Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät
Institut für Informatik

Commit Conventions: Conception and Impact of the Conventional Commit Specification in Open Source

Bachelorarbeit

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von: Anna Freidl
geboren am: 17.01.1998
geboren in: Gräfelfing

Gutachter: Prof. Dr. Lars Grunske
Prof. Dr. Timo Kehrer

eingereicht am: ..... verteidigt am: .....

Abstract - In modern software development, clear and standardized commit messages are essential for the collaboration and maintainability of projects. This thesis investigates the adoption and impact of the Conventional Commits Specification (CCS) in open-source software projects. By utilizing a dual-method approach consisting of keyword-based document analysis and pattern recognition of commit messages, we examined the prevalence, consistency, and implications of CCS across a diverse dataset of repositories. The findings indicate that while CCS adoption remains limited, it demonstrates a growing trend, particularly among projects using modern languages such as TypeScript and Rust. The analysis revealed that CCS adoption can lead to more structured and descriptive commit practices, enhancing project maintainability and collaboration. Further, this study highlights common custom commit types and explores the potential for expanding CCS guidelines to accommodate evolving developer practices. Our findings provide a foundational overview for future research to deepen the understanding of CCS, its barriers, and its long-term benefits for software development practices.

Contents

1 Introduction

In software engineering, effective communication and code maintainability are paramount for the success of software development projects [21]. In collaborative environments such as open-source software development, these factors become even more important due to the involvement of numerous contributors. Typically, collaborative software development projects are supported by version control systems (VCS) such as Git [29, 30]. A VCS not only stores all code changes but also manages concurrent access to development artefacts [30]. Within these systems, commits and their associated messages serve as a historical record of changes, providing essential context for the modifications made to the code base.

Commits are snapshots of the codebase at specific time points, and their messages are crucial for documenting the ongoing development of a software project. They allow developers to track the history of changes and the reasons for the changes, and facilitate collaboration between team members. For long-lived projects, such as the Linux kernel[1], commit messages can serve as the main source of information for future developers trying to understand what changes were made and why they were made when the original maintainers have left the project [26]. Poorly structured or inconsistent commit messages can therefore hinder collaboration, make debugging more difficult, and impede the automation of development processes [31].

In collaborative software projects, especially with a large number of collaborators, creating clear and consistent commit messages is a challenging. Such inconsistencies can lead to problems in tracking the progress, understanding the reasons for changes and onboarding new contributors who rely on commit logs to familiarise themselves with the project. Additionally, automation tools rely on well-structured commit messages to function properly. Therefore, irregular commit practices not only affect human readability, but also undermine the effectiveness of the automated processes designed to improve the maintainability and efficiency of the project.

The software development community has recognized these challenges and introduced various guidelines and conventions for writing commit messages to improve clarity and consistency [6, 12, 15]. An important evolution in this area is the emergence of the **Conventional Commits Specification (CCS)** [4] a standardised message format for documenting code changes. Invented by the open-source community in 2017, CCS offers a structured approach to improve the clarity and consistency of commit histories. By adhering to a predefined format, the CCS facilitates comprehensibility and readability.

The CCS differs from other commit conventions by having a clear and machine-readable structure, which increases the potential for automating release processes and improving communication between developers. The specification can also be used across all lan-

[1] https://github.com/torvalds/linux

guages and projects. By standardizing the way commit messages are written, CCS aims to eliminate the inconsistencies and inefficiencies associated with unstructured commit messages. For example, using a standardized format allows tools to automatically create changelogs, determine semantic version updates [22], and enforce policies for commit messages.

Although the importance of commit messages is widely recognized, previous research has mainly investigated the quality of commit messages [11], the categorization of software maintenance activities [25], and the impact of commit message quality on software maintenance [27]. For instance, Dyer et al [11] found that a significant proportion of commit messages in open source projects lacked descriptive content, which can hinder understanding and collaboration. However, these studies did not fully consider the impact and adoption of introducing standardized commit message conventions such as CCS in software projects. The structured nature of Conventional Commits (CC) can provide more consistent and informative data, which in turn can influence the results of previous studies by allowing a more accurate analysis of software development practices. For example, if commit messages are in a standardized format, automated tools that rely on commit messages may work more effectively and potentially change the conclusions about programmer behavior, code quality, and project maintainability drawn from commit history analysis. Given the lack of research on the actual use and adoption of CC in software projects, there is a need to systematically investigate the impact of CCS adoption on project commit behavior and maintainability. Understanding the level of adoption of CCS and the consistency of its use can provide valuable information about its effectiveness and its impact on development practices.

This thesis attempts to fill this gap by examining the adoption of CCS in different projects, analyzing the changes in commit behaviors associated with its implementation, and assessing their influence on developer practice. Through empirical analysis of real open source repositories and commit behaviour, this descriptive study primarily aims to provide empirical evidence of the adoption rates of CC, evaluate the consistency of application across different project dimensions, and assess the tangible effects of CCS application on the behaviour of committers.

By contributing to a better understanding of the effectiveness of standardized committing conventions, the results of this research can influence the development of tools and guidelines to promote the consistent and effective use of Conventional Commits and ultimately improve communication, maintainability, and project outcomes within open source communities.

The subsequent sections of this thesis are organized as follows: Section 2 provides an overview of the history of commit conventions as well as a deeper insight into the CCS and related work. Section 3 outlines the research methodology, including the formulation of research questions. Section 4 presents the results of our analyses, Section 5 discusses these findings, and Section 6 concludes the work and suggests directions for future research.

2 Background and Related Work

Commit messages are an essential part of collaborative software development and serve both as a communication tool between developers and as a historical record of a project's development. To fully understand the role and impact of standardized commit conventions such as Conventional Commits Specification (CCS), it is important to explore the historical evolution of commit message practice and the origins of structured specifications. This section begins with an overview of the **History of Commit Conventions** in section 2.1, which traces the origins and evolution of commit messaging practices and highlights key milestones that have shaped current standards. Subsequently, section 2.2 discusses the **Conventional Commit Specification (CCS)**, describing its framework, principles and importance for promoting consistency and clarity of commit histories.

2.1 History of Commit Conventions

The practice of writing commit messages has evolved significantly since the early days of software engineering. In 1976, Swanson [25] established a basic framework by categorising software maintenance activities into three distinct types: corrective (fixing errors), adaptive (accommodating changes), and perfective (enhancing the software). This typology has proved to be remarkably durable, and subsequent research [16] [19] has consistently returned to these categories as a way of understanding and classifying the nature of changes to code. Current work in the field can be divided into 3 groups of general approaches: Categorizing Commits [19], Developing Commit Conventions [27], Automated Tools and Advanced Applications [22] [24].

Building on Swanson's work [25], Mockus and Votta [19] took a closer look at the relationship between maintenance activities and commit messages. They identified three reasons for code changes that overlapped with Swanson's categories and thus identified specific keywords that often signalled the nature of a change. For example, terms such as "fix", "bug" and "error" often indicate corrective commits, while "add" and "new" often indicate adaptive changes. This research highlighted the potential for using commit messages as a valuable source of information about the reasons behind code modifications.

Today, the benefits of commit messages are well known, serving as historical records, facilitating collaboration, and helping with debugging and maintenance [31]. A well-written commit message provides context for why a change was made and what problem it solves. This not only makes it easier for others to understand the codebase, but also helps future developers understand the rationale behind past decisions [27].

Despite the recognized importance of good commit messages Dyer et al. [11] found in their 2015 study that about 14% of commit messages in over 23,000 Open Source projects

were completely blank, 66% of messages contained only a few words, and only 10% of commits contained messages with descriptive English sentences. Although there has been some research since then on how to improve the quality of commit messages [27], Ma and Lopez found in a recent study [17] that about 95% of commit messages in student projects had less than one descriptive sentence (less than 15 words), which is even worse than in Dyer et al.'s paper (90%) [11].

Recognizing the need for improvement, several projects have developed their own guidelines and conventions. Early efforts, such as the Angular [6], jshint [15], and ember guidelines [12], laid the groundwork by suggesting structured formats, typically including a type, scope, and short summary. These formats aimed to make commit messages more concise, consistent, and easier to parse for both humans and machines. As illustrated in Figure 1, the variation in these commit conventions highlights the necessity for a standardized specification to unify practices across diverse projects. This evolution led to the development of the Conventional Commits Specification, a more general and widely used standard that simplifies the Angular conventions in practice, and simply specifies the essentials of commit message conventions.

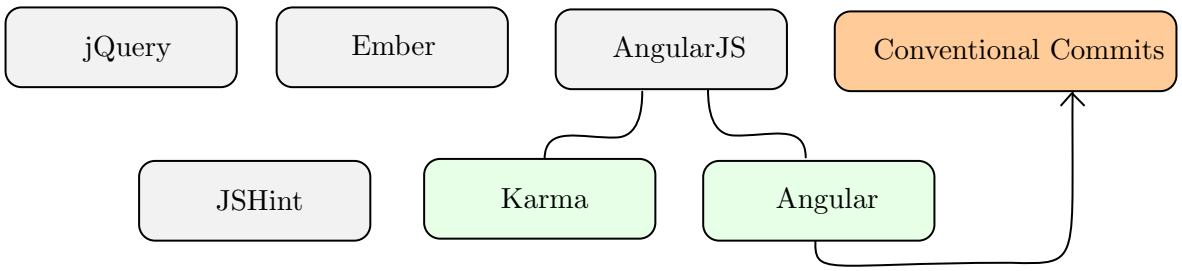

Figure 1: Commit Convention Variations

2.2 Conventional Commit Specification

The search for a uniform and standardized commit message format led to the development of the Conventional Commits Specification (CCS) [4]. CCS was formalized in 2017 in response to the patchwork landscape of commit conventions and aimed to create a clear and consistent framework that can be used universally in open source and commercial software projects. By introducing a standardized format, Conventional Commits (CC) are meant to improve the readability and automation capabilities of commit histories and enable better collaboration and more efficient project management.

The CCS simplified the structured formats introduced by previous projects by defining the essential components of a commit message. Essentially, the specification prescribes a format that categorizes commit messages into types, each of which represents a different

type of change. The standard types defined by CCS and the corresponding associated code changes are as follows:

1. **feat (Feature)**: Introduces a new feature or functionality to the project.
   Example: Adding a new user authentication module.

2. **fix (Bug Fix)**: Resolves a bug or issue in the codebase.
   Example: Correcting a null pointer exception in the login process.

3. **docs (Documentation)**: Involves changes or additions to the project's documentation.
   Example: Updating the README file with installation instructions.

4. **style**: Pertains to changes that do not affect the functionality or logic of the code, such as code formatting and styling adjustments.
   Example: Reformatting code to adhere to a consistent indentation style.

5. **refactor**: Involves code changes that improve the code structure, readability, or maintainability without fixing a bug or adding a feature.
   Example: Renaming variables for better clarity or restructuring a function to reduce complexity.

6. **perf (Performance)**: Enhances the performance of the application without altering its functionality.
   Example: Optimizing a database query to reduce execution time.

7. **test**: Adds or modifies tests to ensure code reliability and coverage.
   Example: Implementing unit tests for a newly added feature or fixing existing tests that were failing.

8. **build**: Involves changes that affect the build system or external dependencies, including build scripts and package managers.
   Example: Updating the webpack configuration or adding a new dependency to the project.

9. **ci (Continuous Integration)**: Pertains to changes in continuous integration configuration files and scripts, such as CI pipelines and workflows.
   Example: Updating the Travis CI configuration to include new build steps or adding a CircleCI workflow.

10. **chore**: Encompasses routine tasks and maintenance activities that do not modify the ‘src‘ or ‘test‘ directories.
    Example: Updating project dependencies, cleaning up obsolete files, or configuring project settings.

11. **revert**: Reverts a previous commit, effectively undoing changes introduced by that commit.
    Example: Rolling back a feature that caused unforeseen issues in the application.

As shown in Figure 2, each commit message begins with one of these types, followed by a short description of the change. Optionally, additional information such as the scope of the change or a more detailed text can also be added. The Example message indicates that a new feature (`feat`) related to authentication (`auth`) has been added, in particular the implementation of login functionality using JWT (JSON Web Tokens).

```
<type>[optional scope]: <description>
[optional body]
[optional footer(s)]
```

```
feat(auth): add JWT-based authentication for API requests
Implemented JWT authentication for secure API access. Added middleware to handle
token validation and user authentication.
Reviewed-by: Alice
Refs: #456, #789
```

Figure 2: Conventional Commit Format and an Example

This structured approach not only guarantees consistency in commit messages, but also ensures that they contain the necessary information to be effectively analyzed by both humans and automated tools [22]. For example, change log creation and semantic versioning tools can easily interpret CC messages to automate these processes and reduce manual effort for developers.

The adoption of Conventional Commits has been driven by the ability to overcome the shortcomings of previous commit message processes. Studies have repeatedly shown that a significant proportion of commit messages are missing essential information, which can hinder collaboration and maintenance efforts [11, 17]. By enforcing a standardized format, CC help mitigate these issues by ensuring that commit messages are both informative and structured. This consistency is particularly beneficial in large and diverse projects, where multiple contributors may have different habits and experience in writing commit messages.

The emergence of standardized commit message formats has also led to the development of automated tools such as semantic-release [3], which uses structured commit messages to automate semantic versioning and release processes. Semantic versioning (SemVer) [22] is a widely used versioning scheme in which the version numbers consist of three levels: major.minor.patch e.g. version 2.1.0. The major version indicates a breaking change, a minor version jump indicates the addition of a new, downward-compatible function, and a patch version jump indicates a bug fix. The Conventional Commits Specification (CCS) complements SemVer by describing the features, fixes and breaking changes in commit messages in detail, thus facilitating the automatic determination of semantic

version jumps.

In addition, the Commit Conventions Specification in the security domain has already been further developed into so-called Security Commit Messages (SECOM) [24]. SECOM contain security information that are essential for the detection and assessment of vulnerabilities based on commit messages. The idea is for developers to add the CC-type `security` extended by SECOM to their commit message when making security-critical changes. However, the effectiveness of such tools and their impact on real-world development workflows have not yet been fully explored.

This work is one of the first in this field to address the adoption of the Conventional Commit Specification and thus aims to quantify the concrete impact of CC on the way developers structure and document their code changes.

3 Methodology

The aim of this study is to investigate the prevalence and impact of the Conventional Commits Specification (CCS) in open source projects. To achieve this goal, we focus on two central research questions.

3.1 Research Questions

RQ1: *To what extent and with what consistency are Conventional Commits adopted in open source projects?* The motivation behind this question lies in the assumption that the adoption of standardized commit message conventions such as CCS improves code collaboration and maintainability. However, it is unclear to what extent CCS is actually used in open source projects. A deeper understanding of this can reveal patterns and factors that influence adoption, such as the programming language used, the size of the project or its age. Thereby, we may gain insight into the factors that might influence the adoption of standardized commit procedures. Understanding the adoption patterns of CCS can motivate future work to identify barriers to standardization and develop strategies to promote more consistent use of commit message standards across different projects.

RQ2: *How does the adoption of Conventional Commits influence commit behavior before and after their adoption?* If the introduction of CCS leads to more granular and targeted commits, this would confirm the effectiveness of standardized commit conventions in improving development workflows. By examining changes in commit behavior associated with the adoption of CCS, we evaluate their impact on aspects such as commit frequency, code changes, and number of files changed. These findings are important to validate the effectiveness of CCS and for developing future best practices for implementation in different project environments.

To address the two research questions, we conduct a multi-method study comprising three stages. Section 3.2 provides an overview of the data collection procedures and describes our approach to classify commit messages according to CCS. Secondly, we describe the process of identifying the date of CCS adoption and ascertain whether a repository can be identified as conventional. In section 3.3 we describe the methodology we use to analyze the adoption and consistency of CC application through the enriched repositories. In Section 3.4, we describe our methodology to identify commit patterns and discrepancies to gain insight into RQ1. We take a systematic approach to categorize commits, analyze key metrics such as changed files, insertions, deletions, and commit intervals, and evaluate adoption rates across several dimensions, including programming language, project size, and age. Finally, we address RQ2 in Section 3.5, by systematically analyzing the changes in commit behaviors before and after the adoption of CCS by comparing metrics across two distinct periods. Figure 3 displays an overview of this approach. The appendix contains the complete Python code, which we describe subsequently.

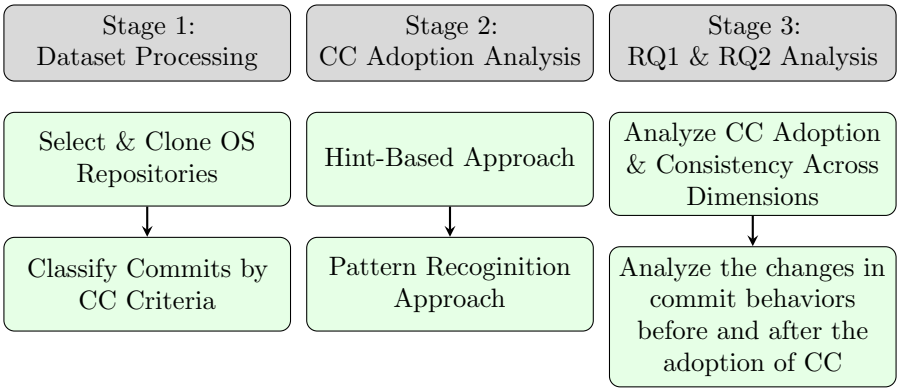

Figure 3: Research Process for Conventional Commits Adoption and Analysis

3.2 Data Collection

To ensure that our dataset contains sufficient high-quality commit messages, we select popular open-source projects with a high level of collaboration. We assume that these projects have frequent updates and a substantial number of commits, which are essential to investigate the usage of Conventional Commits (CC) in real-world code. We sample our dataset from GitHub because Git is by far the most widely used version control system (VCS) [14] and GitHub is the most popular hosting site for public Git repositories [5].

We select the top 10 most-used programming languages [1] on GitHub based on the number of stars (similar to upvotes or likes) that received by projects written in these languages in the first quarter of 2024. The selected languages are: Python, JavaScript, Go, C++, Java, TypeScript, C, C#, PHP, and Rust. For each language we sample 250 of the most popular projects based on their GitHub "star" rating. The dataset therefore comprises 2,500 repositories. However, during the analysis process, we exclude certain

repositories, such as those that were later deactivated by GitHub, moved or were set to private, leaving a total of 2,495 unique projects with a median star count of 16,309 and an average of 368 contributors. Table 1 shows an overview of our selected repositories.

Language	No. of Projects	Avg. Contributors	Avg. Stars	Avg. Age in Years
Python	249	428	37,620	6.5
JavaScript	249	340	35,902	9.7
Go	250	347	23,344	8.1
C++	249	399	19,777	9.0
Java	250	256	20,310	9.1
TypeScript	250	507	34,520	7.1
C	249	500	13,798	9.2
C#	249	172	9,874	8.3
PHP	250	272	9,656	10.8
Rust	250	286	15,101	5.9

Table 1: Summary of Projects by Programming Language

To facilitate our analysis, we use the GitHub REST API [2] to retrieve rich metadata for each repository in our dataset. We retrieve this metadata in JSON format and include various attributes such as the repository name, default branch, primary programming language, homepage-url and important flags that indicate certain repository characteristics. One important attribute we use is the `has_wiki` flag for further analysis, which indicates whether a repository has an associated wiki, and the `clone_url` for each repository, which we use to clone the repositories into our local environment.

We clone the repositories and extract the commit information directly from the Git logs using the `GitPython` [13] library to interact with Git repositories through Python scripts, rather than utilizing the GitHub REST API. This decision is primarily driven by several factors relating to practicality and methodology. Firstly, by cloning repositories we ensure comprehensive access to all commit data without being constrained by the rate limiting of the GitHub REST API. Furthermore we analyze a local copy of the repository, bypassing potential delays and access restrictions, thereby providing a more efficient and scalable data extraction process. The additional benefit of analyzing Git logs locally is that it allows us more granular and flexible data manipulation. This enables the extraction of specific metrics, such as the number of files changed, insertions, and deletions per commit, with greater accuracy and speed. To organize the repositories efficiently, we structure the directory by grouping repositories based on their programming language. For further analysis, we also clone the wiki repositories of those wiki flag points to an existing wiki repository.

After cloning the repositories, we collect all commits from the default branch for each repository starting at the creation of a repository until October 2024 using Git

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

commands, resulting in 24,067,163 commit messages. We chose to pull only from the standard branch to ensure consistency between the repositories, since this is usually the primary development branch. This approach minimizes the inclusion of experimental or outdated branches and provides a clearer overview of the main commit history relevant to project development and CCS implementation. Since we are interested in the CCS adoption of human developers in this study, we focus on human-written commit messages by eliminating commit messages that were automatically generated by tools. To eliminate bot messages, we use Bot Identification by Author-Name (BIN), a pattern-based method from the bot detection technique BIMAN by Mousavi et al. [20]. Table 2 shows several patterns of bot messages that we exclude from our dataset. We identify these patterns both at commit level and at author level. For example, the first pattern example shows a commit message generated automatically during a merge process in Git. When a developer merges changes from one branch (e.g. feature/login) into another (e.g. main), Git generates a merge commit to document the integration of these branches. The message `"Merge branch 'feature/login' into 'main'"` is the default Git message, unless the developer specifies their own message. Merge commits often do not contain new code changes that were written at the time of the merge, but represent the consolidation of existing changes from different branches. At author level, we identify bot commits by the fact that their author name contains bot, embedded between non-alphabet characters, e.g. dependabot [bot]. The commit messages are generated by Dependabot [3], an automated tool provided by GitHub to keep project dependencies up to date. Dependabot scans a project's dependency files, such as package.json or requirements.txt, and automatically creates pull requests to update the dependencies when new versions are available. Since developers can also fill merge commits with a message themselves, we also exclude commit messages where no files have been changed, as this indicates merge commits or empty commits, e.g. related to versioning.

The resulting dataset comprised 19,403,873 commits, with a median of 1,939 commits per repository. For every commit, we extract the following data:

- Commit timestamp and date
- Commit message
- Author's name
- Number of files changed, insertions, and deletions

To gather this information, we use the `git log` command, which allows us to retrieve detailed commit metadata directly from each repository. By specifying custom formatting options in `git log`, we ensure that all necessary information is extracted efficiently and consistently across the dataset.

[3] https://github.com/dependabot

No.	Pattern Example	Description
1	`Merge branch 'feature/login' into 'main'`	Automated merge commits without file changes.
2	`git commit --allow-empty -m "Marking release start"`	Empty commits intentionally created to mark specific events.
3	`v1.2.3`	Tagging a new version without modifying files.
4	`Merge remote-tracking branch 'origin/develop' into 'main'`	Merge commit from a remote-tracking branch without file changes.
5	`[maven-release-plugin] Prepare release v1.0.0`	Automated commits by the Maven Release Plugin without direct file changes.
6	`dependabot[bot]`	Matches author names where 'bot' is preceded by a non-word character or digit.
7	`bot-builder`	Matches author names where 'bot' is followed by a non-word character or digit.
8	`my-bot-account`	Matches author names where 'bot' is surrounded by non-word characters or digits.

Table 2: Patterns of bot messages that were excluded from the dataset

3.3 Identification of Conventional Commit Repositories

This section addresses RQ1 by describing the methods we use to identify repositories that have adopted the Conventional Commits Specification (CCS).

3.3.1 Overview of the Dual Approach

To evaluate the adoption and application of the CCS in projects from two perspectives, we use a dual approach. First we perform a *keyword-based document analysis*, by scanning project documentation for evidence of CCS adoption. Second, we conduct an *Analysis of Commit Message Patterns* to identify CCS structures in commit messages. This methodology enables us to categorize repositories based on their structural compliance with the Conventional Commit (CC) guidelines and contextual evidence of CC usage in their documentation or configurations.

3.3.2 Keyword-Based Document Analysis

We assume that large projects included in our dataset are likely to be more comprehensive due to their larger developer base and more frequent updates, and that such projects are likely to have more comprehensive contributing policies. On this assumption, we hypothesize that repositories that wish to use Conventional Commits (CC) will mention them in some form within the repository.

To identify repositories that may mention the Conventional Commits Specification (CCS) in their documentation or utilize tools to enforce CCS, regardless of whether it has been applied, we employ a hint-based approach. In this approach, we look for explicit references to the use of CC in the documentation, configuration files, Git hooks, and associated repository websites. Table 3 shows a summary of the locations we examine and keywords we search for. This table provides an overview of the components we examine and the considerations behind them, supplementing the description of our hint-based approach. For example, we check for the presence of `commitlint` or `commitizen` in Git hook scripts, which are usually found in the `.husky/` or `.git/hooks/` directories. These hooks, which are often configured to run automatically before every commit, enforce CCS compliance by checking commit messages for a specific format. For example, a `commit-msg` hook can reject any commit message that does not follow the Conventional Commits format, ensuring consistent and standardized commit practice across the entire repository.

Location Checked	Keywords Searched For
Documentation Files / Wiki Files / Project Website or Homepage	"Conventional Commits", "Conventional Commit", "Conventional Changelog", "Commit Message Convention", "www.conventionalcommits.org", "commitizen", "commitlint", "semantic-release"
Configuration Files	Presence of files: `commitlint.config.js`, `.commitlintrc`, `.commitlintrc.js`, `.commitlintrc.json`, `.cz-config.js`, `.czrc`, `.versionrc`
Dependency Files	Dependencies on "commitizen", "commitlint", "standard-version", "semantic-release"
Git Hook Scripts	Content of hook scripts in `.husky/` and `.git/hooks/` directories containing "commitlint", "commitizen"
Project Website or Homepage	Evidence of CC use, mentions of CC practices, badges indicating adherence to CC standards

Table 3: Keywords and Phrases Used in the Hint-Based Approach

By examining these elements, we aim to identify projects where developers have committed to CCS compliance made efforts to integrate CC practices into the development workflow.

Our methodology systematically searches for specific keywords and phrases that indicate the use of CC within different components of each repository. These keywords come primarily from the official Conventional Commits website [4], which references tools and practices relevant to CC adoption. By sourcing our keywords from this authoritative CC resource, we ensure that our search targets are relevant and recognized indicators of CC implementation .

3.3.3 Analysis of Commit Message Patterns

To identify repositories that adhere to the CCS, we implement a pattern-matching approach that focuses on analyzing the structure of commit messages. As Boehm et al. [7] points out, examining software quality at the commit level can provide valuable insights into how individual changes affect the development of the software as a whole. This method allows us to objectively evaluate the use of CC by examining the actual commit history, regardless of whether CC is explicitly mentioned in the documentation or in configuration files.

In the first step, we analyze each commit message to see if it matches the CC format. According to the CCS, a Conventional Commit message has the following structure (cd. Section 2.2): a mandatory type indicating the type of change (e.g., `feat` for new features, `fix` for bug fixes), an optional scope in brackets specifying the affected section of the codebase, an optional breaking change indicator denoted by an exclamation mark after the type or scope, and a mandatory short description summarizing the change made.

Using regular expressions, we analyze commit messages and extract relevant components. We use a specific pattern designed to capture the type, optional scope, breaking change indicator, and description. This allows us to systematically analyze and parse the commit messages across different repositories.

Commit messages are classified on their adherence to the CC format and the types used. A commit is considered conventional if it follows the CC structure and uses one of the standard types by definition of the CCS. The standard types include `feat`, `fix`, `docs`, `style`, `refactor`, `perf`, `test`, `build`, `ci`, `chore`, and `revert`. Recognizing that developers might adopt custom types to suit the specific needs of their projects, we also classify commits with custom types that follow the CC structure as Conventional Commits but are not using the CC standard types as seen in Figure 4. Continuing with the results of this analysis, we classify each commit in each repository according to whether it is conventional, i.e. whether it corresponds to the CC structure, and whether it is a standard type or custom type CC. In addition, we collect information about the number of standard types and custom types for all commits in a repository during iteration. This gives us a complete overview of the commits, the distribution of the Conventional Commits with standard type as well as the user-defined and Non-Conventional Commits and the distribution of the user-defined types.

3.3.4 Analysis of the Adoption Date

While identifying individual Conventional Commits, including those with custom types, provides insight into the use of CC conventions, we focus on determining the formal adoption of the standard CCS. For calculating the Conventional Commit Proportion (CCP) to determine repository classification, we focus on standard CC types to maintain

```
Commit 6b46e37
Date: Feb 6, 2024
```

```
operator: Integrate support for OpenShift-managed credentials in Azure
(#11868)
```

Figure 4: Example of a Commit Message in CC Format With Custom Type

consistency across repositories because the strength of the CCS lies precisely in the manageability of the commit types and we do not know at this point whether the custom commit types are a deliberate extension of the commit types. Excluding custom types from the adoption rate calculation ensures that the rate reflected adheres to the standard CCS and provided a consistent metric for comparing repositories.

To identify whether a repository has adopted the CCS, we use a two-step approach based on the (CCP) in its commit history. We define the CCP as the percentage of Conventional Commits with standard CC types relative to the total number of commits in the repository:

$$\textbf{Conventional Commit Proportion (\%)} = \left(\frac{\text{Number of Conventional Commits with Standard Type}}{\text{Total Number of Commits}} \right) \times 100$$

First, we classify a repository as **consistently conventional** if it has a CCP of 80% or more. No generally acknowledged source explicitly states 80% as the threshold for classifying a repository as a 'Conventional Commit (CC) project'. However, the 80% threshold is often cited in research as a rule of thumb for 'majority acceptance' or 'sufficient coverage' indicating consistent use of Conventional Commits throughout the project's history and suggesting formal adoption of the CC specification from the outset [8, 10, 18, 23, 32]. For consistently conventional repositories, we record the creation date of the repository as the adoption date.

For repositories that do not meet the 80% threshold, we analyze the commit history to determine the date of CC adoption using change point detection with binary segmentation [28]. Change point detection is a statistical method that identifies points in a time series where statistical properties change significantly. In our context, a change point refers to a point in the commit sequence where there is a noticeable abrupt change in commit practices - more specifically, a sudden increase in the use of standard Conventional Commits.

We implement change point detection using the `ruptures` library [9], focusing on detecting increases in the proportion of Conventional Commits. Although binary seg-

mentation can detect both increases and decreases in statistical properties, our analysis specifically targets significant increases that would indicate the adoption of CC practices. The process of change point detection involve the following steps:

1. **Representing the binary sequence:** We represent the commit history as a binary sequence, where 1 represents a Conventional Commit with a standard type, and 0 represents the remaining commits.

2. **Application of binary segmentation:** The Binary Segmentation Algorithm 1 searches for a single change point in this sequence that best explains a significant increase in Conventional Commits. After finding the change point, we calculate the date using the index of the commit at the change point and take a closer look at the sequence after the change point.

3. **Validation criteria:**
   - **CC Proportion after the change point:** The proportion of Conventional Commits after the change point must exceed a threshold of 50%. In contrast to the stricter 80% threshold for whole-repository classification, this 50% threshold reflects a more realistic pattern for early-adopter adoption. When adopting CC practices, By setting a 50% threshold, we capture projects in which CC practices have become the norm, without requiring immediate full compliance. This approach allows us to identify real changes in commit behavior and to account for
   - **Minimum number of commits:** We require at least 200 commits after the change point to ensure a substantial amount of post-change data for meaningful statistical analysis. This criterion confirms that the change in commit practices we observe persists over time, reducing the impact of anomalies or short-term fluctuations. We acknowledge that requiring a minimum of 200 commits may exclude smaller projects, but this criterion is essential to ensure reliability and validity in our analysis. In time series analysis and change point detection, a sufficient number of data points after the change point is critical for accurate detection and confirmation of significant changes [28]. Smaller sample sizes may not provide enough evidence to distinguish between random fluctuations and true change points.

Algorithm 1 Binary Segmentation (simplified with frequency-based cost function)

Input: Commit sequence of 0s and 1s, cost function based on frequency of 1s, stopping criterion

Initialize: Set of breakpoints $L \leftarrow \{\}$, start with the entire sequence as a single segment

repeat

1. Identify potential change points: For each segment, calculate the best point for splitting based on a cost function measuring the frequency of 1s.

2. Select the best change point: Find the point that maximizes the difference in the frequency of 1s between the two resulting segments and store it as a new change point.

3. Recursive splitting: Split the segment at the identified change point and repeat the analysis on each new segment.

until Stopping criterion is met

Output: Set L of identified change points

If these criteria are met, we record the date of the commit at the change point as the CC adoption date for the repository. Therefore, if neither the criteria for the adoption date nor the 80% threshold apply, the CC adoption date is not set and we classify the repository as non-conventional.

To visualize the detection of the change point, we create a heatmap that shows the commit history of each repository. Figure 5 shows the heatmap for the Repository AutoGPT[4]. In this heatmap, each cell corresponds to a commit, with Conventional Commits shown in dark gray and non-conventional commits in white. The chronological order of commits is shown along the x-axis.

The vertical red solid line in the figure shows the calculated change point by the Binary Segmentation Algorithm, which corresponds to the CC adoption date in the repository. This point represents a significant increase in the use of Conventional Commits. As can be seen in the figure, the proportion of Conventional Commits increases significantly after the change point, illustrating the effectiveness of the method we use to detect the CC adoption date. The heatmap provides a clear visual representation of the repository's transition to the adoption of Conventional Commits, supporting our analysis and corroborating the results obtained using statistical methods.

Binary segmentation provides a balance between computational efficiency and analytical precision. With a time complexity of $O(n \log n)$, the algorithm is suitable for large datasets with extensive commit histories. It detects change points based on statistical significance rather than arbitrary thresholds, allowing for more accurate identification of adoption dates. The method also accounts for fluctuations in commit frequency over

[4] https://github.com/Significant-Gravitas/AutoGPT.git

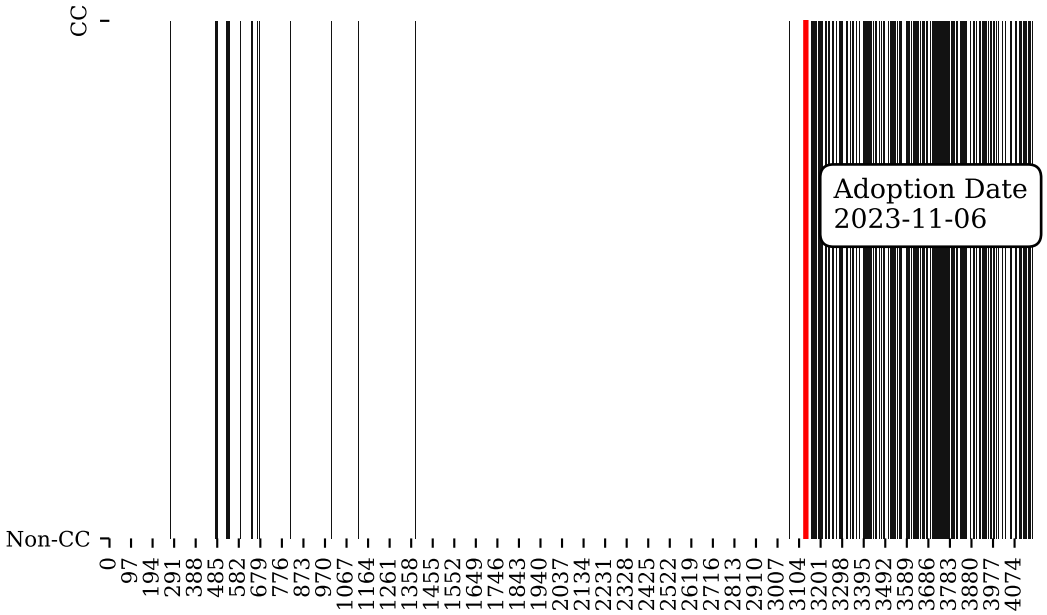

Figure 5: Change Point Detection in the Adoption of Conventional Commits for Repository `AutoGPT`

time, enabling a more adaptable and precise analysis. By focusing on repositories with an established CC adoption date, we ensure that our analyses are based on deliberate and consistent use of the CC specification, increasing the reliability of our research.

Initially, we investigated other methods for determining CC adoption. We considered a threshold-based approach that determines the point at which a repository reaches a certain cumulative rate of Conventional Commits. However, this method requires multiple iterations over all commits, resulting in a time complexity of $O(n^2)$, which was impractical for repositories with large commit histories. We also tried chunk analysis, where we divided the commit history into fixed-size chunks and computed the fraction of Conventional Commits within each chunk. While this method reduced computational complexity, it resulted in arbitrary segmentation and could miss nuanced adoption patterns, especially in repositories with uneven commit activity.

3.3.5 Metadata Summary and Preparation for Analysis

At the end of the data enrichment process, we store a comprehensive summary of the metadata for each repository to facilitate comparisons and further investigation. Figure 6 illustrates the data contained in the summary using an example repository. The summary contains three main categories of information for each repository:

1. **Basic Repository Metadata:** This category captures essential information that we extract in the first section 3.2 using the Github API for each repository, including the unique repository ID, name, owner, primary programming language, creation date, and repository size. These metadata fields provide context for each repository and allow sorting and filtering by basic characteristics.

2. **Commit statistics:** This section captures the statistics we collect during the commit message parsing in section 3.2 about the commit history in each repository. It contains the total number of commits as well as a breakdown of conventional and unconventional commits. In addition, the distribution of standard Conventional Commit types and user-defined types used in the repository are specified, providing information on the diversity and frequency of the commit categories within the project.

3. **Analysis results:** Finally, this section summarizes the most important results of the analysis of the CC implementation from chapter 3.3. Where applicable, the date of CC adoption is provided, reflecting the point at which the repository began to follow the Conventional Commits Specification. We also record the Conventional Commit Proportion, which we calculate as the percentage of standard Conventional Commits relative to the total commit history. We use this rate to determine whether the repository can be categorized as "consistently conventional", which indicates strong adherence to CC standards. In addition, according to the hint-based analysis in section 3.3.2, the flag is set depending on whether the CC Specification is mentioned in any form in the project or on the homepage.

Repository Name: `jina`
Owner Type: `Organization`
Language: `Python`
Size: `1,421,794 bytes`
Created At: `2020-02-13`

Commit Statistics

Total Commits: 6,011
- **CC Standard Commits:** 5,909
- **CC Custom Commits:** 12
- **Unconventional Commits:** 90

Standard CC Types:
fix: 1,563, test: 381, feat: 1,022, refactor: 660, chore: 976, ci: 316, docs: 863, style: 35, build: 45, perf: 35, revert: 13

Custom Types:
cd: 4, tests: 4, cicd: 1, stlye: 1, doc: 1, refacor: 1

Analysis Results

Conventional Commit Proportion: 98%
Consistently Conventional: True
CC Adoption Date: `2020-02-13`
Mention of CC: True

Figure 6: Overview of Summary of Repository `jina`[5]

3.4 Analysis of the Adoption and Consistency of CC

To address **RQ1** - `To what extent and with what consistency are Conventional Commits (CC) adopted?` - we analyze the repositories in our dataset based on the metadata and additional commit information we are obtaining so far in our study. Building on the identification of CC adoption described in Section 3.2, we evaluate both the prevalence of CC adoption in all repositories and the consistency of its implementation in each repository.

We consider a repository as CC repository if it has a valid adoption date. Repositories with an adoption date can be divided into two categories: those that have consistently used CC from the beginning and those that have switched to CC at a later point in their development history. This distinction allows us to understand not only the overall adoption of CC, but also the adoption patterns between different projects.

To understand the extent of CC adoption, we calculate the proportion of repositories that have adopted CC out of the total number of repositories in our dataset. This metric provides a comprehensive overview of the prevalence of CC practices in the selected open source projects.

We also examine the consistency of CC implementation in CC repositories by analyzing the Conventional Commit Proportion (CCP). This involves calculating the extent to which standardized commit practices are followed, enabling a deeper understanding of the consistency of CC implementation in the commit histories of the projects.

In addition, we compare repositories that mention the use of CCS at some point in the project with the actual implementation of CC, which is indicated by the presence of a valid adoption date. This comparison helps us to identify discrepancies between repositories that mention the use of CC and those that implement it in practice. Such discrepancies provide insight into the correspondence between documented intentions and actual transfer practices.

To gain a deeper understanding of the factors that influence CC adoption, we analyze repository characteristics such as primary programming language, repository size, owner type and creation date. We investigate whether certain programming languages have higher rates of CC adoption, whether larger projects are more likely to adopt and consistently use CC, and whether the age of a project correlates with the likelihood of adopting CC practices. By examining the correlations between these characteristics and metrics of CC adoption, we aim to uncover patterns that may explain differences in CC adoption in the open source community.

Furthermore, we investigate the distribution of standard CC types and user-defined types within the repositories. By analyzing the frequency and diversity of commit types used, we try to understand how developers use the CC specification and whether they extend it with custom commit types to meet the specific needs of their project. This analysis explores the flexibility of the CCS and its adaptability in different projects.

3.5 Analysis of Commit Behaviors Before and After CC Adoption

To address **RQ2**— `How does the adoption of Conventional Commits influence commit behaviors before and after its adoption?`— we focus on repositories that have a valid CC adoption date. For these repositories, we divide the commit history into two periods: the pre-adoption period, comprising all commits made before the CC adoption date, and the post-adoption period, including all commits from the adoption date onwards. This temporal segmentation allows us to compare commit behaviors and assess any changes associated with the adoption of CC.

[5]https://github.com/jina-ai/jina

We analyze various commit-level metrics in both periods, such as the average number of insertions, deletions, and files changed per commit. Additionally, we examine the length of the commit messages to evaluate changes in the verbosity and clarity of commit descriptions. By comparing these metrics between the pre-adoption and post-adoption periods, we aim to identify significant changes in commit practices that may be attributed to the adoption of CC.

Our analysis focuses on assessing whether the adoption of CC leads to more granular and focused commits, characterized by smaller changes and more descriptive commit messages. We hypothesize that adhering to the CC specification encourages developers to commit changes more frequently, with each commit representing a single logical change, and to write commit messages following the structured format prescribed by CCS, thereby improving the overall quality and readability of the project's commit history.

By analyzing the commit behaviors before and after CC adoption, we aim to provide empirical evidence on the influence of CC practices on developer behavior within the repositories studied. This approach allows us to evaluate the practical impact of adopting the CC specification on software development practices in open-source projects.

4 Results

This section presents the findings of our study, organized according to the research questions outlined earlier. The results provide insights into the adoption rates of Conventional Commits (CC), the consistency of their application across different project dimensions, and the impact of CC adoption on commit behaviors.

4.1 Dataset Overview

Our dataset comprised 2,495 repositories, which, after preprocessing, contained a total of 19,403,873 commits. These repositories span ten programming languages, with Python, JavaScript, and Go being the most represented. On average, each repository had 351 contributors, reflecting substantial collaboration. The repositories averaged 21,987 stars, indicating high community interest, and had an average age of 8.36 years. Table 4 summarizes the key repository metrics, including the average and median values for stars, number of commits, number of contributors, and age of the repository.

We also analyzed the repositories based on their ownership type. As showed in Table 5 of the total repositories, 1,692 are owned by organizations, and 803 are owned by individual users. On average, repositories owned by organizations have more contributors compared to user-owned repositories. Organizational repositories also have a slightly higher average star count compared to user-owned repositories.

	Metric per Repository	Average	Median
0	Stars	21987.3	16309
1	Commits Count	7777.1	1939
2	Contributors	350.59	149
3	Size (in kB)	198476	31273
4	Age (in years)	8.36	8.46

Table 4: Metric Statistics

Owner	No. of Projects	Avg. Contributors	Avg. Stars	Avg. Age in Years
Organization	1692	447	22342	8.6
User	803	147	21241	7.9

Table 5: Summary of Repository Characteristics by Owner Type

4.2 RQ1: Extent and Consistency of CC Adoption

To address RQ1—"To what extent and with what consistency are Conventional Commits (CC) adopted?"—we analyzed the repositories for CC adoption and consistency.

A repository is classified as a CC-repository if it has a valid adoption date. Based on this criterion, 334 repositories (13.39%) were found to have adopted the Conventional Commit Specification (CCS). Among these, 74 repositories were consistently conventional, utilizing CC from their inception, while 260 repositories switched to CC practices at a later stage of development. The majority (2,161 repositories) had not adopted CC practices. Overall, the total adoption of CC practices in Figure 7 over time shows that CC adoption has steadily increased from around 2% in 2017 to around 13% in 2024.

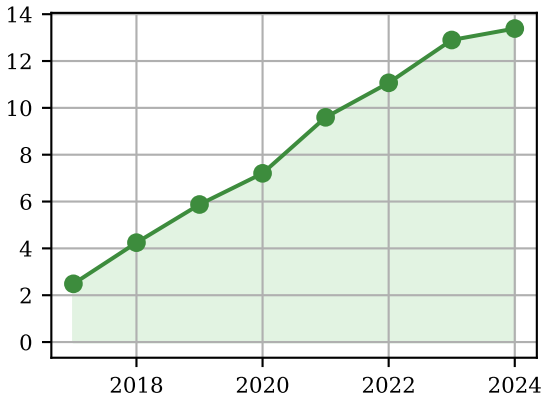

Figure 7: Adoption Rate over Time

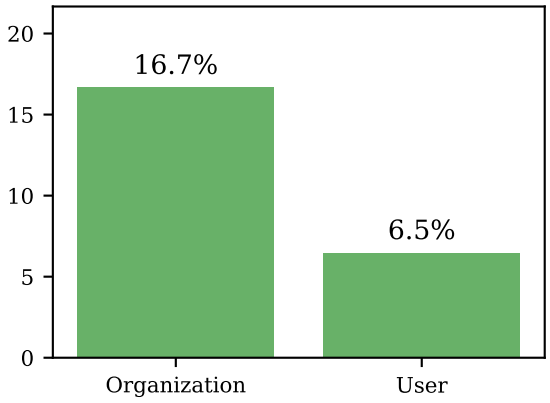

Figure 8: Adoption Rate by Projectowner

We assessed the consistency of CC implementation by analyzing the Conventional Commit Proportion (CCP) within CC repositories. As can be seen in Figure 9 consis-

tently conventional repositories exhibited an average CCP of 89,21%, indicating strong adherence to CC standards from the outset. For repositories that adopted CC later on, the average CCP is about 52%. However, if we only look at the period after adoption, these repositories, with a later CC adoption, also have a CCP of 78.21%.

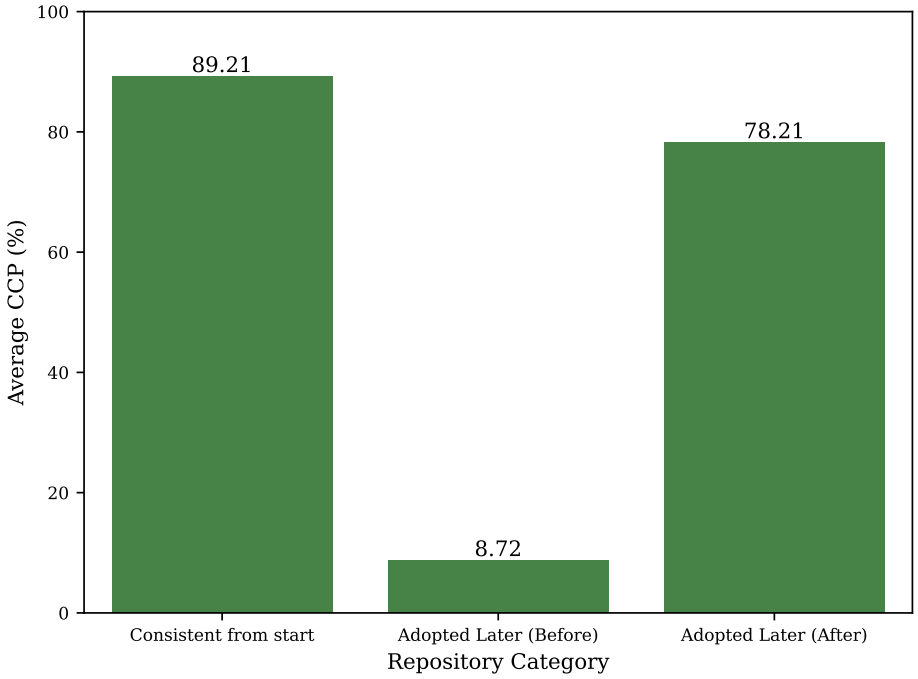

Figure 9: Average (CCP) Before and After Adoption

Figure 10 compares the average percentages of commit messages with standard and custom commit types in three categories. The first category shows the proportion of commit messages with standard and custom commit types over the total dataset, the second category only covers CCS-repositories and the third shows the commits of the remaining repositories that are not conventional.

Firstly, the results show that the methodology for identifying CCS-compliant repositories is highly effective, with non-CCS repositories containing less than 2% of conventional commits. Second, the results indicate that custom-type commits are predominantly used in non-CCS repositories, further supporting the reliability and robustness of the classification approach. By contrast, CCS repositories have a lower frequency of custom-type commits than the dataset average, reinforcing the validity of the classification as a reliable method for identifying CCS compliance.

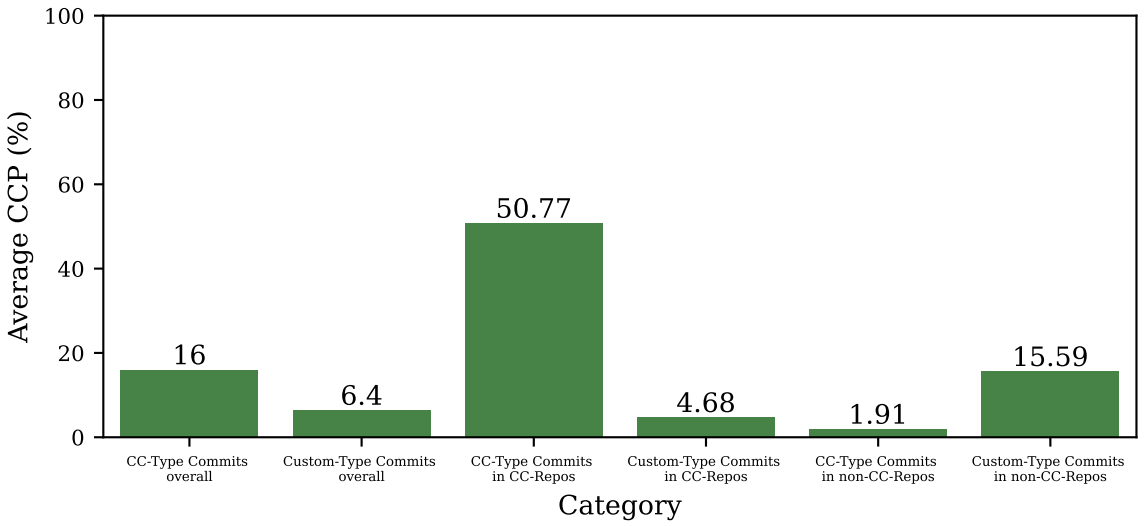

Figure 10: Average Percentages of CCS-Compliant and Custom-Type Commits

4.2.1 CC Indication Flag versus Actual Implementation

To investigate CC indication versus the actual adoption, we classified the repositories according to the existence of a CC indicator, which denotes the mention of CC in the documentation or configuration of a repository, and actual CC adoption. The comparison revealed noteworthy results (Table 6). About 67% of repositories with a CC indication had a valid CC adoption date, confirming their adherence to CC practices. Conversely, about 10% of the repositories without a CC claim had nevertheless adopted CC practices.

	CC Adopted	**CC Not Adopted**	**Total**
CC Indication	89	43	132
No CC Indication	235	2076	2311
Total	324	2119	2443

Table 6: Comparison of CC-Indication and CC-Adoption

4.2.2 CC Adoption by Repository Characteristics

We examined how primary programming language, owner type, project age, and project size correlate with the adoption of CCS.

- **Primary Programming Language**

  Figure 11 illustrates the CC adoption rates across different programming languages. TypeScript projects demonstrated the highest adoption rate at 48.4%, followed by Rust at 20.8% and JavaScript at 19.28%. In contrast, languages such as C, C#, and PHP had lower adoption rates, ranging from 1.61% to 5.20%.

- **Owner Type**

  As shown in Figure 8, projects owned by organizations exhibited a significantly higher CC adoption rate compared to those owned by individual users. Specifically, 16.7% of organizational projects adopted CC, whereas only 6.5% of user-owned projects did so.

- **Project Age**

  Figure 12 shows the adoption rate of CC in relation to project age. The acceptance of CC decreases with increasing project age, with newer projects showing significantly higher adoption rates.

- **Project Size**

  Figure 13 presents the adoption rate of Conventional Commits (CC) by project size category, measured in percentiles. The project size categories are divided into "Small" (0-25th percentile), "Medium" (25-50th percentile), "Large" (50-75th percentile) and "Very Large" (75-100th percentile) based on the size in kB. Adoption of CC practices increases from smaller to larger projects, with "small" projects having an adoption rate of 8.7%, followed by "medium" projects at 13.1%. The highest adoption rate is observed in "large" projects at 17.7%, indicating that CC practices are particularly widespread in this category. For 'very large' projects, the adoption rate decreases slightly to 14.1%, although it remains relatively high compared to smaller project categories.

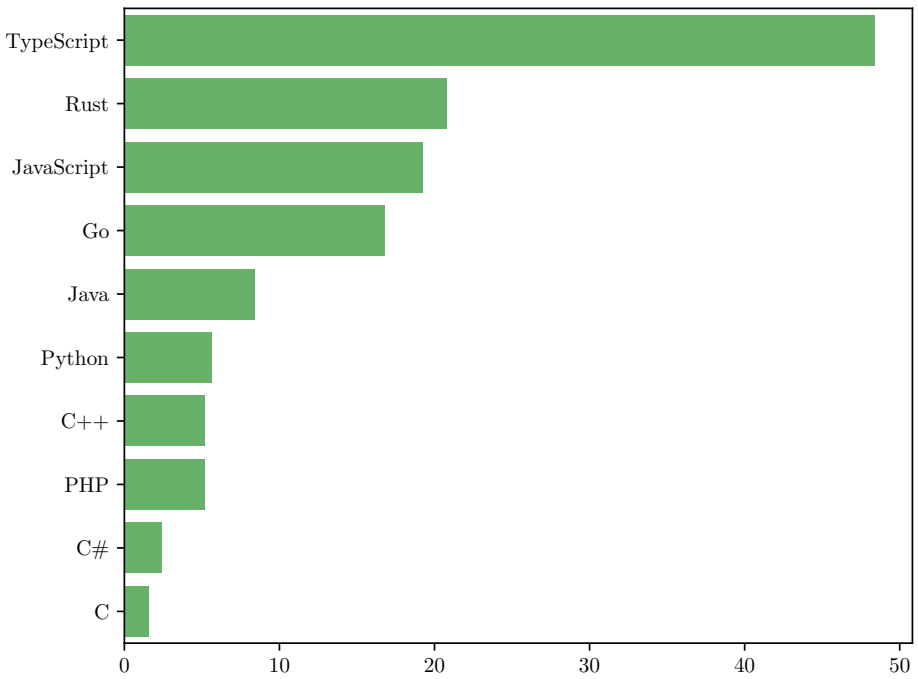

Figure 11: Adoption Rate of CC by Language

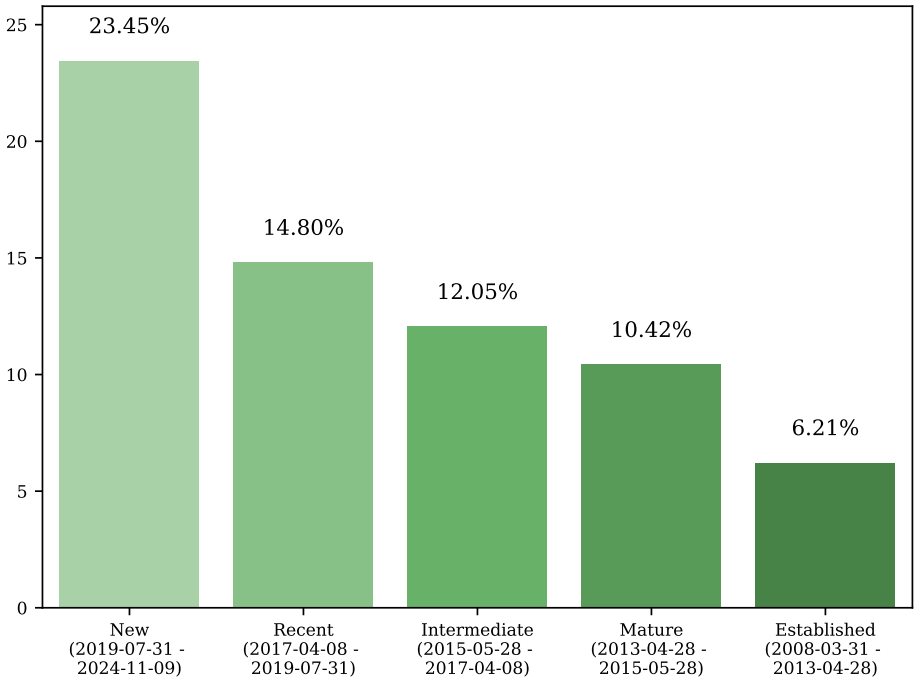

Figure 12: Adoption Rate by Project Age

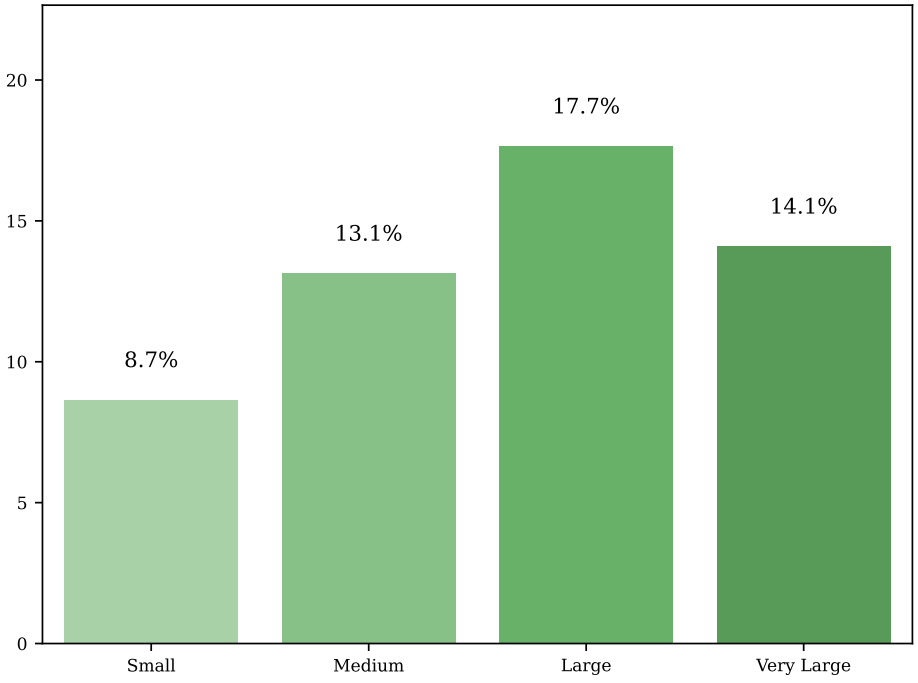

Figure 13: CC Adoption Rate by Project Size

4.2.3 Distribution of Standard and Custom CC Types

Figures 14 and 15 present visualizations of the distribution and impact of different Conventional Commit (CC) types on the analyzed repositories.

- **Standard CC Types**

  Figure 14 illustrates the distribution of the standard CC types according to the total number of commits. It is particularly noticeable that the "fix" type is used most frequently with over 400,000 occurrences, followed by "docs" and "feat" with over 200,000 commits each. Other types, such as "style", "test" and "perf", appear much less frequently, with values below 20,000, indicating a lower frequency of style, testing and performance-related changes.

- **Impact on Codebase Metrics**

  Figure 15 shows the impact of each CC type on the codebase metrics, with a focus on insertions and deletions. Here, the "feat" type has the largest impact on code additions with over 1.2 million insertions, closely followed by "docs", which also has a significant number of insertions. Although the "fix" type is widespread, it has a more balanced distribution of insertions and deletions. In contrast, types such as "style" and "test" show only minimal effects on the changes to the codebase.

- **Custom Commit Types**

  Figure 16 shows the distribution of the 20 most frequently used custom commit types and illustrates how contributors extend the CC specification to adapt it to the specific requirements of their projects. The most frequently used custom types are "staging", "arm" and "net", which occur more than 60,000 times each.

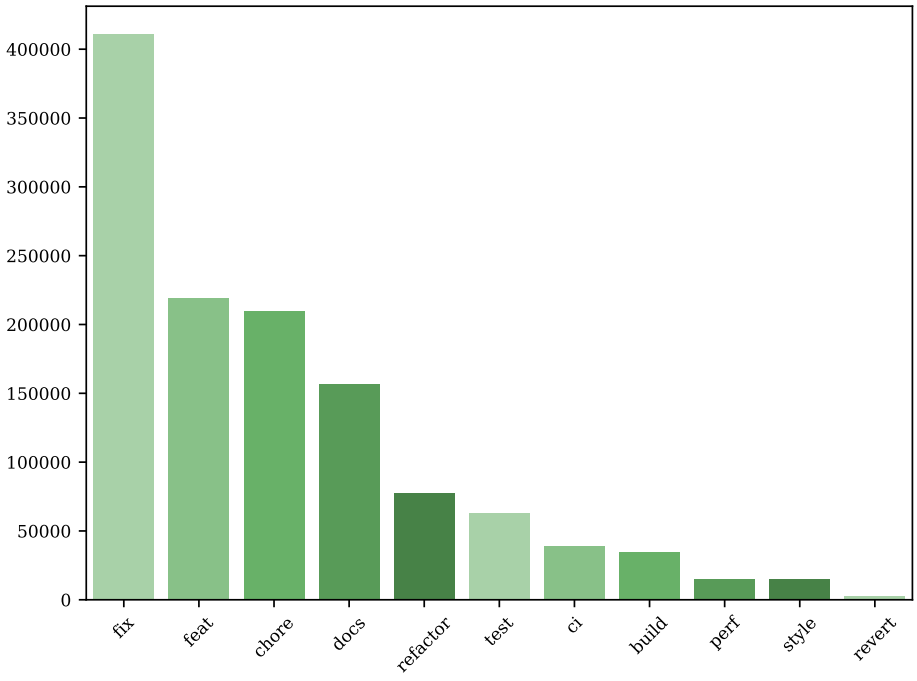

Figure 14: CC-Type-Distribution

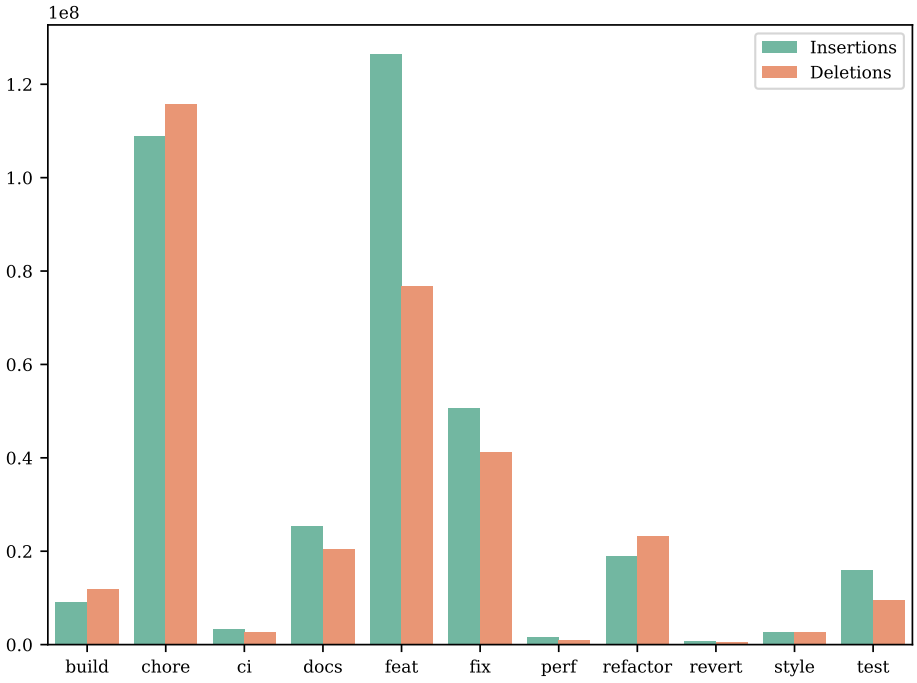

Figure 15: CC Types Impact on Codebase Metrics

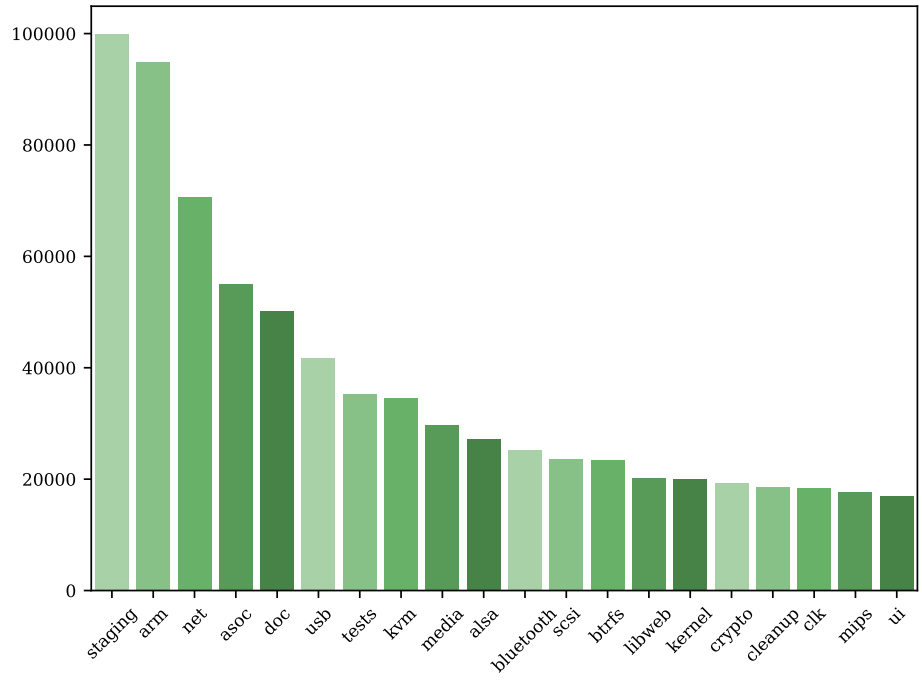

Figure 16: Custom-Type-Distribution

4.3 RQ2 Commit Behaviour Before and After CC Adoption

To address RQ2—"How does the adoption of Conventional Commits influence commit behaviors before and after its adoption?"—we analyzed commit-level metrics for repositories that adopted CC during their development history.

Table 7 summarizes the commit metrics before and after the introduction of conventional commits (CC) in the investigated repositories with an adoption date. The volume of commits before the adoption date of CC is 734,875 and after the adoption date 980,408. The average number of changed files per commit increased by about 20.4% from before adoption to after adoption. In contrast, the average number of insertions per commit fell by around 11.3%, while the average number of deletions per commit also fell by around 6.9%. The frequency of commits has also decreased significantly, from an average of 732 to 491 commits per day, while the average length of commit messages has increased by approximately 17%. As can be seen in Figure 9 the increase in the conventional commitment proportion from before the introduction to after the introduction is quite significant.

Additionally, we examined the correlations between CC adoption and various repository metrics. The correlation coefficients are as follows:

- **Size**: 0.01 (negligible)
- **Total commits**: -0.03 (negligible)

- **Repository age**: -0.16 (slight)

These values indicate that there is no significant correlation between CC adoption and repository size or total number of commits. However, the negative correlation with repository age shows that newer repositories are more likely to adopt CC practices.

Metric	Before Adoption	After Adoption
Total Commits	734,875	980,408
Files Changed (Avg)	6.26	7.54
Insertions (Avg)	304.12	269.76
Deletions (Avg)	221.70	206.48
Commitmessage Characters (Avg)	40.80	47.87
Commits per Day (Avg)	732.19	491.23

Table 7: Commit Metrics Before and After CC Adoption

5 Discussion

This study provides valuable insights into the factors influencing the adoption and use of the Conventional Commit Standard (CCS) in various projects and contexts. In the following, the main findings are discussed and put into the context of practical implications.

5.1 Interpretation of the Results

5.1.1 Extent and Consistency of CC adoption

Our analysis shows that CC adoption is present in about 13.39% of the repositories examined. Of these, a minority (22%) were consistently conventional from the beginning, while the majority adopted CC at a later stage. This moderate degree of adoption indicates that CC practices are recognized but not yet implemented across the board. However, a significant increase in the adoption rate of CCS over the last few years (from 2% in 2017 to around 13% in 2024) shows that CCS is becoming increasingly popular. This suggests that while the adoption of CC is not yet widespread, it has already gained considerable traction in certain communities. The repositories that have adopted CCS have a very high rate of conventional commits (78%) after its introduction. Repositories that have relied on CC from the outset consist almost entirely of CC-compliant commit messages (approx. 89%). The increase in acceptance over time shows that CCS is perceived as a valuable standard for improving code quality and transparency in the community.

Particularly noticeable is the high adoption rate in TypeScript projects at 48.40%, followed by Rust at 20.80% and JavaScript at 19.28%. This could indicate a greater

awareness and support for CC in these language communities, possibly influenced by modern development paradigms and the use of tools that promote CC compliance. Rust and Typescript in particular, as noted in the dataset collection (Section 3.2), are among the languages with a younger average age, which in turn reinforces the correlation assumption between project age and CCS adoption. The lower adoption rates for traditional languages such as C, C# and PHP, which are often used in established and sometimes less agile environments, suggest that the conventions of the respective developer communities and the nature of the projects play a role in the decision to use CCS. It is important to recognize that CC adoption depends not only on technical factors, but also on cultural and social aspects within developer communities.

Possibly related to this, organization-owned projects show a significantly higher adoption rate with an adoption rate of 16.7% compared to user-owned projects with an adoption rate of 6.5%. This suggests that organizations are usually dependent on good project management due to the size of their number of contributors and their structure and are more interested in clear structures and easy traceability of changes. User projects, which are often run by individual developers or small teams, may feel less need for standards and use simpler practices instead.

The comparison between the mention of CCS practices in the repository and the actual CC adoption shows a high correlation of about 66.43%. This confirms that the mention of CC in the documentation can be a robust indicator of actual adoption. However, the finding shows that about 33.57% of the repositories that mention CC have not actually adopted it. This could indicate a discrepancy between documented intentions and actual practices, possibly caused by changes in project requirements or incomplete documentation updates. The fact that 10.24% of repositories without CC indication have nevertheless adopted CC practices suggests that the adoption of CC is not always communicated through the documentation or not through public channels but, for example, a private contributor platform. The integration of CC practices through external tools and automation may also be a reason why their use is not explicitly mentioned in the project description.

Another interesting finding is the significant variation in the adoption rate of CCS with the age of the project: new projects that have started since 2019 show the highest adoption rate, while older projects that started before 2013 show significantly lower rates. These results suggest that CCS is being integrated into newer projects from the outset, as the willingness to adapt new standards is often greater in younger projects. Older projects with established workflows and commit practices, on the other hand, may find it difficult to abandon existing conventions and introduce CCS. This could indicate that there are hurdles and difficulties in adapting existing processes to CCS. The size of a repository, on the other hand, does not correlate with the introduction of CCS.

The commit types also appear at first sight to be used as intended by the CCS, so the largest code changes take place when a feature (feat) is developed, while small changes

are linked to changes in style or text functions. The most commonly used fix type has a relatively balanced number of insertions and deletions, which is quite standard for a bugfix, as code is usually replaced by other code with the same function. For the chore type, on the other hand, the CCS does not envisage any changes to the source or test files, but rather a change to the files or project configuration. This is also reflected in the almost identical number of insertions and deletions. The use of chore, docs and refactor appears regularly, which indicates frequent maintenance work, documentation and code optimizations. Overall, the distribution of commit types and their metrics give a good indication of how a repository works.

In addition to the standardized CC types, some projects have introduced individual, user-defined CC types to better meet specific requirements and activities. The variety of these user-defined CC types reflects the different needs and specializations of the projects.

The user-defined CC types, such as `arm`, `kvm`, `alsa`, `bluetooth` or `crypto`, designate specific components or modules that are processed in the projects. These types allow developers to categorize commits more precisely, which is particularly useful for complex or modular projects. By introducing such specific types, teams can track and analyze changes more clearly, which improves the maintainability and understanding of the project. However, the introduction of such user-defined types also poses challenges, as it makes standardisation more difficult, which runs counter to the aim of CCS, and an excessive number of user-defined types can lead to confusion and make projects less comparable.

5.1.2 Influence on Commit Behavior

The analysis of commit behavior before and after the introduction of CC showed that the average number of changed files per commit increased, while the average number of inserts and deletions per commit decreased. This indicates that the commits became more focused and smaller code changes were spread across slightly more files. These changes in commit behavior reflect a shift towards more detailed and organized commits, which can improve code review processes and enable better collaboration. While the increase in changed files may seem counterintuitive, it may indicate that developers are grouping related changes that affect multiple files into single, coherent commits with standardized messages.

The commit frequency dropped significantly following the adoption of Conventional Commits (CC), decreasing from 732.19 commits per day before adoption to 491.23 commits per day afterward. This reduction suggests that with the implementation of CC guidelines, developers may be making fewer but possibly more structured and meaningful commits. The lower frequency could indicate a shift towards a more targeted organisation of change, consistent with the CC standard that promotes clarity and specificity in commit messages. However, it is also possible that older projects in particular are in a different phase e.g. that only consists of bugfixing and therefore no longer work on the

code as frequently. This is also a topic of future research. This change might reflect a more deliberate approach to documenting code changes, focusing on quality over quantity in commit activity. Based on the proportion of conventional commits compared to non-conventional commits, it can be said that the repositories that have introduced CCS continue to do so at a high level and have only a few outliers.

5.2 Threats to Validity

5.2.1 Internal Validity

A potential threat to internal validity is the accuracy of CC adoption detection. Our method relies on analyzing commit messages and identifying adoption data that may not capture all instances of CC usage, especially when commit messages deviate slightly from the specification. Misclassification could influence the reported adoption rates. Also the pattern recognition matching the CCS was not extended to include variations such as misspellings in the searched keywords. The given example in table 3 includes "stlye", "tests", "doc", "refacor" custom types although they are clearly intended as standard commit types. Furthermore, thresholds chosen (50 percent after change point detection and 80 percent for consistent CC) are based on best practice in software development research, but may still miss relevant data. Another important point that also provides material for further research is that we deliberately focus only on repositories that show an increase in conventional commits, and exclude repositories that have introduced and then removed CCS. It can therefore not be assumed that we have discovered all repositories that have implemented CC at one point in time if that project has then decided to remove CCS.

In addition, the analysis of indicators within CC repositories is based on a possibly incomplete selection of keywords and search locations, which may have led us to overlook indicators for CCS in repositories.

Furthermore, although we have taken into account recent research on bot filtering, it is possible that not all automated commits were correctly identified and excluded. This could affect the results, as bot commits may not comply with CC standards.

5.2.2 External Validity

The study focuses on popular repositories on GitHub with a high star count, which may not be representative of all open source projects. The results can generalise information for bigger, more active or well-maintained open source project but smaller or in general less active repositories may show different adoption patterns. Our dataset is limited to public projects on GitHub and our observations might not be generalizable to closed-source projects. Other version control platforms such as GitLab or Bitbucket were not considered. Therefore, the results cannot be easily transferred to projects on these platforms or to other version control systems. However, many professional developers

participate in projects on GitHub, and our dataset covers with our selected programming languages over 80% of projects in the open source landscape [2].

5.3 Implications for Software Development Practice

The introduction of CCS leds to smaller commits, which points to more focused and better organized commits that can improve the code review process and facilitate collaboration. New projects especially benefit from the integration of CCS from the start, while older projects may face challenges in adapting existing processes. Therefore, organizations should take specific measures to facilitate the adoption of CCS, for example by providing comprehensive documentation and integrating CCS reviews into continuous integration pipelines.

Tool developers play a crucial role in promoting CCS, as they provide tools that support the creation of Conventional Commit messages and check their compliance with CCS. In addition, the existence of custom commit types in some projects shows that adaptability and flexibility are important to meet the specific requirements of different projects. This underlines the need to further develop CCS standards and make them customizable to ensure broad acceptance and effective use in diverse development environments.

Overall, the increasing adoption of conventional commits contributes significantly to the efficiency and quality of software development processes. It not only promotes the automation and organization of development processes, but also improves communication and collaboration within development teams. In order to identify further hurdles and effects of the application of CCS, research should continue to investigate how CCS can be optimized and disseminated in different contexts. For example, by analyzing projects that may have introduced and discontinued CCS or the effects of using CCS on project success and in different development phases and by different developer groups.

6 Conclusion

This study is one of the first in its field to provide a comprehensive insight into the factors driving the adoption and use of the Conventional Commit Specification across different projects. The analysis shows that CCS is currently used in about 13.39% of the repositories analyzed, with adoption increasing significantly in recent years. Especially in modern languages such as TypeScript, Rust and JavaScript, a high adoption rate can be observed. This discrepancy may reflect the different requirements and workflows in different programming languages. In addition, organization-owned projects had higher CCS adoption rates than user-owned projects, likely due to structured workflows that facilitate standardized handoff procedures.

A crucial part of this work was to develop the methodology to determine whether

a repository can be classified as conventional and, if the project did not adopt CCS from the very start, to find out the corresponding date of adoption of CCS. With this methodology, the gap in previous research discussed in the introduction was closed. The change point detection of the application of CCS in repositories can therefore be used for further work or serve as a basis for further development.

A key finding of this study is that the introduction of CCS changes the commit behavior. In particular, we observed a decrease in commit frequency, indicating a possible shift towards more structured, meaningful commits in line with CCS guidelines. The average number of files changed per commit increased after implementation, which may indicate a trend towards grouping related changes into fewer, more comprehensive commits. In addition, the lower number of inserts and deletes suggests that the adoption of CCS encourages more concise changes, contributing to a clearer commit history.

Despite the positive findings, there are also challenges and limitations to the introduction of CCS. The discrepancy between the mention of CCS in the documentation and the actual adoption points to possible implementation barriers, such as incomplete documentation updates or the non-public communication of CCS usage. In addition, the introduction of user-defined commit types shows that flexibility and adaptability are necessary to meet the specific requirements of different projects, but this can make standardization more difficult.

This thesis presented many attributes of CCS descriptively, offering a snapshot of its current state of application. In doing so, it lays a foundation by providing a broad overview of various aspects of CCS, paving the way for future research to focus more deeply on specific facets of its adoption and impact, potentially leading to more refined explanatory approaches. Future studies could dive deeper and examine more complex metrics, such as the influence of CCS on project success, or investigate strategies to reduce barriers to CCS adoption, particularly for older, smaller, or user-owned projects. Additionally, exploring the long-term benefits of CCS for software quality and maintainability could offer valuable insights for the development community.

In summary, this research supports the effectiveness of CCS in promoting well-structured commit messages and improving collaborative development. By standardizing commit messages, CCS not only improves project documentation, but also lays the foundation for automation tools that rely on structured commit data, making it a valuable asset for the open source community.

References

[1] Githut 2.0. https://madnight.github.io/githut/.

[2] Pypl popularity of programming language index. https://pypl.github.io/PYPL.html. Accessed: 2024-11-09.

[3] semantic-release. [Online]. Available: https://github.com/semantic-release/semantic-release.

[4] Conventional commit specification, 2024. [Online]. Available: https://www.conventionalcommits.org/en/v1.0.0/.

[5] 6sense. GitHub Market Share, 2024. [Online]. Available: https://6sense.com/tech/source-code-management/github-market-share.

[6] Angular. Commit message format, 2024. [Online]. Available: https://github.com/angular/angular/blob/main/CONTRIBUTING.md#commit-message-format.

[7] P. Behnamghader, R. Alfayez, K. Srisopha, and B. Boehm. Towards better understanding of software quality evolution through commit-impact analysis. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 251–262, 2017.

[8] Y. Boubekeur, G. Mussbacher, and S. McIntosh. Automatic assessment of students' software models using a simple heuristic and machine learning. In *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '20, New York, NY, USA, 2020. Association for Computing Machinery.

[9] Centre Borelli. Binary Segmentation (BinSeg) - Ruptures Documentation, 2024. Accessed: 2024-11-08.

[10] J. M. Champagne and D. L. Carver. Discovering relationships among software artifacts. In *2020 IEEE Aerospace Conference*, pages 1–11, 2020.

[11] R. Dyer, H. Nguyen, H. Rajan, and N. Tien. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. pages 422–431, 05 2013.

[12] Ember.js. Commit message guidelines, 2024. [Online]. Available: https://github.com/emberjs/ember.js/blob/main/CONTRIBUTING.md#commit-message-guidelines.

[13] GitPython. The gitpython library for interacting with git repositories, 2024. [Online]. Available: https://github.com/gitpython-developers/GitPython.

[14] S. E. Inc. Beyond Git: The Other Version Control Systems Developers Use. Website: https://stackoverflow.blog/2023/01/09/beyond-git-the-other-version-control-systems-developers-use/, 2023. Accessed: 2024-11-01.

[15] JSHint. Commit message guidelines, 2024. [Online]. Available: https://github.com/jshint/jshint/blob/main/CONTRIBUTING.md#commit-message-guidelines.

[16] J. Y. D. Lee and H. L. Chieu. Co-training for commit classification. In *Proceedings of the Seventh Workshop on Noisy User-generated Text (W-NUT 2021)*, pages 389–395. Association for Computational Linguistics, Nov. 2021.

[17] I. Ma and C. V. Lopes. Improving the quality of commit messages in students' projects. In *2023 IEEE/ACM 5th International Workshop on Software Engineering Education for the Next Generation (SEENG)*, page 1–4. IEEE, May 2023.

[18] H. Malik, H. Hemmati, and A. E. Hassan. Automatic detection of performance deviations in the load testing of large scale systems. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1012–1021, 2013.

[19] Mockus and Votta. Identifying reasons for software changes using historic databases. In *Proceedings 2000 International Conference on Software Maintenance*, pages 120–130, 2000.

[20] M. Mousavi, M. R. Lyu, and I. King. BIMAN: A bi-mode network for automatic detection of software bots. *arXiv preprint arXiv:1905.10620*, 2019.

[21] M. Pikkarainen, J. Haikara, O. Salo, P. Abrahamsson, and J. Still. The impact of agile practices on communication in software development. *Empirical Software Engineering*, 13:303–337, 06 2008.

[22] T. Preston-Werner. Semantic versioning. [Online]. Available: https://semver.org/.

[23] M. T. Rahman, R. Singh, and M. Y. Sultan. Automating patch set generation from code reviews using large language models. In *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering - Software Engineering for AI*, CAIN '24, page 273–274, New York, NY, USA, 2024. Association for Computing Machinery.

[24] S. Reis, R. Abreu, H. Erdogmus, and C. Păsăreanu. Secom: Towards a convention for security commit messages. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pages 764–765, 2022.

[25] E. B. Swanson. The dimensions of maintenance. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE '76, page 492–497, Washington, DC, USA, 1976. IEEE Computer Society Press.

[26] X. Tan and M. Zhou. How to communicate when submitting patches: An empirical study of the linux kernel. *Proc. ACM Hum.-Comput. Interact.*, 3(CSCW), Nov. 2019.

[27] Y. Tian, Y. Zhang, K.-J. Stol, L. Jiang, and H. Liu. What makes a good commit message? In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 2389–2401, New York, NY, USA, 2022. Association for Computing Machinery.

[28] C. Truong, L. Oudre, and N. Vayatis. Selective review of offline change point detection methods. *Signal Processing*, 167:107299, 2020.

[29] A. Zagalsky, J. Feliciano, M.-A. Storey, Y. Zhao, and W. Wang. The emergence of github as a collaborative platform for education. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, CSCW '15, page 1906–1917, New York, NY, USA, 2015. Association for Computing Machinery.

[30] J. Zhu, M. Zhou, and A. Mockus. Effectiveness of code contribution: from patch-based to pull-request-based tools. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 871–882, New York, NY, USA, 2016. Association for Computing Machinery.

[31] N. N. Zolkifli, A. Ngah, and A. Deraman. Version control system: A review. *Procedia Computer Science*, 135:408–415, 2018. The 3rd International Conference on Computer Science and Computational Intelligence (ICCSCI 2018) : Empowering Smart Technology in Digital Era for a Better Life.

[32] Y. Zou, B. Ban, Y. Xue, and Y. Xu. Ccgraph: a pdg-based code clone detector with approximate graph matching. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 931–942, New York, NY, USA, 2021. Association for Computing Machinery.

7 Data Availability

The data used for this study and the associated code are publicly available at the following GitHub repository: `https://github.com/annafreidl/ConventionalCommitsStudy`

Appendix

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den November 11, 2024 .....