CIMMI: A Maturity Model for CI/CD Practices

Master's Thesis

Simcha Vos

CIMMI: A Maturity Model for CI/CD Practices

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Simcha Vos
born in Groningen, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

CIMMI: A Maturity Model for CI/CD Practices

Author:	Simcha Vos
Student id:	5082110

Abstract

Modern software development has to conform to high standards. Developers rely on Continuous Integration and Continuous Delivery (CI/CD) to automate the software lifecycle. However, many developers are falling short in reaping all its benefits. Developers are often misguided due to a lack of awareness of all automations, and over-engineering specific aspects does not yield the expected outcomes.

Developers need guidance on their automation focus. A deeper understanding of the automation landscape is necessary to fully leverage the full potential of CI/CD, along with a structured framework that guides developers through these possibilities. We accomplish this by applying the concept of maturity to CI/CD practices.

In this work, we analyzed Python and Java repositories on GitHub. Through open coding and card sorting, an automation taxonomy was created to chart the automation landscape. We explore an approach that helps developers identify the most valuable automations for the current state of the project, aiming to ensure a balanced approach to different automation aspects and maximize the value of automation effort. We utilize this taxonomy in a survey and series of interviews to create a maturity model, which we validated through developer feedback on our repository analysis.

It was found that maturity has a strong positive association with key repository metrics, such as commit frequency. The majority of developers perceive the maturity model as effective and would use it to guide their CI/CD efforts.

Preface

This thesis represents the end of my academic journey at the TU Delft. It marks the conclusion of my time as a student, which I reflect on as a period of enormous personal and academic growth. This journey would not have been possible without the support of many individuals, and I would like to express my gratitude to everyone who has supported me.

First and foremost, I would like to thank my supervisor, Sebastian Proksch, for his guidance throughout my thesis. Our productive collaboration already began long before this thesis, as he was my responsible professor during the Bachelor's Research Project. His creativity in taking a higher-level perspective and envisioning new approaches to problems is inspiring and incredibly influential.

I also wish to thank Shujun Huang. Many parts of this thesis required external collaboration, and she was always there to provide it. I am proud of the results that we achieved together, and thankful for her continuous feedback on the progress of my thesis.

During my years at university, I am thankful to have met many friends. I want to thank everyone I met at the D.S.T.V. Obvius tennis association for all the memorable matches, competitions, and unforgettable moments both on and off the court. I also want to thank my colleagues, with whom I have spent countless hours both at work and outside of work. Two friends from my hometown of Groningen, Arjan and Ruben, also deserve special mention for their support and the memorable times we've shared, even after I left for Delft.

Lastly, I want to thank my family for their endless support throughout this journey. Although Delft is a long way from Groningen, I never felt alone and always knew I would be welcome at any time. I look forward to many more moments with them on my side.

To all of you: this thesis would not have been possible without your support!

Simcha Vos
Delft, The Netherlands
May 2025

Contents

Chapter 1

Introduction

Software is ingrained into almost every aspect of modern life. From pouring a hot beverage at the coffee machine in the morning to managing complex supply chains or analyzing large datasets, software is constantly evolving. Society needs software to be quick, efficient, secure, and easy to release. Currently, software is held to very high standards, and developers must keep their products compliant with new industry regulations, security protocols, and user expectations. At the same time, the demand for software developers is expected to grow significantly in the coming years, imposing more strain on software development [5]. The need for a highly efficient development process quickly becomes apparent.

The implementation of DevOps principles and practices has led to significant improvements in software development efficiency [30]. The backbone of this framework is Continuous Integration and Continuous Delivery (CI/CD). In the last decade, it has revolutionized software development, making it efficient and trivial to automate key parts of the development process. The result of these automation efforts is a more robust development cycle that enables quick releases, reduces unexpected behavior, and facilitates adherence to technical and business requirements.

Despite the widespread adoption of DevOps and CI/CD, many projects have fallen short in reaping all their benefits. In practice, many repositories lack maturity in key automation areas, while crucially, filling these gaps can significantly improve the efficiency of software development [13]. Developers may struggle to determine which automations they should implement next. By overspending on automation efforts in a few domains, they can unknowingly introduce an imbalance in their CI/CD. Their hardship is only logical, as there is no clear way to navigate the endless possibilities of automation. Developers lack structured guidance to assess and prioritize automation efforts, as the theory of CI/CD does not provide a clear notion of progression or maturity. This leaves developers without a way to assess how well their practices align with effective automation. On a large scale, this results in significant inefficiencies in software development and shortcomings in its quality. This uncertainty among software developers undermines the potential of modern software practices.

Other fields have established maturity models as a way of guiding organizations towards process improvement. Tailoring a maturity model to CI/CD practices can support developers by providing an efficient approach to automation that empowers qualitative software development. Inspiration is drawn from Capability Maturity Model Integration (CMMI), which utilizes a proven set of best practices to help organizations assess their current level of capability and performance, guiding them toward improvement [15].

By analyzing numerous commonly used automations and classifying their level of maturity, it becomes possible to suggest to developers which automations they should focus on next. This will help developers maximize their efficiency while keeping software at the high standard to which it is held. In this thesis, empirically derived CI/CD practices are utilized to develop a maturity model, providing a structured path for enhancing automation maturity in projects.

Research questions To guide the approach that this study will take to uncover the automation landscape, the following research questions will be answered.

RQ1 What automations do developers use on GitHub?

First, the possibilities of automation in CI/CD will be explored. The objective is to create a mapping that can be used to discover which automations are used in repositories. Many repositories will have declared automation code and build configurations, but it will not be immediately apparent what these exactly entail and what automations they are implementing. Therefore, this work aims to translate a repository's code into a set of automations that the respective repository is implementing. A combination of open coding and card sorting enables sorting automations into a taxonomy.

RQ2 How can we model the maturity of automations?

Using the taxonomy obtained in the previous research question, it becomes possible to explore the maturity level to which these automation tasks correspond. Establishing an order in the taxonomy of automations will enable developers to identify which automation to focus on next. It will allow for gauging the state of maturity of the automations. To realize this, a maturity model for automations will be created.

Many automations are expected to have some natural ordering, where automation x usually precedes the addition of some automation y. In that case, if a developer does not have either automation x or y, they should first focus on adding automation x. In addition, if they are already at automation y but have not completed x, they should focus on that less advanced automation first. The focus of this research question is therefore to better understand these automations, their relations, and the maturity level to which they correspond.

RQ3 How are the model suggestions perceived in practice?

The model developed in the previous research question requires practical validation. The model's performance will be tested by submitting automation reports to GitHub repositories. These reports will consist of an analysis of the repository's automations, based on the methods of the first and second research questions. The report will conclude with recommendations that developers should focus on next. Reports will be posted on the issue boards of repositories, and after some time, replies and issue status changes will be analyzed. The analysis determines whether the model is perceived as effective and useful to developers.

Results By answering the first research question, a set of automation domains will be obtained. Each of these domains will have specific automations that a project may or may not have. Additionally, it can be observed how often a certain automation is featured in the scraped repositories. The final product of this research question is therefore a taxonomy of automations, consisting of automation domains and tasks.

In the second research question, the automation taxonomy is used to create a maturity model by identifying a partial order of the automations. This order is derived from a survey and a series of interviews.

The result of the final research question is an analysis of the feedback that developers give on the maturity model. This produces a conclusion of how the model is perceived by developers, indicating its effectiveness in practice.

Contributions This thesis presents the following main contributions:

- A taxonomy of automations that groups CI/CD practices in four subdomains;
- A maturity model that maps automation tasks to a maturity level;
- Evidence is provided that maturity positively associates with key repository metrics;
- Evidence is provided that practitioners would use a maturity model.

The source code and datasets used in this thesis are available on Zenodo.org [40].

Chapter 2

Overview

This thesis contains a complex methodology. To facilitate the reader's understanding of the thesis's structure, an overview of all its parts is provided in Figure 2.1. In this diagram, it is evident that the thesis's complete methodology can be divided into three distinct research questions, which will be elaborated upon below.

A dataset comprising GitHub repositories will be used to address the first research question. All source files of these repositories are scanned, and relevant files are downloaded. The automations are extracted from these files for use in the following steps. The extracted automations are then used for open coding, which is conducted with two experts in multiple iterative sessions. Subsequently, card sorting is undertaken together with two experts. Both steps are conducted in three sessions, and external feedback is obtained between sessions to refine the process and outcomes. After card sorting, a prevalence study is conducted to further refine the results. Finally, the automation taxonomy is obtained, which is the product of the first research question.

The second research question builds on the taxonomy developed in the first research question. A survey is created to facilitate the gathering of a large number of respondents.

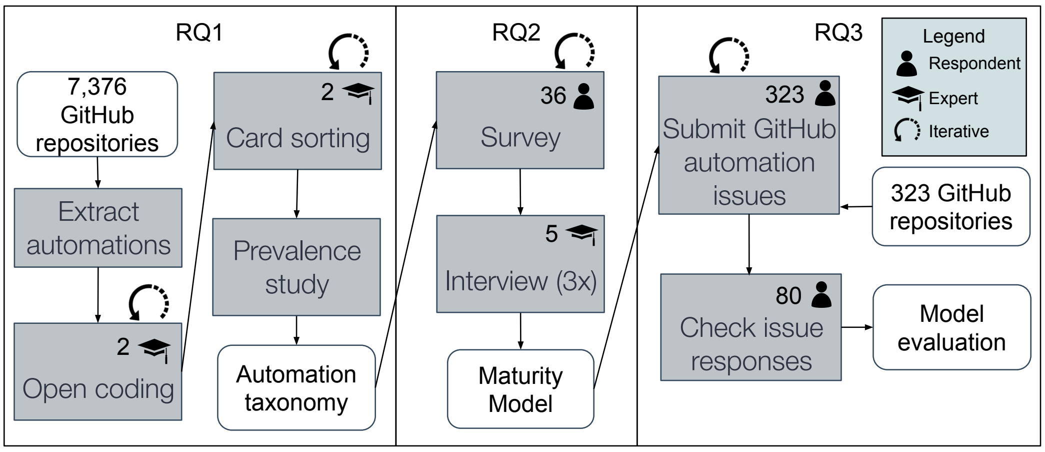


Figure 2.1: Overview of thesis structure

The survey is once again an iterative process; it is first piloted, and the feedback obtained is used to further improve the survey during its run. Once the survey is complete, a maturity model is created based on the results. This preliminary maturity model was proposed to three different domain experts during interviews. The interviews are conducted by one interviewer and one note-taker, who records comments and ideas. These comments are then processed and translated into actionable improvements, which are used to refine and finalize the maturity model.

The maturity model is used to submit GitHub issue automation reports for the third research question. A dataset containing 323 GitHub repositories is analyzed, and for each of these, an issue is submitted that contains an automation analysis. This process is again done in multiple rounds. This allows feedback to be incorporated between rounds and ensures that the quantity of discussions remains manageable. After submission of the issues, the issue responses are processed and replied to. All responses are categorized, and feedback points are gathered from the discussions on the issue. Through this, an evaluation of the model's effectiveness is obtained.

Chapter 3

Related work

This chapter discusses technologies related to modeling automation maturity in software development. In Section 3.1, works related to automation in software development will be discussed. Some papers have created a taxonomy based on these found automations, which will be further discussed in Section 3.2. In this thesis, a maturity model will be applied on top of the taxonomy. In Section 3.3, papers that have proposed maturity models on multiple different types of applications will therefore be discussed.

3.1 Automation in software development

To find related work, the search string ”software AND (”github actions” OR automation OR DevOps OR ”build tools” OR ”build configuration” OR ”continuous integration” OR ”continuous delivery”)” is used in Google Scholar.

Kinsman et al. analyze how developers use GitHub Actions after its release [17]. They found that GitHub Actions was met with an overall positive reception among software developers. They noticed a reduction in the number of commits of merged pull requests and an increase in monthly rejected pull requests. They speculate that this may indicate that project maintainers receive faster and more precise feedback on pull requests, helping them identify major issues in a vast number of contributions.

Wang et al. have focused on one specific automation domain: the relationship between test automation and software quality [41]. They found that projects that improve test automation maturity also obtain product quality improvements and acceleration of the release cycle. Although they discuss projects enhancing their level of test automation maturity, they do not assign specific levels to these projects.

Fitzgerald et al. define ”Continuous *” as an umbrella term encompassing continuous integration, deployment, delivery, testing, and many other related concepts [10]. They state that the integration between software development and its operational deployment needs to be continuous, and ”Continuous *” is required to realize software development principles such as DevOps.

Mohamed proposes a framework that combines agile development, continuous integration, continuous testing, and continuous delivery [23], utilizing automated tools and streamlined processes. He demonstrates that the framework enables releases to be delivered continuously without downtime, with higher quality, and with fewer unnecessary manual processes. As environments become fully automated, there is no need for staff to spend time on manual processes, resulting in increased efficiency.

While many researchers have analyzed GitHub Actions as a developer's automation of choice, there are more ways to implement CI/CD practices, such as through commands in the workflow files or through Maven plugins. At the same time, these works do not focus on furthering insights into the automations they found, such as constructing a taxonomy. This would enable developers to translate their findings into practical steps to enhance their automation efforts.

3.2 Taxonomy of automation

This section discusses papers that involve a taxonomy of automation tools. To find papers, the following search string was used in Google Scholar: ”taxonomy AND automation AND software”.

Sharma and Sodhi propose SEAT, which is a taxonomy to characterize automation in software engineering tools [31]. This aligns with the goals of Automated Software Engineering (ASE), which explores the automation of complex software engineering tasks and tools to support these processes [21]. The authors examine the techniques behind the ASE tools and the purposes for which they are used. They have realized the taxonomy, methods, purposes, properties, and relationships between tools as a graph database. This database can be extended and scaled as more data is gathered, allowing one to uncover hidden relationships and to query SEAT's content directly.

Melegati and Guerra have created DAnTE, which is a taxonomy for the degree of automation of software engineering automation tools [20]. It consists of six levels of degrees of automation. The first level indicates a total lack of automation or support tools. More advanced levels feature tools that use AI to generate solutions automatically. In their paper, they apply DAnTE to two software engineering activities: coding and testing. The authors remark that their taxonomy can help compare various tools and manage their expectations.

Faqih et al. have analyzed GitHub Actions that they encountered in workflow files [9]. They categorized the Actions from all programming languages into five different categories: Version Control Management, Static Code Analysis, Build Automation, Test Automation, and CI/CD Servers. They found that build automations are the most popular, while test automation has the least number of uses.

In these works, a taxonomy was created that can inform developers about the possibilities for automation. However, they do not yet offer direct actionable insights, as the taxonomies do not provide a clear roadmap to guide their automation efforts. Additionally, they do not consider all possibilities of implementing CI/CD practices. Because of this, their taxonomies are not as complete as they could be. In this thesis, more ways of automating will be considered. In addition, by applying the concept of maturity, a clear framework can be established that provides developers with specific directions for their automation efforts.

3.3 Maturity models

To find related works, the search string "(maturity OR mature OR improv* OR capability OR performance) AND (CMMI OR model OR taxonomy)" was used in Google Scholar.

O'Regan discusses the CMMI model, a framework that assists an organization in implementing best practices in software and systems engineering [24]. Implementing CMMI leads to benefits such as cost savings, higher quality, and more. The model consists of five maturity levels, with the higher maturity levels representing advanced software engineering capabilities:

- **Incomplete (0)**: The process does not implement all of the capability level one generic and specific practices. The process is either not performed or partially performed.
- **Initial (1)**: A process that performs all of the specific practices and satisfies its specific goals. Performance may not be stable.
- **Managed (2)**: A process at this level has the infrastructure to support the process. It is managed, i.e., planned and executed following policy; its users are trained; and it is monitored, controlled, and audited for adherence to its process description.
- **Defined (3)**: A process at this level has a defined process: i.e., a managed process that is tailored from the organization's set of standard processes. It contributes work products, measures, and other process improvement information to the organization's process assets.
- **Quantitatively managed (4)**: A process at this level is a quantitatively managed process: i.e., a defined process that is controlled by statistical techniques. Quantitative objectives for quality and process performance are established and used to control the process.
- **Optimizing (5)**: A process at this level is an optimizing process: i.e., a quantitatively managed process that is continually improved through incremental and innovative improvements.

Paulk discusses the evolution of the Capability Maturity Model (CMM) for Software [25]. He establishes the need for best-practice frameworks that can act as the basis for reliable and consistent appraisals. He warns that use of such frameworks can lead to dysfunctional behaviour for organizations that "simply want a certificate to hang on the wall"; however, models do engage the community in a meaningful discussion of best practices. Additionally, they note that frameworks should be abstract and flexible for use in a wide range of environments. While specialized variants, such as the Systems Security Engineering CMM, can be quite useful for niche domains, the source model should be broadly applicable.

Randeree et al. have created a maturity model for Business Continuity Management (BCM), which was tailored for the UAE banking sector [26]. The goal is to support the sustained performance of electronic systems on which their core activities are based. To develop their model, the authors analyzed five existing models and improved the developed model against their formulated objectives through the use of focus groups. They underscore the importance of providing a framework against which organizations can benchmark their current status. In addition, they show that software-based maturity models, such as CMM, rely on the fact that a target maturity level can be achieved only after going through several phased steps. They also demonstrate that the scope of their model can be expanded across various organizational sectors. They conclude their paper by emphasizing that the development of the maturity model is based on the underlying assumption that a more mature BCM process will result in better business continuity capability and results, leading to a higher maturity level.

Reis et al. have identified the state-of-the-art and the scientific gaps in maturity models [27]. They state that in immature organizations, processes are generally improvised, and managers, informally known as firefighters, are focused on solving problems, while deadlines and budgets are routinely exceeded. On the other hand, mature organizations possess a broad process control that enables development, management, and maintenance. Managers monitor the quality of products and processes, and, since deadlines and budgets are based on historical data, performance indicators are usually met. Creating a maturity model involves understanding the capacity of the process area [26]. The authors also discuss the purpose of a maturity matrix, which illustrates the maturity level for each process area and can be applied to evaluate strategic and operational capabilities across various domains. The authors conclude that they have identified numerous scientific gaps, encouraging further study and research on the subject.

Humble et al. define a maturity model for configuration and release management in organizations [14]. In their model, they address all roles involved in the delivery of software. Five levels of practice are defined: regressive, repeatable, consistent, quantitatively managed, and optimizing. They note that communication, collaboration, and automation are essential to obtain a high level, and bugs and defects should be rare.

While considerable research has been conducted on maturity models for various types of applications, CI/CD has not yet received significant attention. The works mentioned above do offer clear frameworks to create another maturity model. While some have focused on software development, they do not provide a complete and comprehensive model of all CI/CD practices. This thesis addresses this gap by applying the concept of maturity to the domain of CI/CD automations. By proposing a maturity model based on the automations encountered, a novel approach is used to enhance CI/CD practices.

Chapter 4

Exploring the automation landscape on GitHub

The first research question asks, ”What automations do developers use on GitHub?” This research question aims to explore the automation ecosystem to gain a better, quantified understanding of the importance and relevance of automation domains and their tasks. The goal is to support developers in their efforts to advance their state of automation. While many developers are motivated to automate their projects, they are not all aware of the automation tasks that can be completed. To provide developers with a comprehensive understanding of the automation landscape, it is necessary to gain a deeper understanding of the automations that developers are using in practice. This will be achieved by analyzing a large dataset of GitHub repositories.

4.1 Dataset for Mining Software Repositories studies

To analyze the automations that developers are using on GitHub, a suitable dataset has to be chosen and sampled. After the dataset has been obtained, it additionally needs to be processed, which will ensure that only data of interest is stored and extracted from the GitHub repositories.

Dataset selection Several key factors should be considered when selecting a dataset. The goal is to scrape a large, representative portion of GitHub repositories, most of which will implement some form of automation. Dabic et al. have recognized the frequent need of researchers to select subject repositories in Mining Software Repositories (MSR) studies and created a web application that allows researchers to obtain a large and representative dataset based on custom filters [8]. In this study, their tool is used to select the dataset based on some custom filters.

Several filters have been selected for this study. The aim is to identify projects that have had recent activity and are not merely toy projects, but rather repositories with ongoing software development. Therefore, a larger number of soft filters are applied to the repositories.

To qualify as a meaningful project for subsequent analysis, a repository is expected to meet each of these filter requirements.

It is important to note that filtered repositories are not required to implement any automations. Their lack of automation is also a relevant result for this thesis. The following filters were used to select repositories for the dataset:

- Number of commits ≥ 10: ensures projects with some development history that are not abandoned;
- Number of contributors ≥ 2: indicates collaborative development rather than personal or toy projects;
- Number of stars ≥ 5: shows some level of community interest and appropriateness of repository;
- Lines of code ≥ 10: filters out empty repositories;
- Languages: Python, Java: focuses on two popular programming languages;
- Forks: excluded: avoids analyzing duplicate projects or derivations of other projects;
- Created before 1/1/2024: ensures that the project was not very recently created;
- Commit after 1/1/2024: ensures that the repository has had recent activity.

The selection of eligible repositories for this study resulted in 17,882 Java repositories and 55,862 Python repositories as of November 2024. To make the dataset size manageable for querying with the GitHub API, random independent sampling was used to select 10 percent of the repositories for each language. This resulted in a final dataset consisting of 7,376 repositories.

Dataset processing All repositories selected in the sampling were scraped using the GitHub API. Only files that are of interest, such as `pom.xml` and `.github/workflows/*.yml-.yaml` files, are downloaded for further analysis. Using Python, the GitHub workflow files are inspected, and each `Run` and `Uses` step is processed. The `Run` element typically contains multiple lines of Unix commands. These commands are either compounded or separated into an array of multiple single Unix commands. The Maven POM files are analyzed using Java and an implementation of `ModelResolver`, which is part of the Maven API libraries. POM files are resolved to their effective POM, while fetching any necessary relative or parent POMs as well as POMs listed on Maven Central. The effective POM is a POM that embeds the project's POM file, any parent POM files, and the super POM file [33]. Some POMs refer to versions of artifacts that are not yet hosted on Maven Central; in such cases, the latest version of the artifact is scraped from the Maven Central repository. Finally, the effective POM is used to obtain the repository's plugins (designated as *groupId:artifactId*), which contain the automations that are of interest.

The result of the dataset processing is sorted based on the frequency of both the workflow RUN and USES elements, as well as the Maven plugins used. This can be used to analyze the most frequently used instances of automations quickly, and it is straightforward to note the fraction of the dataset that utilizes this automation instance.

4.2 Open coding and card sorting to classify automations

Three methods are used to uncover the automation landscape on GitHub. Open coding is used to identify categories that correspond to the various automation tasks present in the dataset. Then, card sorting is used to categorize each automation instance into a task. To finalize the results of this research question, a prevalence study was conducted. Combining these techniques to obtain an automation taxonomy has been effective in deriving meaningful categories in software engineering [37] [38]. The goal of this research question is to categorize the discovered automations within a taxonomy. Card sorting is considered an effective method for deriving taxonomies from data (e.g., Zimmermann [44]).

Open coding The ranking obtained during dataset processing is used to perform open coding. Open coding is the process of breaking down, examining, comparing, and categorizing data [35]. This part of the analysis focuses on naming and categorizing the data obtained from the GitHub repository analysis. The data is broken down into discrete parts. A search through documentation and search engines is followed by an expert discussion to identify categories that match the encountered automations.

During open coding, all plugins, actions, and commands that are used in more than 1% of all repositories were categorized. The found automations are categorized into automation tasks (or objectives). The open coding was performed in multiple sessions together with another expert with extensive knowledge of automations. The process is described in Algorithm 1. Several tasks are involved in open coding. Identifying the primary function requires the experts to recognize the function of the automation. If the automation is not familiar, documentation will be searched for online. Finally, if the function exists in *tasks*, it is considered whether the name or description of the function should be changed.

Upon expert disagreement, it is required to further investigate the functionality of the plugin or workflow job. In these cases, it proved useful to search for occurrences in the source files to determine which goal the developers were using these automations for.

Algorithm 1 Open coding of automation tasks

```
tasks ← []
for each automation do
    function ← identify_primary_function(automation)
    if function is unknown then
        function ← search_documentation(automation)
    end if
    if function not in tasks then
        add_new_task(tasks, function)
    else
        refine_existing_task(tasks, function)
    end if
end for
return tasks
```

Card sorting Card sorting can be used to discover the optimal organization of information [43]. While initially used by psychologists to study how people organize and categorize their knowledge, it is now frequently used in computer science as a quick, inexpensive, and reliable method to generate structure in data [34]. It can additionally be used to create taxonomies [44]. There are two primary methods for performing card sorts. In open card sorting, participants are given cards with no pre-established groupings. Alternatively, in closed card sorting, participants are given cards with an established initial set of primary groups.

Using the categories obtained in the previous step, open card sorting was performed to assign every automation instance to one of the found categories. After this, open card sorting was used once more to combine automation tasks into overarching automation domains and to identify subdomains within these domains. This results in a taxonomy of automations, comprising automation domains, subdomains, and tasks.

The entire process is done in multiple rounds. Between each round, feedback from another expert was obtained to get an external point of view. This feedback was incorporated before each round and discussed at the beginning of the new round with the fellow card sorter.

Prevalence study After the card sorting has finished, a prevalence study is conducted to obtain insights into the taxonomy. Using the taxonomy, all sampled repositories are analyzed again, and counts of each automation task are aggregated. The prevalence study serves several purposes. Found categories can be validated by confirming real-world usage of these automations in the sampled repositories. Additionally, the prevalence study is used to further refine the taxonomy. For example, less popular tasks that are functionally similar can be found and merged. Finally, the automation tasks were examined for their functionality, as some commonplace tasks were merely assistive and did not constitute tasks that should be implemented to achieve better automation maturity. An example of this was a task that involved data processing tools. This task includes tools that are assistive in nature and are not goal-oriented or tied to automating software development. The concept of maturity will eventually be applied to this taxonomy, and implementing such a task is not indicative of a project's automation maturity.

4.3 Extracting automations from GitHub workflows and Maven plugins

The architecture of the system is now described, along with all its intricacies. It consists of five distinct components, each fulfilling a different function. The majority of the system is implemented in Python, while just the PomAnalyzer is implemented as a Java project based on Maven. A diagram of the architecture is shown in Figure 4.1.

RepositoryDownloader The dataset, comprising 7,376 repositories, is provided to the RepositoryDownloader. The RepositoryDownloader interacts with the GitHub API to download all files. It specifically searches for all pom.xml files and all .YML and .YAML files in

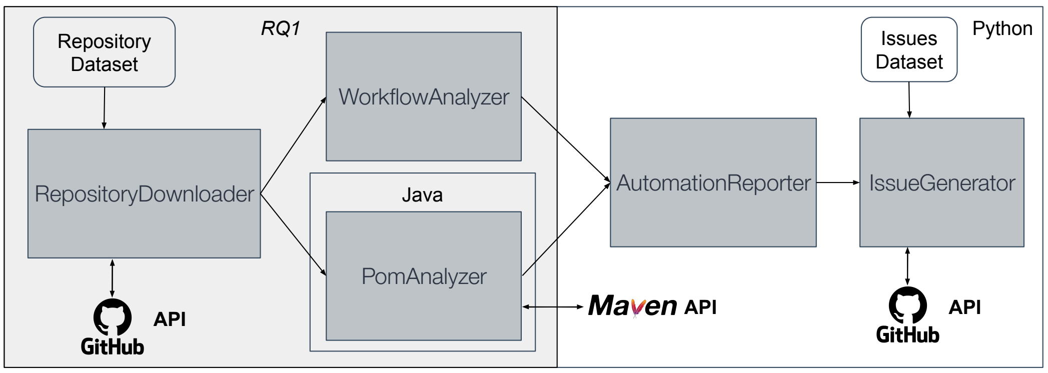


Figure 4.1: System architecture of entire thesis repository

the .GITHUB/WORKFLOWS directory. Due to GitHub rate limits, it is not feasible to download the entire dataset at once. To avoid rate limits, all requests are cached on the local machine. This way, it is not necessary to make more API requests for files that have already been downloaded.

WorkflowAnalyzer The WorkflowAnalyzer examines each workflow file and extracts its automations. Pseudocode for this process can be found in Algorithm 2.

For each encountered RUN element, WorkflowAnalyzer calls PROCESS_COMMANDS. The considerations for handling bash commands will now be further discussed. An example of a bash command used in such an element can be found in Listing 4.1. Commands are often spread across multiple lines, or multiple commands can be combined on the same line using a control operator, such as the pipe operator (|). Additionally, commands can feature flags, which are often not relevant for the automation task itself, but rather specify details about how the automation should be run. A Bash parser is used to combine multi-line commands and separate commands that operators join on the same line. After that, all command flags are stripped from each command. Then, prefixes like SUDO are removed

Algorithm 2 Workflow automation analysis

```
$automations \leftarrow []$
for each job do
    for each step do
        if step contains a Run element then
            append($automations$, process_commands(step[Run]))
        else if step contains a Uses element then
            append($automations$, step[Uses])
        end if
    end for
end for
return automations
```

```
run: |
  mvn clean \
    --quiet --batch-mode -DforceStdout=true \
    -DskipTests \
    package \
  && make deploy
```

Listing 4.1: Example of a workflow RUN: block

from the command. These prefixes were identified because they were consistently present at the beginning of commands with high prevalence and were always followed by a second command. Finally, there are a few exceptions that require custom handling. These exceptions were found by sorting based on the occurrence of the first one or two tokens of a command.

An example is the MVN command; an instance of this command is given in part of Listing 4.1. This example consists of MVN CLEAN and MVN PACKAGE. It can also be noted that the flag -DSKIPTESTS is not removed, which is, in fact, the only flag that is never removed during the stripping of command flags. This exception was found during the analysis of the MVN documentation. In this command, a set of tasks is executed based on the designated phase in the Maven life cycle. An example is that the MVN PACKAGE command triggers several Maven phases as part of the default build lifecycle: e.g., COMPILE, TEST, and PACKAGE. In this case, since the command flag is set to skip tests, the TEST phase will not be included. The resulting bash commands found by processing Listing 4.1 are: MVN CLEAN, MVN COMPILE, MVN PACKAGE, and MAKE DEPLOY.

PomAnalyzer PomAnalyzer is implemented as a Java project built with Maven. It implements a CUSTOMMODELRESOLVER to resolve the POM files, and depends on the Maven model-building API to construct the POMs. The main loop of the code recursively examines each repository and resolves the model in each POM file. The resolution of the model has one major challenge: the parent POM file. This parent POM file can be present in several different locations. Often, it can be found in the root directory, or in a different directory, in which case the RELATIVEPATH parameter is set [33]. In some cases, the parent POM is defined in a remote repository. It is not trivial and sometimes not possible to access these repositories, as only the build server might have access to them. Keshani et al. found that in their combined Maven and GitHub repositories dataset, 1.4% of projects had problems with model resolution due to parent POMs being hosted in repositories other than Maven Central [16].

These issues are resolved in the following way. If a parent POM cannot be found locally, CUSTOMMODELRESOLVER tries to find the parent POM on Maven Central. If the artifact can be found but the specified version is not yet present, it will resolve the parent POM by using the most recent version available on Maven Central. If the entire artifact cannot be found, it can likely only be found on an external remote repository. In these cases, a minimal POM is provided as the parent POM, allowing the model to still be resolved based on the found POM file.

The PLUGINS and PLUGINMANAGEMENT sections of the effective POMs are extracted. While the PLUGINMANAGEMENT section does not necessarily force Maven to use these plugins, it is presumed that if developers set this up, it is a strong indicator for the plugin also being used.

4.4 Results

The first research question asks, ”What automations do developers use on GitHub?” Over 7 thousand repositories were analyzed to answer this question. The results were obtained through open coding, card sorting, and a prevalence study. The product of this research question is a taxonomy of automation domains, each containing subdomains and tasks.

Figure 4.2 shows this final taxonomy. It consists of four main automation domains: Code Quality, Artifacts, Collaboration, and Development. Each domain is divided into one or more subdomains. These subdomains comprise several tasks, which were each found in the repositories analyzed.

Automation subdomains Figure 4.3 shows how often each automation subdomain is implemented. In this graph, a subdomain is considered implemented by a repository if at least one task from this subdomain is executed. The color of its bar can identify the automation domain to which the subdomain belongs. It is worth noting that the Development domain, which comprises the Pipeline subdomain, is the most popular among all. This is likely because it is often necessary to set up the runner; for example, files and environment need to be configured. A few other subdomains are also notably popular: Release management,

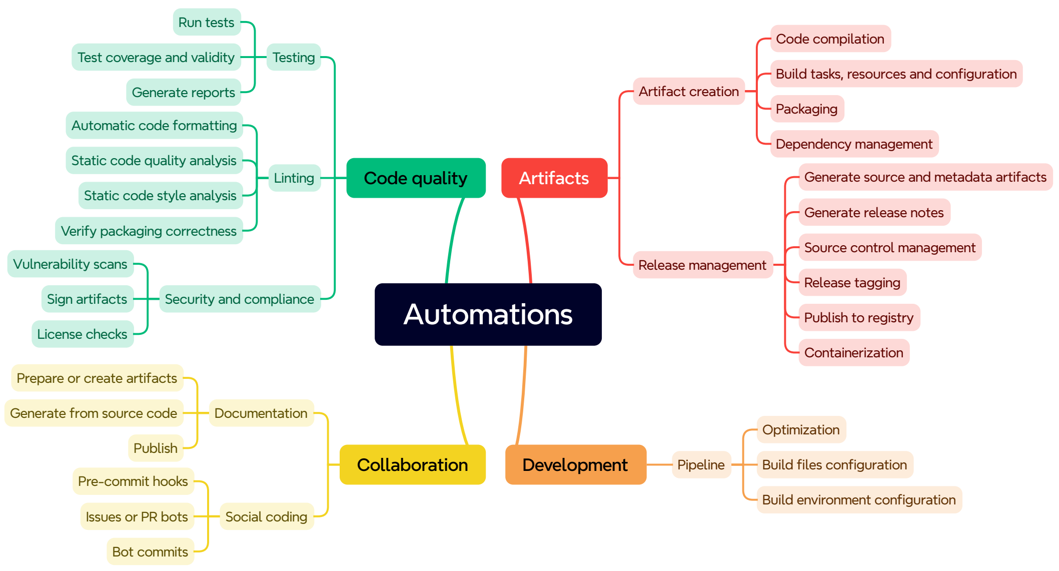


Figure 4.2: Automation taxonomy

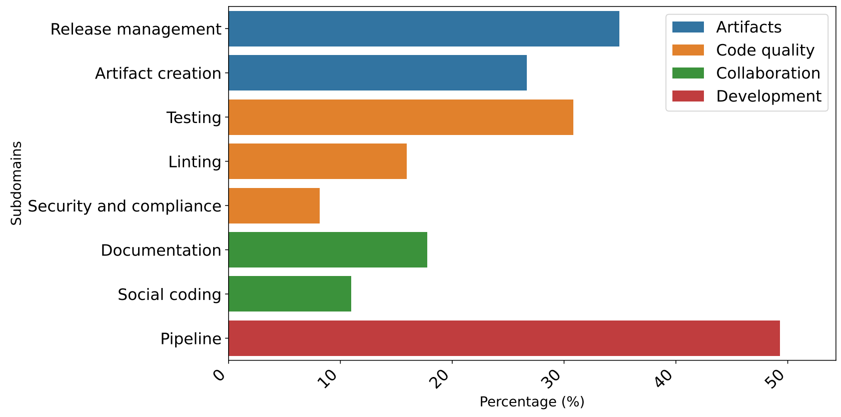


Figure 4.3: Percentage of repositories that perform tasks of the automation subdomains

Artifact creation, and Testing are all found in at least 25% of repositories. One could indeed argue that these correspond to the more fundamental automations. The Artifact creation and Release management subdomains seem strongly connected: to manage a release, one must first create an artifact that can be published. The graph also shows some less popular subdomains. Security and Compliance automations are not often employed, indicating that developers do not spend much time automating their security and compliance efforts, which is alarming.

Automation task frequency Figure 4.4 shows the distribution for all four automation domains.

The Artifacts domain consists of two subdomains: Artifact creation and Release management. The most popular tasks are Packaging and Publish to registry. These tasks also appear to be the most important for their respective subdomains. Some tasks are much less frequent; for example, Generate release notes. This is a task that seems more advanced and is not always needed for simpler projects. It might be argued that less frequent tasks are typically implemented only by the most mature projects.

For the Code Quality domain, the most popular task, which is Run tests, is the main task of the Testing subdomain. Most of the other tasks are present in more than 5% of repositories and resemble common tasks used for performing code quality checks in the pipeline. Some are much less popular: License checks and Sign artifacts are present in 1-2% of repositories. These tasks seem to be quite niche; perhaps they are seen as unnecessary for most repositories, or they could be too complicated. In this case, the effort required to implement these tasks should not be too large. However, the results indicate that most developers have not considered or deemed it necessary to implement these tasks.

The Collaboration domain features six tasks and is featured in a smaller number of repositories. The tasks are slightly harder to implement compared to tasks in other domains. At the same time, it remains essential to invest effort in this domain, as collaboration is fundamental to social coding platforms like GitHub [7]. Each repository that was analyzed was open-source; therefore, one might say that these public repositories would benefit most

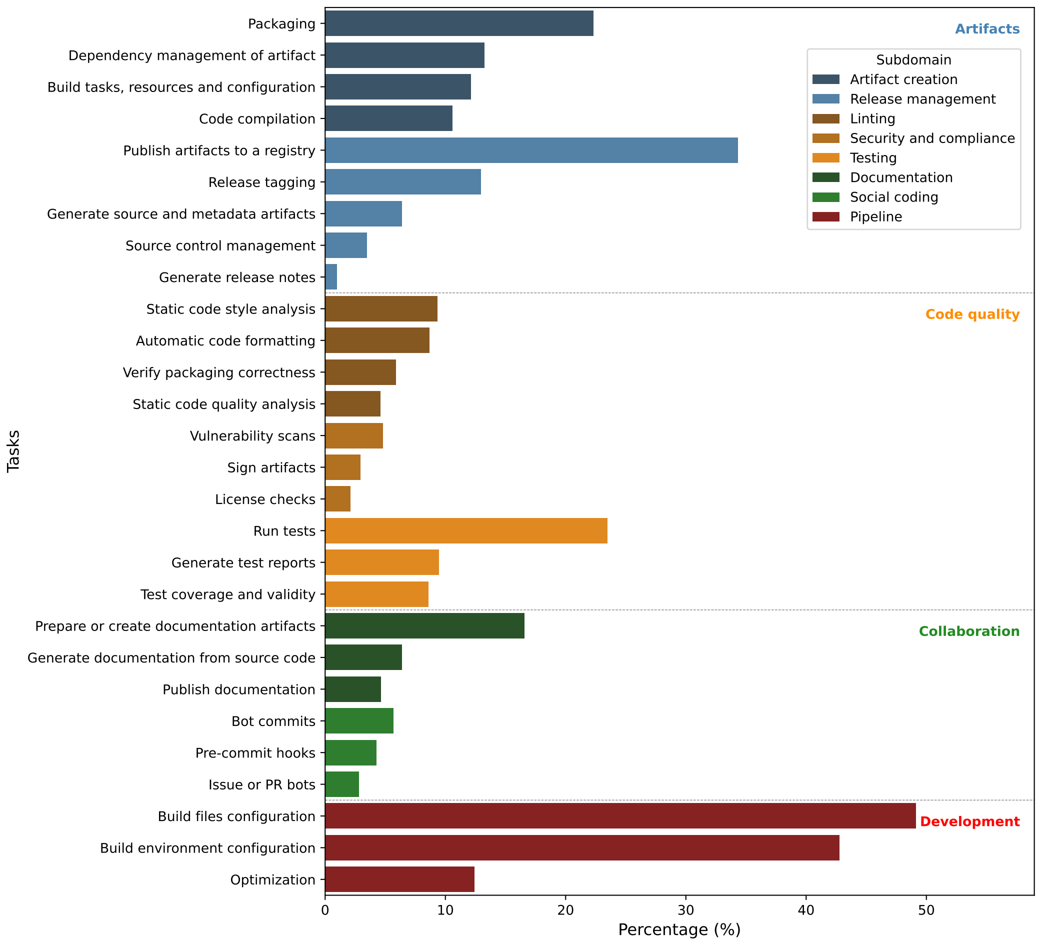


Figure 4.4: Percentage of repositories that perform automation tasks

from tasks completed in this domain, as they often rely on the contributions of others. It seems logical to accommodate developers who want to assist with coding efforts.

The final domain is Development, which is also the most popular domain. It can be seen that the tasks Build files configuration and Build environment configuration are the most prevalent in this domain. This does not come as a surprise, as for most pipelines, it is necessary to pull the required build files and set up the runner's environment.

Some tasks are only used by very few repositories and are therefore not found in the automation taxonomy. The use case of these tasks may be too niche, or there may not be enough knowledge about them. One example might be performance monitoring, such as load testing or identification of bottlenecks in code execution. Some of these examples are only expected in larger, perhaps enterprise-grade projects. Their absence in the taxonomy could be explained because only a fraction of repositories represent such large projects. Additionally, companies might more often depend on closed-source rather than open-source repositories. Within the scope of the thesis, it is not possible to analyze these seldom-used

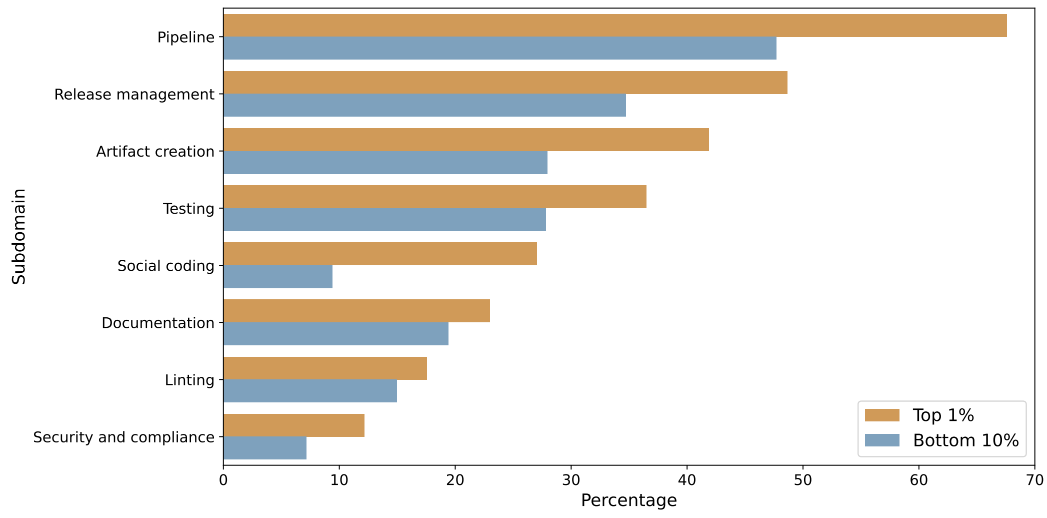


Figure 4.5: Automations in top 1% and bottom 10% most-starred repositories

automation tasks. However, the goal of the model is to provide developers with relevant and general automation suggestions. Since these tasks are rarely used or needed in practice, it is not productive to include them in the taxonomy.

Finally, Figure 4.4 also illustrates various well-known automation tasks, such as running tests. Inherently, some structure appears to be present among the tasks. If we take a look at the Testing subdomain, the following tasks can be identified: Run tests and Generate test reports. To obtain test reports, tests must be run first. One can observe that a relationship seems to exist between the frequency of tasks and the order of implementation. In this case, Generate test reports requires Run tests. In line with this requirement, one can observe that the former task is implemented less frequently.

Stars The number of stars can also be considered in the analysis of automations. One may hypothesize that repositories with a higher number of stars have spent more effort on automation. Additionally, the relation between the number of stars and popularity has often been emphasized [3]. In Figure 4.5, repositories in the dataset with the 1% highest number of stars can be compared to the repositories with the 10% lowest number of stars. It can be observed that repositories with a high number of stars tend to implement more automations than repositories with a low number of stars. This result aligns with the assumption that repositories with higher popularity have also focused more on their CI/CD practices.

Limitations A limitation of the approach is that it is not feasible to card sort every automation used by developers. In the scope of the dataset, every automation that is used in 1% or more repositories has been card sorted. However, the analysis does not cover the long tail that is present in the automation distribution, which is not possible to analyze due to time constraints. In total, 101 Maven plugins were analyzed, representing the top 99% most popular plugins, and 349 plugins were not analyzed. A much larger number of unique actions and bash commands were found in the GitHub workflows. A total of 1630 implementations were not analyzed, while 90 were analyzed, representing the top 99% most popular automations. The large number can be explained by the fact that many of the commands contain custom user input, such as file names. These custom commands are not useful for creating an automation taxonomy. Nevertheless, a large number of automation instances were not integrated into the taxonomy. However, it is worth noting that the primary objective of this research question is to explore CI/CD practices within an automation taxonomy. This means that the primary interest is to understand the automations that people are using in practice and could be recommended to developers. For a quantitative method like card sorting, it is not relevant to investigate automations that are rarely used, as their limited prevalence suggests they are not representative or broadly useful in practice.

To answer the research question, a taxonomy was created based on all the automations encountered in GitHub repositories. This taxonomy categorizes the automations into four distinct domains, each containing a set of automation tasks. Finally, to gain a better understanding of how often these automations are implemented in practice, each automation task has been plotted, allowing one to identify common practices in automation easily. It was found that repositories with more stars include a larger number of CI/CD practices.

Chapter 5

Modeling the maturity of automations

The second research question asks, ”How can we model the maturity of automations?” This research question aims to apply maturity models to the automation landscape uncovered in the previous research question, to understand the maturity level to which the found automation tasks belong. Previously, it has become possible to provide developers with a detailed overview of all possible automations, using the automation taxonomy. This thesis aims to guide developers in their automation efforts. To that end, it is necessary to establish order in the automation taxonomy, providing developers with a clear sense of which automations to focus on first. This order will be imposed on the taxonomy by creating a maturity model. This maturity model can then be used to gauge a project’s state of automation and recommend further automation efforts.

5.1 Obtaining maturity of automations through a survey

A survey is used to obtain an initial model of maturity and is deployed to a group of experienced software developers. According to Wendler, a survey is among the most used research methods for researching maturity models [42].

This study involved human subjects, so it is designed to follow the highest ethical standards. In research, these standards are established by both TU Delft and academic journals. To proceed with the user study, approval from the TU Delft Human Research Ethics Committee (HREC) has been requested. The application for approval contained a statement on informed consent, as well as an HREC checklist and Data Management Plan, and will now be discussed. Thereafter, the survey structure will be elaborated upon, along with the survey procedures for respondents.

Ethics requirements Before involving human respondents, it is essential to follow ethical guidelines and institutional requirements. TU Delft’s Qualtrics was chosen as the survey tool, as this is the university’s preferred tool for surveying. Compared to other tools, it is most trivial to adhere to ethical requirements using this tool.

Positive approval from the TU Delft HREC is required before distributing the survey. A statement of informed consent was formulated, which respondents must agree to to participate in the survey. Additionally, the HREC checklist was completed, which requires researchers to identify risks and provide methods to mitigate them. Finally, a Data Management Plan was constructed. This plan outlines the procedures for securely storing data gathered through the survey.

Question structure The survey first asks the respondent a series of preliminary questions. They must choose their highest degree in a field related to computer science, as well as any relevant work experience they have. Then, two questions are asked about the respondent's experience with CI/CD tools or build automations. They are asked which tools they have used, and then, they are asked how often they use these tools.

The central portion of the survey is presented after the demographics questions. Respondents are shown the automation tasks corresponding to each of the four domains. For each domain, they can select whether they think the task belongs to the 'Basic', 'Intermediate', or 'Advanced' maturity level. While these domains also have corresponding subdomains, it is a deliberate choice that these subdomains do not sort the automation tasks in the survey. Naturally, respondents are expected to try to assign all three maturity levels within these subdomains. However, this is not entirely realistic, as some subdomains may be associated with more advanced automations than others. After assigning the maturity levels, the respondent may optionally provide a remark on any of the questions or tasks.

Finally, the survey concludes with a series of reflective questions about the research approach. In this section, the respondent is asked whether they agree with the study's concept of maturity and whether they believe the survey results can provide meaningful suggestions to repository maintainers. Finally, they are asked what they think is the main reason that some large projects are not at a high level of maturity.

The survey concludes with the option for respondents to enter their email address if they are available for follow-up questions and wish to participate in a raffle for a gift card.

Process of creating the survey Creating a survey requires careful consideration of the questions that respondents will be answering. There is a delicate balance between the time it takes to complete the survey and the quality or depth of the responses. Ethics should also be considered, and the approval of the HREC confirms the adequacy of these considerations.

The survey was created in multiple iterations. The contents of the survey were discussed in multiple meetings with the thesis supervisors and iteratively changed and improved. External feedback was also received during survey preparation. When the survey was finished, it was piloted with a few experts to test its feasibility, determine how long respondents would take, and assess whether the questions were adequately explained to them.

Recruiting respondents Respondents were recruited during two months. Finding respondents was tougher than usual, as the survey's eligibility criteria were quite strict, as it is required for respondents to have first-hand experience with automations during their studies or work.

Respondents were recruited through various channels. Direct and indirect personal contacts were invited to complete the survey. Consequently, some contacts also shared the survey in their networks. The survey was also shared on LinkedIn and on more anonymous social media platforms, such as Reddit, as well as multiple other online computer science and DevOps forums.

Since the survey was generally on the longer side in terms of time, and the eligibility criteria required specialized knowledge, a reward was added for completing the survey. The goal of this reward was to motivate participants to finish the survey. Each user had a chance to win a 20-euro gift card of their choice. At the same time, adding a reward such as a gift card also has drawbacks. There is a risk that the quality of responses is lower and that bots are used to get a large number of responses on the survey. However, according to Singer and Ye, incentives increase response rates, and studies evaluating the effect of incentives on the quality of responses have found no impact on the quality [32].

Quality of respondents Maintaining a high standard of respondent quality is a pressing issue in surveys. The interests of respondents are not always equal to the interests of the survey creators; examples are that respondents wish to finish the survey quickly. This behavior is sometimes referred to as satisficing, which refers to the "expenditure of minimum effort to generate a satisfactory response, compared with expending a great deal of effort to create a *maximally valid response*." [28] Barge et al. recommend researchers identify respondents who are satisficing and then make a judgment about their inclusion in the results [2].

In this study, multiple measures were taken to uphold the quality of responses:

1. On anonymous social media, a source tag was appended to the URL of the survey as a way to quickly track the origin of a respondent. This was not always conclusive, as it was possible to remove the source tag to create anonymity. In the future, it would be better to generate separate links for each platform that cannot be altered, but in Qualtrics, it was not trivial to do so.
2. Each open form response field was manually analyzed for patterns and use of AI.
3. Any responses that were filled in under four minutes were also removed.
4. The eligibility criterion for the survey was also checked; respondents were expected to have at least obtained a degree or have working experience in the field of automations. Additionally, respondents were required to have experience with automations.
5. Finally, one of the survey questions was regarded as a screening question. The automation task ”Run tests” was used for this, as within the scope of the taxonomy, this task corresponds to the Basic maturity level. While respondents who considered the task as Intermediate maturity were not affected by this, anyone who assigned the Advanced maturity level was seen as not having understood the assignment correctly. This filtering step was also discussed with the experts in the expert interview.

These strict measures will substantially reduce the number of responses that can be considered for the results. However, the increase in quality shall allow better and more reliable conclusions.

5.2 Refining the maturity model through expert interviews

A maturity model was derived from the survey's output. However, a survey has several shortcomings that require further investigation. For one, the short timespan of the survey means that some of the automation tasks were not well understood. To eliminate these inconsistencies, a series of expert interviews was conducted to evaluate the survey results and refine the maturity model further.

Besides surveys, an interview is also one of the most used methods for researching maturity models [42]. Another paper in a different field has previously also combined a survey with a series of interviews to create a maturity model [12].

Semi-structured interview The interview was conducted with one interviewer, one note-taker, and one expert. It was conducted as a semi-structured interview, which means that it included pre-planned open-ended questions. The interviewer had the flexibility to ask follow-up questions based on the expert's answers. Semi-structured interviews are particularly advantageous in qualitative research, as they enable the gathering of in-depth insights [1]. While data can be more complex to analyze, the presence of two interviewers makes it possible to provide a more objective assessment of the expert's answers.
The main questions for the interview were:

Question 1 What do you think about the division of tasks in each maturity level?

Question 2 What tasks do you think are missing?

The interview was structured as follows. First, background information was explained using a slideshow. Then, the approach of the thesis was explained, followed by an explanation of which part of the thesis the experts were involved in. For additional insight, it was also explained that a simple automation does not necessarily imply that this task is part of a less mature automation. To mitigate the shortcomings of the survey, particular emphasis was placed on making the concept of automation maturity as clear and comprehensible as possible.

The central part of the interview was a discussion of each automation domain. The expert was shown the maturity model for that domain, which consisted of a list of tasks that belonged to each of the maturity levels. Next to that, a frequency graph was shown, comparable to Figure 4.4. This graph was intended to help the expert understand how frequently these tasks were implemented. For each domain, the expert was also asked to consider any tasks that they would identify as missing.

A total of three expert interviews were conducted, each lasting approximately 45 minutes. The three experts were recruited through personal connections. Each expert interviewed had a different background. The first expert came from an academic background, the second expert had worked in a large company, while the third expert had an open-source development background. The diversity enables the creation of a maturity model that can be applied to various types of projects.

5.3 Creating a maturity model

We will now integrate the survey outcomes and expert interviews to create the maturity model. Several key aspects have been considered for this process and will be discussed in detail.

Partial order To order a set of objects, there are two main choices for ordering: a total order or a partial order. For the maturity model, it has been chosen to order the automation tasks using a partial order. This means that the maturity model categorizes automation tasks into three bins, corresponding to three maturity levels, and that there are four partial orders, each for one of the four automation domains. More concretely, each task is sorted on one of the three maturity levels, but within these levels, the tasks are not implicitly ordered. A total order, however, has one more property that a partial order does not have. In a total order, for any two objects, one of the two is bigger than the other [22]. In the case of automation tasks, it is assumed that the comparison between two tasks does not always yield a clear result indicating which is larger (or, in this case, more mature) than the other. While it may be statistically possible to derive a difference in maturity between any two tasks, it is not expected that a total order will yield benefits for this study. The goal is to provide developers with a simple way to assess their state of automation. This is the benefit of having three levels that the developer can easily understand. Additionally, any slight difference may be meaningless in comparison to other factors that can also indicate whether an automation should be implemented earlier or later. By grouping automation tasks into the same level in the ordering, it is possible to suggest automation tasks to developers based on other criteria. An example of this is the percentage of projects that feature this automation. Such a secondary criterion is more meaningful than a slight difference in maturity between any two tasks.

K-means clustering Using the results of the survey and expert interviews, the maturity model can now be created. The initial maturity model was created based on the survey results. For each automation task, the average maturity level, as reported by respondents, is calculated as a decimal value between 1.0 and 3.0. Each task within a domain shall now be mapped to a maturity level, ensuring that all levels are represented. This division is calculated using the K-means clustering algorithm. K-means is an iterative clustering method that minimizes distances within clusters, making it suitable for grouping tasks into discrete maturity levels. In other studies, K-means clustering has been used to categorize survey responses into multiple different groups [18].

To illustrate the K-means clustering, Figure 5.1 shows the clustering of the Code Quality automation domain. In this figure, eight automation tasks are observed, which are distributed along the x-axis according to their average maturity level, ranging from 1 (basic) to 3 (advanced). Given this axis, if a task has a score of 1.0, every respondent has indicated that this task belongs to the basic maturity level. While a task with a score between 1.0 and 2.0 may have any combination of votes for basic, along with votes for either intermediate, advanced, or both. Using K-means clustering, the algorithm iteratively clusters the closest distances until it reaches a total of three clusters. The average maturity score of these

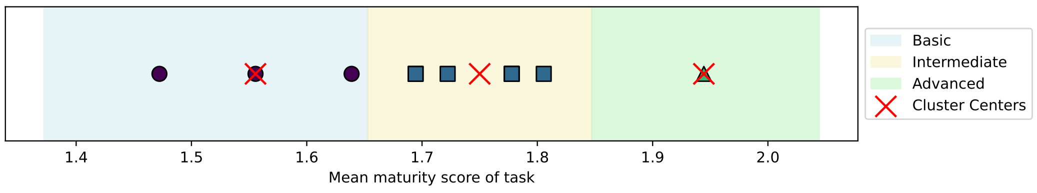


Figure 5.1: K-means clustering of Artifacts automation domain

clusters is indicated with a cross mark in Figure 5.1. The shapes of the dots indicate the maturity level to which they belong. In this case, tasks with a circle were mapped to the basic maturity level, tasks with a square to the intermediate level, and tasks with a triangle to the advanced level.

After the maturity model was presented in the interview, feedback was obtained through discussions. Together with one expert, the feedback points were discussed and were implemented to obtain the final maturity model for this research question.

5.4 Automation maturity scoring methodology

In this study, the maturity of repositories was scored on four axes, corresponding to each automation domain. As a result, it is not straightforward to provide quantitative insights into the repositories analyzed in this study. There are multiple options to combine the four scored maturity levels into one metric that indicates the maturity score of the entire repository across all automation domains.

An intuitive way to capture the overall maturity is to select the average maturity across all automation domains. This metric will be referred to as the *average score*. This approach captures progress and credits developers for every level that they have achieved. There is, however, a downside to this: the concept of maturity does not lend itself well to such a scoring metric. That is because basic levels are more important than more advanced levels. An example is that someone who has achieved the basic maturity level in all automation domains is doing a better job at balancing their automations than someone who has reached the advanced stage in one domain but has not done anything in the other domains.

In the scoring metric, the goal is to reward developers who effectively balance their automations. To this end, the scoring metric of CMMI might be considered [36]. The appraisal requirements for CMMI state the following:

> “A maturity level for a staged representation is achieved if all process areas within the level and within each lower level are either *satisfied* or *not applicable*.“

This means that in CMMI, the maturity level is chosen as the lowest maturity level across all areas. In this case, it would mean that developers would be at the lowest level for any of their automation domains. This score will be referred to as the *CMMI score* hereafter.

Finally, it is possible to calculate a score that is more lenient to developers who have missed just one level but have completed other automation levels. This scoring metric will be called the *joker score*, reflecting the fact that a repository may use a joker if it has missed a maturity level. Additionally, this metric exhibits *jumping behavior*, in the sense that it grants extra levels if subsequent levels have been completed. For example, a repository might have completed all maturity levels, except the intermediate level for the Collaboration domain. That means that, according to the CMMI score, they would be at the basic level. In this case, the joker will be assigned to the intermediate level. Since the joker is used for the intermediate level, the repository will now also be credited for the completed automations in the advanced level (because of the jumping behavior). This leads to an overall advanced maturity score for the entire repository according to the joker score.

5.5 Results

The second research question asked, ”How can we model the maturity of automations?” Two approaches have been used to create the maturity model that helps developers navigate the taxonomy established in the previous research question.

Survey Approximately half of the respondents had obtained a Bachelor’s degree, the other half had received a Master’s degree. A few respondents had either a PhD, were still students, or had completed another form of education. The median number of years of experience was five years. All respondents had used automation tools before, around 22% had used GitHub workflows, while around 14% had experience with Maven plugins.

A total of 197 responses were collected. Half of these responses (102) were unfinished. These respondents often only opened the informed consent page or stopped the survey on the first page containing questions. On two occasions, a large number of bot replies were found. These often contained AI-generated text on open questions, all contained bogus emails, used VPNs, completed the survey in an extremely short time (under four minutes), and triggered fraud scores on Qualtrics. Given that all of these responses included an email address, the gift card raffle may have attracted unwanted attention from malicious respondents. A total of 53 such responses were identified. In addition, as mentioned, the automation task ”Run tests” was used as a screening question, and a response of either ”Basic” or ”Intermediate” was required. This step filtered out five participants. Finally, the eligibility criterion for this survey was having experience with automation. To this end, everyone who has not graduated in computer science or another related field, or has not had any work experience, was disqualified from the survey due to a lack of knowledge or experience. This affected just one respondent. The total number of respondents after filtering was 36.

The first maturity model was created by applying the K-means algorithm to the survey results and is presented in Table 5.1. This maturity model was then further developed in the expert interviews, resulting in the final maturity model that is the output of this research question.

Although the maturity model assembled from the survey responses was not entirely satisfactory, it provided a suitable foundation for the expert interviews. This was because the survey respondents were able to appropriately assign many of the tasks that were presented to them. However, a few tasks did not end up at the expected maturity level. Additionally,

Table 5.1: Maturity model as established from the results of the survey

Subdomain	Basic	Intermediate	Advanced
Code quality	Run tests	Test coverage and validity; Generate test reports; Automatic code formatting; Static code quality analysis; Static code style analysis	Verify packaging correctness; Vulnerability scans; Sign artifacts; License checks
Collaboration	Generate documentation from source code; Prepare or create documentation artifacts	Commit validation; Bot commits	Publish documentation; Issues or PRs management
Artifacts	Code compilation; Dependency management; Source code version control	Build tasks, resources, configuration; Packaging; Generate source and metadata artifacts; Release tagging; Publish to registry	Generate release notes; Containerization; Push container to remote
Development	Build environment configuration	Build files configuration	Optimization

K-means clustering did not always yield an even distribution among the three maturity levels, which is not ideal for addressing the following research question. An example might be that the Basic maturity level contains a large number of tasks, which could result in a significant chance for repositories to be demoted to the lowest level of maturity if they fail to complete just one of the tasks.

Agreement on approach The survey also asked respondents for their opinions on the approach of the study. They were asked if they agreed with the notion that each automation task corresponds to a certain level of maturity. They were also asked if they thought it would be possible to provide meaningful automation recommendations to developers based on their state of maturity. Respondents were asked to share their responses using a 5-point Likert scale, ranging from ”strongly disagree” to ”strongly agree.” To confirm whether respondents agree—that is, statistically provide a higher agreement than neutral—a Wilcoxon signed-rank test was performed [29]. For the first statement, the test yielded $W = 89$, $p < 0.001$, indicating that respondents generally agree with the statement, using an alpha level of 0.05. The second statement yielded $W = 156$, $p < 0.05$, which remains statistically significant given an alpha level of 0.05; however, it shows that respondents expressed slightly lower confidence in its feasibility. Both statements will be further verified in practice in the later chapters.

Expert feedback The points raised by the experts will now be discussed. For code quality, one expert recommended changing Automatic code formatting to the basic level. Based on the fact that Static code style analysis inherently precedes Automatic code formatting, it was decided not to implement this suggestion. For collaboration, it was noted that tasks may vary from company to company. Additionally, it was suggested to rename Generate documentation from source code and Prepare or create documentation artifacts. One expert recommended categorizing Generate documentation from source code as intermediate, as far fewer repositories had implemented this. For the artifacts category, it was recommended to rename and move Commit validation to the advanced category, together with Generate source and metadata artifacts. Additionally, Source code version control should be renamed to more accurately represent the implementations. To make the model more straightforward, an expert suggested merging Push container to remote into Publish artifacts to registry. It was finally noted that containerization is not always applicable to every project. To maintain the model's generality, the implementations of the containerization task were moved to the packaging task, as containerization is a specialized form of packaging. For the final category, Development, it was recommended to exclude Git's checkout actions (which is responsible for inserting a repository's files) and move Build environment configuration to the Basic category. However, in another expert interview, it was eventually discussed that it is better to keep checkout actions, as well as to keep Build files configuration in the Basic category. Subsequently, Build environment configuration can remain in the intermediate category. This choice was made because, without the checkout action, there is not always a need to configure build files during the build process.

The final maturity model is based on the survey and features improvements based on expert feedback and is presented in Table 5.2. It can be observed that for many tasks, their level of maturity is comparable to the frequency of these tasks in the GitHub repository analysis in the first research question.

Table 5.2: Maturity model as established from the results of survey and expert interviews

Subdomain	**Basic**	**Intermediate**	**Advanced**
Code quality	Run tests; Static code style analysis	Test coverage and validity; Generate test reports; Automatic code formatting; Static code quality analysis	Verify packaging correctness; Vulnerability scans; Sign artifacts; License checks
Collaboration	Prepare or create documentation artifacts	Generate documentation from source code; Pre-commit hooks; Bot commits	Publish documentation; Issues or PRs management
Artifacts	Code compilation; Dependency management	Build tasks, resources, configuration; Packaging; Release tagging; Publish to registry	Generate release notes; Generate source and metadata artifacts; Source control management
Development	Build files configuration	Build environment configuration	Optimization

Repository metrics Finally, to gain insight into the relation between maturity level and repository metrics, several features were plotted against the maturity level of each repository in the first dataset. Many of these features have a strong correlation to properties such as the popularity and whether the repository is maintained. A trend line was added to more easily reflect how the median maturity score changes in the different maturity levels. Joker score, which was discussed in Section 5.4, was used as the scoring method. It is also worth noting that the advanced category is not represented in the graphs. Since there are no repositories that satisfy the advanced maturity level, this level will not be considered here.

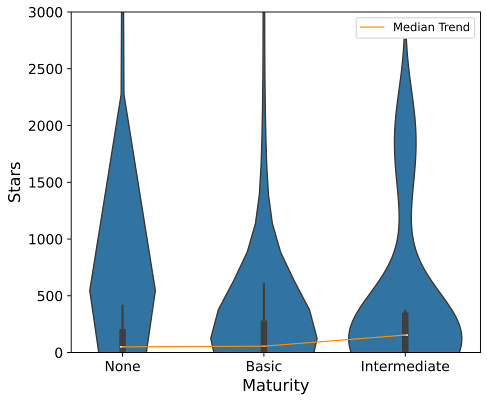
The plots are all visualized as violin plots for each maturity level. The thickness of the violin indicates the density of data, more specifically, the number of data points present. A thin part of the violin shows that there are almost no repositories in that range, while a thick part indicates many repositories. The trend line passes through the median value of each maturity level and can be used to determine whether the values on the two axes are correlated. Within each violin, the quartile and whisker values are shown as a vertical line.

The number of stars is plotted against the maturity score in Figure 5.2a, which is one of the most well-known features of a repository. There are several outliers spanning hundreds of thousands of stars, which are not visualized. It can be noted that the median trend line of maturity has a slight increase, indicating that the central tendency of star counts (as measured by the median) increases with the maturity level. At the same time, there is no apparent difference in the distribution of the different maturity levels; however, a significant number of repositories without any achieved maturity level exist, which have more than 1000 stars. In Figure 5.2b, the number of forks is plotted against the maturity level. The number of forks follows a similar distribution to that of the number of stars. Borges et al. found a strong correlation between stars and forks, which can account for this similarity that was found [3]. Another work states that the number of stars of a repository is a direct measure of its popularity [4].

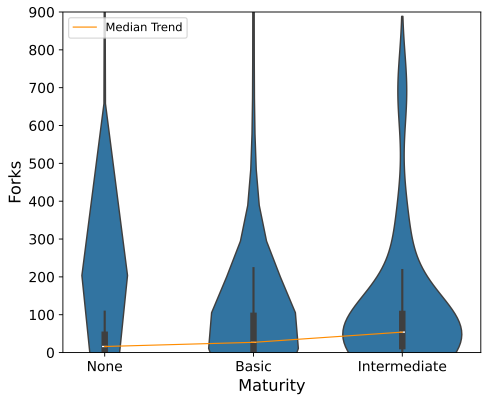
According to Han et al., the number of open issues is one of the most important factors of repository popularity. A clear increasing median trend is visible in Figure 5.2c. The repositories without a maturity level show a dense base with few very high values, suggesting most low-maturity repositories have few issues. The basic and intermediate repositories show a more spread-out distribution of open issues. A larger number of open issues can correlate with larger, more active projects.

The correlation between commit frequency and maturity level is evident in Figure 5.2d. Notably, projects of intermediate maturity have a higher number of commits per month. Coelho et al. have proposed an approach to measure the level of maintenance activity of GitHub projects [6]. They have found that the number of commits within a period is the most relevant feature for determining whether a project is actively maintained. In the graph, it can be observed that projects with a higher number of commits also exhibit a higher level of automation maturity.

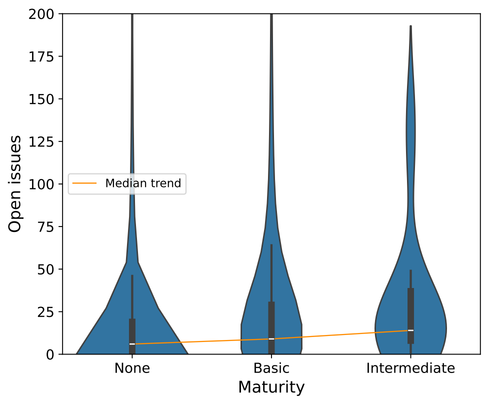
Finally, an essential attribute of a repository is age. The correlation between automation maturity and age is not readily apparent. Nevertheless, if one constricts the scope to younger projects that are not older than five years, a significant trend can be noticed in Figure 5.2e. Projects that are less than two years old are rarely at a more advanced stage of automation maturity. On the other hand, from the distribution, it follows that projects that are over



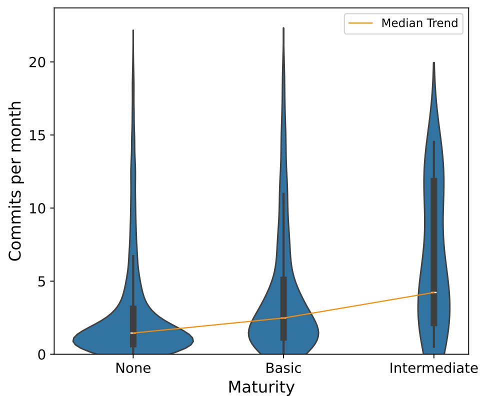
(a) Stars and maturity level



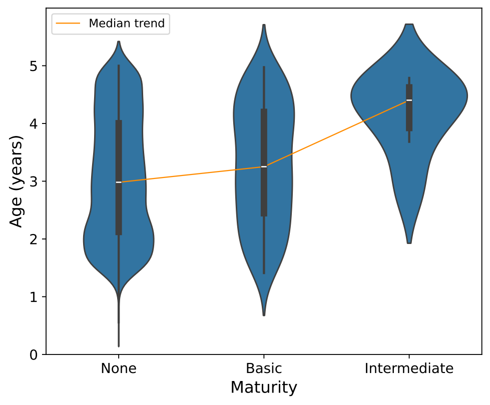
(b) Forks and maturity level



(c) Open issues and maturity level



(d) Commit frequency and maturity level



(e) Age and maturity level in young projects

Figure 5.2: Associations between repository metrics and automation maturity levels

Table 5.3: Kruskal-Wallis test for distribution of repository metrics in different maturity levels. Degrees of freedom = 2.

Metric	**Maturity level**			**Statistic**	**p-value**
	None	Basic	Intermediate		
Stars	50	56	154	6.031	0.049
Open issues	6	9	14	39.627	< 0.001
Forks	16	27	54	79.742	< 0.001
Commit frequency	1.4	2.5	4.2	94.928	< 0.001
Age (< 5 years)	3.0	3.3	4.4	151.808	< 0.001

four years old are more often at intermediate maturity than those without any completed automation maturity level.

Distribution of repository metrics Table 5.3 is based on the same data as Figure 5.2. One can observe that the data reveals a consistent upward trend for all repository metrics as the maturity level increases. To test whether these observed differences were statistically significant, the Kruskal-Wallis test was employed. The test is a non-parametric alternative to ANOVA, and can be used to assess the differences among three independently sampled groups on a single, non-normally distributed continuous variable [19]. This study uses it to evaluate whether the three maturity levels exhibit different distributions for five repository metrics. The null hypothesis assumes the distributions of the groups are the same.

Statistical analysis revealed significant differences across the examined repository metrics (Table 5.3). The strongest associations were observed for commit frequency and age in young projects. The association between stars and maturity was the weakest of the five, but the relationship remains statistically significant, as the p-value is less than 0.05.

The research question was answered by developing a maturity model. It was found that a survey alone was inadequate for creating a satisfactory model. Therefore, expert feedback was gathered and integrated into the maturity model. Maturity was modeled using a partial order, and k-means clustering was employed to assign maturity levels to each task. By interviewing experts from different domains in computer science, the maturity model is effective for various types of projects. The Kruskal-Wallis test suggests a positive association between maturity and several important repository metrics.

Chapter 6

Perception of model in practice

The third research question asks, ”How are the model suggestions perceived in practice?” In the previous two research questions, an automation taxonomy and maturity model were obtained. The primary objective of the thesis is to develop a maturity model that assists developers in guiding their automation efforts. By providing feedback to developers on their automations, it becomes possible to verify the model's effectiveness in practice. This allows the study to gain a better understanding of how developers perceive the model and encourages researchers to produce future work that utilizes the automation taxonomy and maturity model.

6.1 Showing automation analysis to developers on GitHub

The model's performance will be evaluated in practice by submitting automation reports on the Issue boards of GitHub repositories. The structure of the report will be discussed, along with the dataset selection. Finally, the method of categorizing open responses by developers on the issues is elaborated upon.

Ethics committee Permission for analyzing developers' feedback on the automation reports has to be requested from the TU Delft HREC. During the submission of the initial HREC application for the previous survey, this permission was requested. The HREC checklist that was filed also considered the implications of processing developer feedback on GitHub, and mitigation plans were established. Finally, the Data Management Plan submitted with the application is also applicable to this part of the thesis.

Report structure The report contains a short introduction explaining the report. Next to that, a radar chart is shown, indicating to developers which maturity level they have achieved for each domain. An example is shown in Figure 6.1. This chart, for example, shows that while the project is at advanced maturity in the Development domain, it lacks maturity in the Code quality domain.

The automation analysis is displayed in a table. This table can be found in Figure 6.2. In the table, tasks are categorized by the domain to which they belong. For each domain and

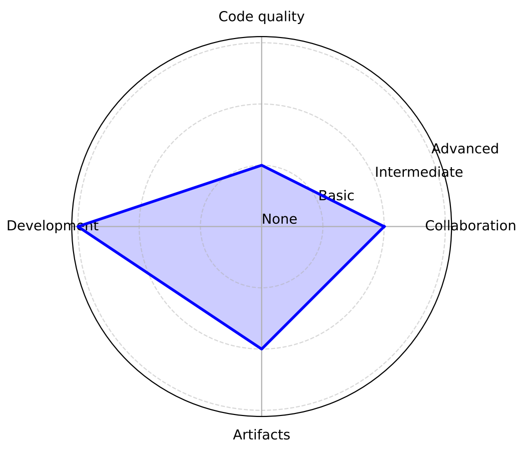


Figure 6.1: Example of a radar chart showing automation maturity

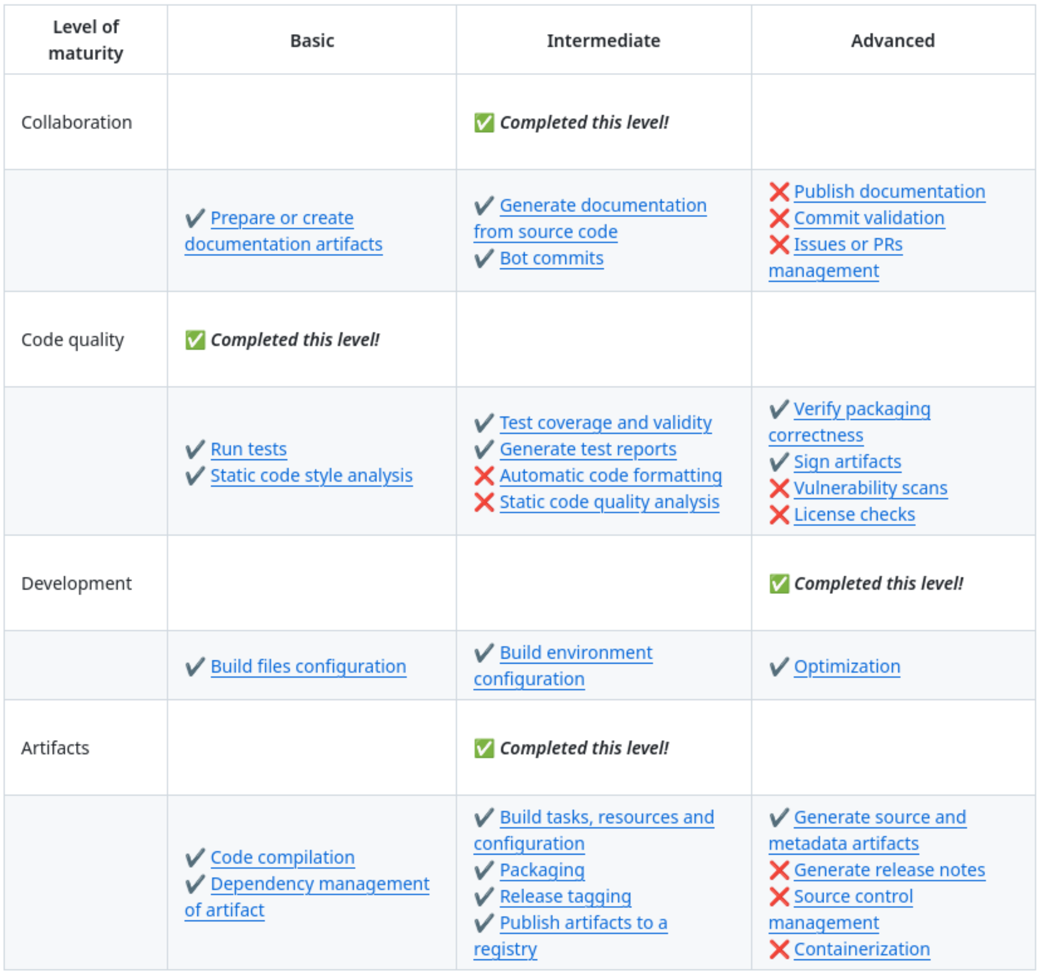
maturity level, the tasks are listed, along with a checkmark or cross mark indicating whether the task has been completed. Each task is clickable through a hyperlink that forwards the developer to a Wiki hosted on GitHub. This Wiki contains all information on each automation task, as well as the frequency of its implementation. This wiki can be found on GitHub Pages [39]. In this table, it is also easily observable which maturity level each automation domain has reached.

Finally, two automation tasks that the repository has not implemented yet are recommended. It is also listed how often these tasks are implemented by the GitHub repositories that were analyzed in this thesis. These two tasks are sorted on the following basis. Tasks belonging to the lowest maturity level are selected to be eligible for recommendation. Of those tasks, two of the tasks that have the highest implementation percentage among the analyzed repositories are suggested to the developer. At the end of the report, the developer is explicitly asked to leave some feedback based on the automation report.

The reports are automatically generated in GitHub flavored markdown [11]. To quickly post a large number of issues on the Issues boards of repositories, the submission of issues has been automated. The complete system architecture has been presented in Figure 4.1.

Dataset selection The dataset for this research question is based on the dataset that was constructed for the first research question. However, here, only Java repositories that used Maven as a build tool were considered. Since these projects are designed to be packaged and distributed, developers are most likely to be interested in providing feedback on the automation analysis. To ensure that only Maven projects are processed, all projects with Gradle files were excluded from the dataset. To create issues for the repositories in the dataset, a final filter is applied: the issues board must be turned on, allowing new issues to be submitted. In total, 5% of repositories in the original dataset are sampled to arrive at a manageable dataset within the timeframe of this thesis. This dataset is noticeably smaller, as the responses to each issue posted in the repository are manually examined.

Figure 6.2: Example of a maturity analysis table as shown in issues



Level of maturity	Basic	Intermediate	Advanced
Collaboration		✅ ***Completed this level!***	
	✔ Prepare or create documentation artifacts	✔ Generate documentation from source code ✔ Bot commits	❌ Publish documentation ❌ Commit validation ❌ Issues or PRs management
Code quality	✅ ***Completed this level!***		
	✔ Run tests ✔ Static code style analysis	✔ Test coverage and validity ✔ Generate test reports ❌ Automatic code formatting ❌ Static code quality analysis	✔ Verify packaging correctness ✔ Sign artifacts ❌ Vulnerability scans ❌ License checks
Development			✅ ***Completed this level!***
	✔ Build files configuration	✔ Build environment configuration	✔ Optimization
Artifacts		✅ ***Completed this level!***	
	✔ Code compilation ✔ Dependency management of artifact	✔ Build tasks, resources and configuration ✔ Packaging ✔ Release tagging ✔ Publish artifacts to a registry	✔ Generate source and metadata artifacts ❌ Generate release notes ❌ Source control management ❌ Containerization

Analyzing responses The goal is to have as much developer participation as possible on the GitHub issues that are posted. There are two ways to measure developer feedback: one is to ask developers to enter their response according to some words that can later be analyzed programmatically. The other approach is to adopt a more human-centered approach and allow developers to respond as they see fit. To encourage developers to respond to the issues, no restrictions are imposed on their responses by the issue itself. This way, developers feel more inclined to respond to the issues. The downside is that all issues need to be manually checked and responded to, which can be a burden on the thesis's time management.

The feedback given by developers was categorized as follows. It is verified whether the issue was marked as 'Closed' or 'Completed' by the repository maintainers. Regardless of its status, it is checked whether any of the maintainers have replied to the issue. Responses to the issue were checked to contain one or more of the following indicators:

Interest Maintainer expresses interest, either by explicitly saying so, or by responding to some concrete part of the automation analysis;

Not interested Maintainer expresses no interest, either by explicitly saying so, or by responding with a vague message or a funny image to them;

Spam Maintainer expresses that they think the issue is spam, either by renaming or categorizing the issue as spam, or by responding with a message indicating spam;

Not applicable Maintainer expresses that the issue does not apply to them by explicitly saying so;

No English response Maintainer did not reply in the English language;

Feedback Maintainer provides detailed feedback on any aspect of the posted issue. Feedback is split into three categories: feedback on the automation recognition, feedback on the model, and feedback about recognizing automations in external instances of build tools.

All replies that were provided within two weeks of the submission of the issue were replied to. Once every feedback point was discussed, the maintainers were finally asked if they were interested in using a similar tool to analyze their automation maturity. A provided example would be that the tool could be a badge in a README or embedded in one of their workflows.

6.2 Results

The third research question asked, ”How are the model suggestions perceived in practice?” To answer this research question, a large number of issues containing an analysis of a project’s automations were submitted. The subsequent discussions as reply to the issue were finally analyzed.

Replies Figure 6.3 shows the distribution of replies by developers. A total of 323 issues were posted on repositories. Exact statistics can be found in Table 6.1. While all these repositories have seen activity in the past few months, as per the dataset requirements, a

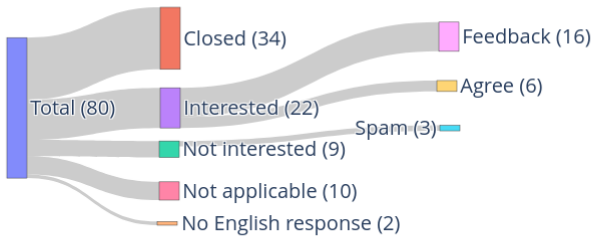


Figure 6.3: Responses to posted issues

Table 6.1: Responses and characteristics of replies. Numbers can overlap for one repository if a reply features multiple characteristics.

Metric	**Count**
Total issues	323
Closed	66
Commented	46
Not closed nor commented	241
Would use tool	17
Would not use tool	4
Repositories with feedback	23
Feedback: automation recognition	20
Feedback: model	5
Feedback: external instances	6

large number of the issues did not receive a response and were also not closed. It is assumed that these developers have not yet checked the issues in their repository, or that these repositories are no longer maintained. In this case, the distribution of these issues without a response should follow closely that of the issues with a response.

Around 10% of all posted issues were closed by the developers without replying. Their motivation remains unclear; likely, they did not appreciate the analysis being posted on their GitHub issues board. Most people who explicitly disagreed with the analysis were outspoken about it and wanted to share their opinions.

A larger number, around 15% of issues, got a reply. These maintainers were often motivated to share their opinion on the analysis of their repository. Most of these developers were interested in the automation analysis. Some of them provided feedback, and others confirmed their agreement with the analysis. A summarization of all feedback and conducted discussions in the issue boards will be provided in the following paragraph. It was interesting to note that only a few developers commented on the concept of maturity, and even then, they merely commented on the position of an automation within the maturity level.

At the same time, 19% of maintainers who commented indicated that they were not interested in the issue. Some of them thought the account that posted the analysis was a spam bot. After these initial replies, the text accompanying the issues was modified to make it sound more personal. A few other developers indicated that the analysis was not applicable, usually because the repository was merely a concept example to teach others, or because the project was not built on Maven. This could happen because they used nested dependencies that contained pom.xml files. Finally, some developers replied in a language other than English.

In their replies, half of the developers formulated feedback. The feedback did not only come from developers who indicated that they were interested, but also from developers who did not think the analysis suited their repository. This feedback was primarily divided into three areas. Most of the feedback was focused on the automation recognition. Usually, this was because the analysis did not recognize every plugin or workflow step that developers are using, which was already a known shortcoming of the card sorting. Feedback was also received on the maturity model itself, meaning the ordering of the automation tasks. Most of this type of feedback was directed towards the containerization task; during a final expert discussion, this task was combined with packaging. Finally, maintainers pointed towards external tools that they were depending on to deploy their automations.

Everyone who replied received a detailed response to their points and was finally asked whether they would find such an automation analysis useful. The exact implementation of such a tool was subject to interpretation. However, it was suggested that it could be implemented as a badge in their README, embedded in their workflow, or possibly serve as a workflow template on GitHub. Seventeen developers responded that they would find such a tool useful, while four developers did not see the benefit of such a tool.

One noticeable disadvantage of the dataset was that it included projects that were examples, as developers usually do not intend to bring these examples into production. Another disadvantage was the inclusion of several Minecraft repositories. These developers are typically self-taught, young, and not interested in automations. Due to this, their replies are less relevant to the purposes of this study.

Summarization of discussions Many discussions were held in the comments of the issues that were posted. These discussions will now be summarized.

Some developers had no interest in the study. An example was a Minecraft developer who deemed automations not relevant for a single maintainer. Someone else mentioned that the analysis was not the kind of conversation the developer was looking for. Another developer explained that they are already an automation expert and know what could be improved, but will only make those changes when they find the time.

Other developers criticized the general approach. A developer remarked that they would first need to have an interest in maximizing automations. At the same time, another mentioned that goal-driven automation would be more effective than automating for the sake of automation.

Developers also mentioned that they were used to a different approach to automation. An example is that they noted code formatting is enforced in the IDE of their developers, but is not part of their CI/CD pipeline.

Some developers pointed out that the analysis missed automations. Some mentioned that they were using Ant scripts for generating documentation, others were using tools such as Sonar, or executing automations on their own Jenkins or GitLab instances. Finally, some executed their automations in external repositories, as part of a multi-repository project structure.

Recommendations for the study were also suggested. One developer recommended scanning badges in READMEs. Another developer suggested it would be helpful to have a workflow that can serve as a starting point, which could be made available in GitHub's 'Choose a workflow' menu. Someone mentioned that for them to use the tool, there would need to be a way for the community to influence the list of CI/CD practices. They found it essential that the project would be open source, with developers submitting pull requests to modify the practices. Another developer recommended that the Wiki should contain basic information on how to add each practice to the project. Many developers mentioned that they would prefer an analysis tailored to their type of project, which can make recommendations on automation maturity based on their specific needs.

Encouragingly, some developers recognized the importance of the study and complimented the efforts as a positive step forward for the state of automation in software development. One passionate developer even stated the following about projects without automation:

> “[...] I believe projects [with barely any automations], even one-liners, when left on the public domain should be maintained to some level, otherwise we leave a lot of garbage out there. It’s the same as with the environment, we gotta clean up after ourselves if we want a good future for the next generations and not drown in trash.”

Finally, one developer even started implementing automations based on the feedback provided. This developer already had numerous basic automations in their repository. The issue recommended that they implement a static code analysis tool. The developer followed through with this feedback point and added such a tool to their pipeline. Another developer added the issue to a milestone set out for a future release.

The discussions generated many valuable comments, and developer engagement was easily achieved by actively responding to their feedback.

Quantitative analysis repositories The maturity scores for the repositories analyzed in this research step are presented in Table 6.2. The scoring metrics were previously outlined in Section 5.4.

Table 6.2: Maturity scores for analyzed repositories

Metric	**Score**
Average score	
All	0.91
Closed without comment	0.94
Interested	1.18
CMMI score	
All	0.14
Closed without comment	0.14
Interested	0.36
Joker score	
All	0.58
Closed without comment	0.62
Interested	0.77

The preferred choice for a scoring metric is either the CMMI score or the joker score, as they capture the inherent order of maturity across tasks. It can be noted that the maturity scores of repositories of developers who were interested are noticeably higher than the average repository that was analyzed. The maturity scores of every analyzed repository are quite close to the scores of repositories that closed without leaving a comment. Across the board, the joker score is indeed more forgiving for repositories. This can be noted since the scores are higher. Without the aforementioned jumping behavior of the joker score, which is responsible for incrementing multiple levels if one lower level is missed, the scores should maximally increase with $1/\#domains$. However, it is worth noting that the scores are higher, which can be attributed to the Joker's ability to increase more than one level through its jumping behavior. This way, repositories that have missed just one level are accommodated.

The perception of the model by developers has been investigated to answer the final research question. Through the engaging discussions, it became clear that many developers are interested in the analysis of their automation maturity. While plenty of feedback was received, a fair number of developers confirmed the relevance of investigating a project's state of automations. 81% of developers ($n = 17$) stated that they would be interested in using a tool to gauge their project's state of automation.

Chapter 7

Discussion

This thesis has aimed to bring order to the chaos of automations. A taxonomy was created by analyzing GitHub repositories, providing a clear picture of what developers are automating. Maturity is applied to this taxonomy to create a maturity model. The goal of this model is to direct developers towards the automations they should focus on next. Finally, the model is applied in practice to GitHub repositories to determine if it can produce valid recommendations for developers.

The taxonomy sheds light on the automation ecosystem in modern software development. At first glance, it is not clear what automations developers are using in their repositories. Using the created taxonomy, it is now possible to quickly obtain an overview of the employed automations in software repositories. It enables one to gain a better understanding of CI/CD practices and guides developers towards a deeper awareness of all possibilities.

Unpopular automations The automation ecosystem features a long tail, comprising a vast number of GitHub Actions, Bash commands, and Maven plugins that are rarely utilized. This indicates that many common automations are not standardized, resulting in projects developing their own automation solutions. Without established frameworks to guide CI/CD practices, developers often proceed with unique automation approaches. It is not immediately clear whether developers are implementing these automations for legitimate niche use cases or if they are simply reinventing the wheel. The taxonomy can reduce the fragmentation of the automation landscape by establishing commonly used tools, allowing developers to automate their projects efficiently.

Within the scope of this study, it is not feasible to analyze the long tail; however, the large number of unpopular automations gives rise to opportunities for future work. To better understand the long tail, documentation can be used to uncover the functionality and place it in the automation taxonomy. One approach to make this feasible is to build on Large Language Models (LLMs). Future work can investigate whether LLMs are capable of mapping arbitrary tools within the established automation taxonomy. Assets like documentation can be used to identify the function of these less popular automations. This thesis can serve as a foundation for such future work, where the LLM could be trained on the current taxonomy and documentation of its automations.

In future work, the distribution of the long tail can also be further investigated. The primary focus of this thesis was the distribution of the commonly used automations. However, if the popular automations are taken out of the equation, patterns might emerge in the distribution of those that are less popular. This can produce insights into niche use cases or emerging automation trends that may not be apparent when focusing solely on commonly used automation tools.

External automations This work has focused on Maven plugins and GitHub workflows. However, some developers appear to automate in other unique or unexpected ways. Some outsource automation tasks to external build servers or utilize external files that reside in other branches or repositories. Others define scripts in their workflow files or create their own GitHub Action templates that they depend on in the workflows. Future work could further investigate these external automations and analyze which automation tasks they are, in fact, implementing. This would enable an even more comprehensive suite of recognized automation instances.

The taxonomy helps developers identify potential tasks for future automation efforts. However, this thesis aims to provide even more direction to developers. The addition of maturity to the taxonomy enabled the creation of a maturity model for automations. The model can be used to gauge the level of maturity of a repository across all automation domains. This enables conducting automation efforts efficiently: new projects can start with the more fundamental automations, while projects that have already achieved the lower levels of maturity can focus on achieving higher levels. Findings suggest a positive association between repository metrics and the level of automation maturity. Better CI/CD practices lead to positive outcomes, such as increased community engagement, faster release cycles, and greater popularity. This is encouraging, as developers who follow the maturity model will be able to enhance their CI/CD practices and reap the benefits of these outcomes.

Survey effectiveness The survey was not as conclusive as hoped. There may be multiple reasons for the survey's shortcomings. Presumably, the understandability of the survey and the clarity of the information presented were misjudged. Respondents had to think about and understand many automation tasks. Within the time constraints, it may not be feasible for them to understand each task well enough to place it in the appropriate maturity level. Additionally, the concept of automation maturity is not straightforward to comprehend and apply to automation tasks. A couple of tasks were assigned to maturity levels that did not always make sense if one obtained more background information on these tasks. During an expert interview, more time is allotted to discuss the meaning of certain automation tasks, resulting in fewer unexpected maturity assignments.

Additionally, while the reward of a gift card might help encourage respondents to go through the effort of finishing the survey, malicious actors are also encouraged to cheat the system. Many spam and AI responses were identified; however, even with considerable effort, it cannot be ruled out that all malicious responses are caught.

Advanced maturity In practice, no repositories were found that met the automation standard set out by the advanced maturity level. It seems probable that a tiny fraction of repositories would have fully completed their automations; yet, this was not found.

A reason might be that automations belonging to the advanced maturity level are already rare in repositories, and to fully mature a repository, all these rare automations must be implemented. Since these automation tasks are very rare, only a few instances were encountered during card sorting. Even the more popular tools for these rare automations will quickly disappear in the long tail, simply because the automations are still rarely implemented on a larger scale. The joker score provides more leniency to meet the advanced maturity criterion; however, even when using this metric, no repository qualified for it. As suggested earlier, future work could address the problem of the long tail and identify additional instances of these rare automation tasks.

Automation evolution The concept of maturity inherently relates to chronological progression. The results showed that age has a strong correlation with maturity in younger projects. In this thesis, complete histories of repositories were not considered. However, many insights can be gained from the order in which automations are implemented. Based on the assumptions of this thesis, one would hypothesize that basic automations are generally implemented at a younger repository age than advanced automations. In GitHub, it is possible to view the history of a repository and go back in time. It would be insightful to analyze the project maturity at different points in time. Future work can investigate whether less mature automations are implemented at the beginning of a project's history. In contrast, advanced automations may not have been added yet or may have been added only recently.

The maturity model can order automations and offer direction for future automation efforts. The goal of this study is to apply the model in practice by helping developers guide their automations. This enables developers to automate their repositories in the most efficient way possible. To verify the effectiveness in practice, the model has been used to provide detailed analysis and guide automation efforts in GitHub repositories.

Deploying the model in practice offers several benefits. Developers have an inherent interest in feedback about what they are developing and will indicate whether they generally agree with the way their repository was analyzed. Additionally, since developers are often experts in their own repositories, they are most familiar with whether the assigned maturity level is correct. By validating the model's effectiveness in practice, the tool can be extended, and future work can be carried out using a similar approach.

Perception of issues After posting the issues, many developers responded in some way to the issues that were posted in their repositories. At the same time, some of these developers closed the issue without providing a comment. Their motivations would be interesting, but it is only possible to make assumptions about their choice. Many might have had a disinterest in automations or have found the issue not worth the time to respond. The average maturity score of repositories that closed the issue without commenting was lower than the score for developers who indicated interest. Possibly, developers were not motivated as they were confronted with a lack of automations. However, many positive replies were also received.

Many offered detailed feedback and interesting discussion points, and multiple ways of continuing the work on this model were suggested. The fact that many developers naturally went along with the notion of maturity being connected to automations indicates that it makes sense to them that there exists an ordering among their automations, in this case, envisioned through a maturity model. The engagement by developers should be seen as a sign of encouragement for those looking to continue research on automations.

Improving the model This research ends with the analysis of the feedback on the GitHub automation report. However, the received feedback creates the opportunity to build on the automation model in future work. The feedback points can be incorporated to create a revised automation model.

A much-received request from developers, left on the posted issues, was to support multiple types of projects. An example is that a library requires different automations than an application. The current maturity model is unable to distinguish between these use cases, instead attempting to generalize these different automations to shared tasks suitable for all types of projects. An interesting area for future work would be to gain a deeper understanding of various types of projects and analyze their distinct automation needs. This work can replicate the methodology presented in this thesis and apply it to various types of projects. The maturity model created in this thesis can serve as a core module that is applicable to any repository, and extension modules that cater to specific types of projects. Finally, the current model only verifies whether an automation has been implemented. An extended version of the model could also assess the quality of these implementations.

Threats to validity While this study offers numerous valuable insights into CI/CD practices, certain threats to validity should be acknowledged.

The results and the maturity model might not generalize well to all types of software projects. For example, the automation needs of a library are different from those of a web service. Developers who responded to the posted issues also suggested that multiple maturity models could be created to accommodate various kinds of projects. To address this need, automations that were used for distinct types of projects but had a similar goal were combined. The proposed maturity model will therefore still capture the CI/CD practices that apply across different types of projects.

To provide the most accurate automation analysis to developers, it is necessary to chart any automations that they are using. However, there exist many ways to automate a repository. While some developers use tools that are still embedded within a GitHub repository, others utilize tools that operate outside of the repository. This poses a risk to the effectiveness and completeness of the analysis. While it was found that some developers experienced inaccuracies due to their external automation tools, the majority of developers rely on the analyzed automation approaches. Less than 10% of repositories that commented on the posted issues were using such external tools. In these cases, these tools typically only cover a fraction of the automations implemented in that repository. This means that the analysis still captures the vast majority of automation practices used by most development teams.

Chapter 8

Summary

This thesis revealed the interest of developers in advancing their automation maturity and has laid a foundation for future research in guiding developers toward maturity. Through open coding and card sorting, an automation taxonomy was created that charts the current CI/CD practices. It was found that popular repositories spend more effort on automation. Insights gathered from a survey and a series of expert interviews bring order to the automation taxonomy by introducing a maturity framework. The created maturity model enables the analysis of a repository and the assessment of its automation maturity, utilizing the suggested Joker scoring metric. Actionable insights can be immediately offered to further improve the maturity balance in a repository. It has been demonstrated that the maturity of a project has a positive association with key repository metrics, such as the commit frequency. Finally, the maturity model was applied to live repositories. The state of automation of their repository was analyzed, and next automation steps were recommended. Many developers engaged in discussion, and the interest in the automation analysis is apparent. The received feedback from developers opens up further dimensions for future research. Most developers indicated that they would use a maturity model to guide their automation efforts. Strong interest from developers suggests an apparent demand for a maturity model to accelerate CI/CD practices.

Bibliography

[1] William C Adams. Conducting semi-structured interviews. *Handbook of practical program evaluation*, pages 492–505, 2015.

[2] Scott Barge and Hunter Gehlbach. Using the theory of satisficing to evaluate the quality of survey data. *Research in Higher Education*, 53(2):182–200, 2012.

[3] Hudson Borges, Andre Hora, and Marco Tulio Valente. Understanding the factors that impact the popularity of GitHub repositories. In *2016 IEEE international conference on software maintenance and evolution (ICSME)*, pages 334–344. IEEE, 2016.

[4] Hudson Borges, André C. Hora, and Marco Túlio Valente. Predicting the popularity of GitHub repositories. *CoRR*, abs/1607.04342, 2016. URL `http://arxiv.org/abs/1607.04342`.

[5] Bureau of Labor Statistics, U.S. Department of Labor. Occupational outlook handbook: Software developers, quality assurance analysts, and testers, 2024. URL `https://www.bls.gov/ooh/computer-and-information-technology/software-developers.htm`. Accessed May 14, 2025.

[6] Jailton Coelho, Marco Tulio Valente, Luciano Milen, and Luciana L Silva. Is this GitHub project maintained? measuring the level of maintenance activity of open-source projects. *Information and Software Technology*, 122:106274, 2020.

[7] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in GitHub: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on computer supported cooperative work*, pages 1277–1286, 2012.

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

[9] Adam Rafif Faqih, Alif Taufiqurrahman, Jati H Husen, and Mira Kania Sabariah. Empirical analysis of CI/CD tools usage in GitHub Actions workflows. *Journal of Informatics and Web Engineering*, 3(2):251–261, 2024.

[10] Brian Fitzgerald and Klaas-Jan Stol. Continuous software engineering and beyond: trends and challenges. In *Proceedings of the 1st International Workshop on rapid continuous software engineering*, pages 1–9, 2014.

[11] GitHub. GitHub flavored markdown specification. `https://github.github.com/gfm/`, 2023. Accessed: 2025-04-09.

[12] Anastacio Pinto Goncalves Filho, Jose Celio Silveira Andrade, and Marcia Mara de Oliveira Marinho. A safety culture maturity model for petrochemical companies in Brazil. *Safety science*, 48(5):615–624, 2010.

[13] M. Lokesh Gupta, Ramya Puppala, Vidhya Vikas Vadapalli, Harshitha Gundu, and C. V. S. S. Karthikeyan. Continuous integration, delivery and deployment: A systematic review of approaches, tools, challenges and practices. In Gangamohan Paidi, Suryakanth V. Gangashetty, and Ashwini Kumar Varma, editors, *Recent Trends in AI Enabled Technologies*, pages 76–89, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-59114-3.

[14] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[15] CMMI Institute. CMMI Institute - Home — cmmiinstitute.com. `https://cmmiinstitute.com/`, 2024. [Accessed 23-09-2024].

[16] Mehdi Keshani, Tudor-Gabriel Velican, Gideon Bot, and Sebastian Proksch. Aroma: Automatic reproduction of maven artifacts. *Proceedings of the ACM on Software Engineering*, 1(FSE):836–858, 2024.

[17] Timothy Kinsman, Mairieli Wessel, Marco A Gerosa, and Christoph Treude. How do software developers use GitHub Actions to automate their workflows? In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 420–431. IEEE, 2021.

[18] Fang Liu, Dan Yang, Yueguang Liu, Qin Zhang, Shiyu Chen, Wanxia Li, Jidong Ren, Xiaobin Tian, and Xin Wang. Use of latent profile analysis and k-means clustering to identify student anxiety profiles. *BMC psychiatry*, 22:1–11, 2022.

[19] Patrick E McKight and Julius Najab. Kruskal-wallis test. *The corsini encyclopedia of psychology*, pages 1–1, 2010.

[20] Jorge Melegati and Eduardo Guerra. DAnTE: a taxonomy for the automation degree of software engineering tasks. In *Generative AI for Effective Software Development*, pages 53–70. Springer, 2024.

[21] Tim Menzies. Automated software engineering. *Automated Software Engineering*, 2024. URL `https://link.springer.com/journal/10515`.

[22] Albert R. Meyer and Eric Lehman. Chapter 7: Partial orders. `https://dspace.mit.edu/bitstream/handle/1721.1/104426/6-042j-spring-2010/contents/readings/MIT6_042JS10_chap07.pdf`, 2010. URL `https://dspace.mit.edu/handle/1721.1/104426`. Course Notes from MIT 6.042J: Mathematics for Computer Science.

[23] Samer I Mohamed. Software release management evolution-comparative analysis across agile and DevOps continuous delivery. *International Journal of Advanced Engineering Research and Science*, 3(6):236745, 2016.

[24] Gerard O'Regan. *Capability Maturity Model Integration*, pages 255–277. Springer International Publishing, Cham, 2017. ISBN 978-3-319-57750-0. doi: 10.1007/978-3-319-57750-0_16. URL `https://doi.org/10.1007/978-3-319-57750-0_16`.

[25] Mark C Paulk. A history of the capability maturity model for software. *ASQ Software Quality Professional*, 12(1):5–19, 2009.

[26] Kasim Randeree, Ashish Mahal, and Anjli Narwani. A business continuity management maturity model for the UAE banking sector. *Business Process Management Journal*, 18(3):472–492, 2012.

[27] Thalita Laua Reis, Maria Augusta Siqueira Mathias, and Otavio Jose de Oliveira. Maturity models: identifying the state-of-the-art and the scientific gaps from a bibliometric study. *Scientometrics*, 110:643–672, 2017.

[28] Caroline Roberts, Emily Gilbert, Nick Allum, and Léïla Eisner. Research synthesis: Satisficing in surveys: A systematic review of the literature. *Public Opinion Quarterly*, 83(3):598–626, 11 2019. ISSN 0033-362X. doi: 10.1093/poq/nfz035. URL `https://doi.org/10.1093/poq/nfz035`.

[29] Patrick Schober and Thomas R Vetter. Nonparametric statistical methods in medical research. *Anesthesia & Analgesia*, 131(6):1862–1863, 2020.

[30] Mali Senapathi, Jim Buchan, and Hady Osman. Devops capabilities, practices, and challenges: Insights from a case study. *CoRR*, abs/1907.10201, 2019. URL `http://arxiv.org/abs/1907.10201`.

[31] Shipra Sharma and Balwinder Sodhi. SEAT: A taxonomy to characterize automation in software engineering. *arXiv preprint arXiv:1803.09536*, 2018.

[32] Eleanor Singer and Cong Ye. The use and effects of incentives in surveys. *The ANNALS of the American Academy of Political and Social Science*, 645(1):112–141, 2013.

[33] Prabath Siriwardena. *Maven Essentials*. Packt Publishing Ltd, 2015.

[34] Donna Spencer and Todd Warfel. Card sorting: a definitive guide. *Boxes and arrows*, 2(2004):1–23, 2004.

[35] Anselm L Strauss and Juliet Corbin. Open coding. *Social research methods: A reader*, pages 303–306, 2004.

[36] SCAMPI Upgrade Team. Appraisal requirements for CMMI, version 1.2 (ARC, v1. 2). *Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University*, 2011.

[37] Pablo Valenzuela-Toledo and Alexandre Bergel. Evolution of GitHub Action workflows. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 123–127. IEEE, 2022.

[38] Pablo Valenzuela-Toledo, Alexandre Bergel, Timo Kehrer, and Oscar Nierstrasz. The hidden costs of automation: An empirical study on GitHub Actions workflow maintenance. In *2024 IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*, pages 213–223. IEEE, 2024.

[39] Simcha Vos. Automation thesis wiki. `https://github.com/simchavos/automations/wiki/Automation-Thesis-Wiki`, 2025. Accessed: 2025-05-25.

[40] Simcha Vos. CIMMI: A maturity model for CI/CD practices, 2025. URL `https://doi.org/10.5281/zenodo.15388799`.

[41] Yuqing Wang, Mika Mäntylä, Zihao Liu, and Jouni Markkula. Test automation maturity improves product quality – quantitative study of open source projects using continuous integration, 2022. URL `https://arxiv.org/abs/2202.04068`.

[42] Roy Wendler. The maturity of maturity model research: A systematic mapping study. *Information and software technology*, 54(12):1317–1339, 2012.

[43] Jed R Wood and Larry E Wood. Card sorting: current practices and beyond. *Journal of Usability Studies*, 4(1):1–6, 2008.

[44] Thomas Zimmermann. Card-sorting: From text to themes. In *Perspectives on data science for software engineering*, pages 137–141. Elsevier, 2016.