Canonical. (2005). *Bazaar version control system* [Accessed: 2025]. <https://bazaar.canonical.com>
- CircleCI. (2023a). *A Brief History of DevOps, Part III: Automated Testing and Continuous Integration* [Accessed: 2026-02-11]. <https://circleci.com/blog/a-brief-history-of-devops-part-iii-automated-testing-and-continuous-integration/>
- CircleCI. (2023b). *About CircleCI* [Accessed January 2026]. <https://circleci.com/about/>

- CircleCI. (2023c). *CI/CD for Microservices* [Accessed January 2026]. <https://circleci.com/microservices/>
- CircleCI. (2023d). *CircleCI Company History* [Accessed January 2026]. <https://circleci.com/company/>
- CircleCI Documentation. (2023). *Deploying to Cloud Providers with CircleCI* [Accessed January 2026]. <https://circleci.com/docs/deployment/>
- CruiseControl Project. (2001). *CruiseControl Continuous Integration Tool* [Accessed: 2026-02-11]. <https://cruisecontrol.sourceforge.net/index.html>
- Curate Partners. (2025). *DevOps with Bamboo: Integrating CI/CD for Efficient Delivery* [Accessed January 2026]. <https://tinyurl.com/bamboo-cicd>
- Drone.io. (2025a). *Drone — Container Native Continuous Delivery* [Accessed January 2026]. <https://drone.io/>
- Drone.io. (2025b). *Drone CI/CD Documentation* [Accessed January 2026]. <https://docs.drone.io/>
- Eclipse Foundation. (2025). *Hudson CI: Meet Hudson* [Accessed January 2026]. https://wiki.eclipse.org/Hudson-ci/Meet_Hudson
- Fowler, M., & Foemmel, M. (2000). *Continuous Integration (Original Version)* [Accessed 2023-04-15]. Retrieved April 15, 2023, from <https://martinfowler.com/articles/originalContinuousIntegration.html>
- GenAI. (2024). *2025 Top 10 Risk And Mitigations for LLMs and Gen AI Apps*. Retrieved January 1, 2026, from <https://genai.owasp.org/llm-top-10/>
- GenAI. (2025). *Agentic AI Threats and Mitigations*. Retrieved January 1, 2026, from <https://genai.owasp.org/resource/agentic-ai-threats-and-mitigations/>
- GitHub. (2021). *Introducing GitHub Copilot AI pair programmer* [Accessed February 2026]. <https://github.blog/news-insights/product-news/introducing-github-copilot-ai-pair-programmer/>
- GitHub. (2025a). *About GitHub Apps* [Accessed January 2026]. Retrieved January 1, 2026, from <https://docs.github.com/en/apps/overview>
- GitHub. (2025b). *About GitHub CLI* [Accessed January 2026]. Retrieved January 1, 2026, from <https://docs.github.com/en/github-cli/github-cli/about-github-cli>
- GitHub. (2025c). *About the REST API* [Accessed January 2026]. Retrieved January 1, 2026, from <https://docs.github.com/en/rest/about-the-rest-api/about-the-rest-api?apiVersion=2022-11-28>
- GitHub. (2025d). *About using GitHub Apps* [Accessed January 2026]. Retrieved January 1, 2026, from <https://docs.github.com/en/apps/using-github-apps/about-using-github-apps>

-
- GitHub. (2025e). *About Webhooks* [Accessed January 2026]. Retrieved January 1, 2026, from <https://docs.github.com/en/webhooks/about-webhooks>
- GitHub. (2025f). *Comparing GitHub Actions and GitHub Apps* [Accessed January 2026]. Retrieved January 1, 2026, from <https://docs.github.com/en/actions/get-started/actions-vs-apps>
- GitHub. (2025g). *Installing a GitHub App from GitHub Marketplace for your organization* [Accessed January 2026]. Retrieved January 1, 2026, from <https://docs.github.com/en/apps/using-github-apps/installing-a-github-app-from-github-marketplace-for-your-organizations>
- GitHub. (2025h). *Introduction to GraphQL* [Accessed January 2026]. Retrieved January 1, 2026, from <https://docs.github.com/en/graphql/guides/introduction-to-graphql>
- GitHub. (2025i). *Release of CoPilot*. Retrieved January 1, 2026, from <https://github.com/newsroom/press-releases/coding-agent-for-github-copilot>
- GitHub. (2025j). *Using GraphQL Clients* [Accessed January 2026]. Retrieved January 1, 2026, from <https://docs.github.com/en/graphql/guides/using-graphql-clients>
- GitHub. (2025k). *Webhook Events and Payloads* [Accessed January 2026]. Retrieved January 1, 2026, from <https://docs.github.com/en/webhooks/webhook-events-and-payloads>
- GitHub. (2026). *Introducing Editor Support for Action Files* [Accessed February 2026]. <https://github.blog/changelog/2026-01-29-github-actions-smarter-editing-clearer-debugging-and-a-new-case-function/#editor-support-for-action-yml-files>
- GitHub Docs. (2023). *Building and Testing with Travis CI* [Accessed January 2026]. <https://docs.github.com/en/actions/migrating-to-github-actions/migrating-from-travis-ci-to-github-actions>
- GitHub Next. (2025). *GitHub Agentic Workflows (gh-aw)* [Accessed: 2026-02-11]. <https://github.github.com/gh-aw/>
- GitLab. (2023a). *GitLab CI/CD* [Accessed January 2026]. <https://docs.gitlab.com/ee/ci/>
- GitLab. (2023b). *The DevOps Platform* [Accessed January 2026]. <https://about.gitlab.com/platform/>
- JetBrains. (2025a). *Continuous Integration with TeamCity* [Accessed January 2026]. Retrieved April 7, 2025, from <https://www.jetbrains.com/help/teamcity/continuous-integration-with-teamcity.html>
- JetBrains. (2025b). *TeamCity Feature Comparison* [Accessed January 2026]. https://resources.jetbrains.com/storage/products/teamcity/docs/Comparisons_TeamCity.pdf

- JetBrains. (2026). *TeamCity*. Retrieved February 11, 2026, from <https://www.jetbrains.com/teamcity/>
- Kawaguchi, K. (2007). *Hudson*. Retrieved February 11, 2026, from <https://web.archive.org/web/20140701020639/https://www.java.net//blog/kohsuke/archive/20070514/Hudson%20J1.pdf>
- Machiraju, Vijay. (2021). *Change in Azure Pipelines Grant for Public Projects* [Accessed: 14.10.2022]. Retrieved October 14, 2022, from <https://devblogs.microsoft.com/devops/change-in-azure-pipelines-grant-for-public-projects/>
- McVoy, L. (1998). *A solution for growing pains*. <https://lkml.iu.edu/hypermail/linux/kernel/9809.3/0957.html>
- Mendy, M., Rios, N., & Rybinski, M. (2020). *The New Pricing Model for travisci.com* [Accessed: 14.10.2022]. <https://blog.travis-ci.com/2020-11-02-travis-ci-new-billing>
- Microsoft Azure. (2025). *Azure DevOps Pipelines* [Accessed January 2026]. <https://azure.microsoft.com/en-au/products/devops/pipelines/>
- Microsoft DevOps Blog. (2025). *What is New with Azure Pipelines* [Accessed January 2026]. Retrieved May 19, 2019, from <https://devblogs.microsoft.com/devops/whats-new-with-azure-pipelines/>
- Netlify. (2025a). *Create Deploys with Continuous Deployment Using Netlify* [Accessed January 2026]. <https://docs.netlify.com/site-deploys/create-deploys/>
- Netlify. (2025b). *Netlify Build: Continuous Integration and Deployment* [Accessed January 2026]. <https://www.netlify.com/platform/core/build/>
- OpenAI. (2022). *Introducing ChatGPT* [Accessed February 2026]. <https://openai.com/index/chatgpt/>
- OWASP Foundation. (2023). *OWASP Top 10 CI/CD Security Risks* [Accessed 2026]. <https://owasp.org/www-project-top-10-ci-cd-security-risks/>
- PCQuest. (2025). *Customizing Continuous Integration* [Accessed January 2026]. Retrieved January 1, 2026, from <https://www.pcquest.com/customizing-continuous-integration/>
- Postmake. (2025). *AppVeyor* [Accessed January 2026]. <https://postmake.io/tools/appveyor>
- Travis CI. (2021). *Travis CI Turns 10* [Accessed January 2026]. <https://www.travis-ci.com/blog/2021-01-20-travis-ci-turns-10>
- Travis CI. (2023a). *About Travis CI* [Accessed January 2026]. <https://www.travis-ci.com/about-travis-ci>
- Travis CI. (2023b). *Travis CI for Open Source* [Accessed January 2026]. <https://www.travis-ci.com/open-source>

Appendix A

Interview Questionnaire

The interview questions were structured in 6 categories. Some questions were conditional to the responses received on previous questions:

1. General questions about the respondent:
 - a) Please briefly introduce yourself.
 - b) Report on your past and current experience in collaborative software development, and on the kinds of projects you are or have been actively involved in, for which CI/CD tools have been used.
 - c) What is or was your involvement in those projects?
 - d) How many years of experience do you have with CI/CD?
2. General questions about CI/CD usage:
 - a) When did you *first start to use* a CI/CD tool in those projects and what was the reason at that time?
 - b) What are *currently* the main reasons for using CI/CD in those projects?
3. Questions about specific CI/CD tool usage:
 - a) Which different CI/CD tools have you used in the past, or are you currently using?
 - b) Why did you or the project maintainers decide to use that particular CI/CD tool?

- c) What are the resources (in terms of budget, hardware, personnel, etc.) and effort that are or were available and required for creating, hosting and maintaining the CI/CD infrastructure for your projects?
 - d) *[If one of the reported CI/CD tools was Travis:]*
 - Was Travis a kind of default choice, or was it a deliberate choice?
 - Are you aware of Travis ' changes in its free plan for public repositories? Has your project been affected by these changes?
 - e) *[If none of the reported CI/CD tools was Travis or GitHub Actions:]*
Why haven't you ever used Travis or GitHub Actions?
 - f) What were the main reasons for using these CI/CD tools, and what were/are the advantages and shortcomings of each of them according to your experience?
4. Questions about CI/CD migration: [These questions should be answered for every project that was reported by the respondent.]
- a) Did the project migrate from some CI/CD tool to another one during its lifetime?
 - b) *[In case of negative answer to 4.a:]*
Even if the project did not migrate its CI/CD tool, did you ever consider migrating to another CI/CD tool? If yes, why didn't you carry out the migration?
 - c) *[In case of positive answer to 4.a:]*
 - When did the project perform the migration?
 - From which CI/CD tool to which other CI/CD tool?
 - What drove the decision to migrate, and on which replacement CI/CD tool to adopt? (Was the migration because you disliked something in the existing CI/CD tool? Or because you liked something better in the replacement CI/CD tool?)
 - How much effort and time did it take to do the CI/CD migration and why?
 - What were the main difficulties (if any) in doing the migration?
 - How satisfied were you with the replacement CI/CD tool?
 - d) *[In case the respondent did not mention GitHub Actions as a CI/CD migration target:]*

- Are you aware of GitHub Actions and its increasing popularity? Why do you think this is the case?
 - Did you ever consider using GitHub Actions for doing CI/CD?
 - If not, why not? What is missing in GitHub Actions in order for the project to migrate to it?
 - e) To what extent has the acquisition of GitHub by Microsoft in June 2018 affected you? Did it trigger you to migrate from one platform to another, for example from Github to GitLab or vice versa?
5. Questions about CI/CD tool co-usage:
- a) Did or does the same project use multiple different CI/CD tools simultaneously? Which ones?
 - b) When and for how long have they been used together?
 - c) What is or was the reason for using multiple CI/CD tools within the same project? What is or was the purpose of each CI/CD tool?
6. Closing open-ended question:
- a) Do you have any other important remarks related to CI/CD tool usage that you would like to share with us?

Understanding the Evolution of GitHub Actions Automation Workflows

Social coding platforms such as GitHub have fundamentally transformed collaborative software development by lowering the barriers to participation and enabling large-scale, distributed collaboration. Features such as pull requests, code reviews, and integrated issue tracking allow developers from diverse backgrounds to contribute asynchronously, fostering vibrant open-source ecosystems. As repositories grow in size and activity, manual management becomes increasingly impractical, making automation an essential component of modern development workflows. To cope with these challenges, developers rely heavily on CI/CD automation tools to orchestrate build, test, and deployment processes.

This dissertation investigates the role of GitHub Actions through a combination of large-scale empirical analyses and complementary qualitative insights. It examines how workflows are adopted, how they are used in practice, and how they evolve over time across a diverse set of real-world repositories. The analysis captures both macro-level trends, such as adoption patterns and ecosystem-wide behaviors, and micro-level changes at the level of workflow entities. The findings indicate that while automation through GitHub Actions improves development efficiency, scalability, and consistency, it also introduces new layers of complexity. Workflow configurations become increasingly sophisticated, often involving intricate dependencies, reusable components, and external integrations. As a result, maintaining these workflows requires continuous effort, particularly in response to dependency updates, platform changes, and evolving project requirements.

By systematically uncovering these trade-offs, this dissertation provides a nuanced understanding of the role of CI/CD automation in modern software engineering. It highlights not only the benefits of automation but also the challenges that emerge as workflows evolve and scale. These insights contribute to a deeper understanding of the sustainability and long-term maintainability of GitHub Actions workflows in large-scale open-source projects.