A First Look at Conventional Commits Classification

Qunhong Zeng Yuxia Zhang† Zhiqing Qiu Hui Liu
School of Computer Science & Technology,
Beijing Institute of Technology,
Beijing, China
{qunhongzeng, yuxiazh, zhiqingqiu, liuhui08}@bit.edu.cn

***Abstract*—Modern distributed software development relies on commits to control system versions. Commit classification plays a vital role in both industry and academia. The widely-used commit classification framework was proposed in 1976 by Swanson and includes three base classes: perfective, corrective, and adaptive. With the increasing complexity of software development, the industry has shifted towards a more fine-grained commit category, i.e., adopting Conventional Commits Specification (CCS) for delicacy management. The new commit framework requires developers to classify commits into ten distinct categories, such as "feat", "fix", and "docs". However, existing studies mainly focus on the three-category classification, leaving the definition and application of the fine-grained commit categories as knowledge gaps. This paper reports a preliminary study on this mechanism from its application status and problems. We also explore ways to address these identified problems. We find that a growing number of projects on GitHub are adopting CCS. By qualitatively analyzing 194 issues from GitHub and 100 questions from Stack Overflow about the CCS application, we categorized four main challenges developers encountered when using CCS. The most common one is CCS-type confusion. To address these challenges, we propose a clear definition of CCS types based on existing variants. Further, we designed an approach to automatically classify commits into CCS types, and the evaluation results demonstrate a promising performance. Our work facilitates a deeper comprehension of the present fine-grained commit categorization and holds the potential to alleviate application challenges significantly.**

***Index Terms*—Commit Classification, Conventional Commits, Large Language Model**

I. Introduction

Modern distributed software development relies on version control systems (VCS), such as Git, to facilitate change tracking and collaboration [1]. Developers conduct diverse activities, such as implementing new features or fixing bugs, enacting corresponding code changes, and *commit* them to the VCS, thereby creating a new version. When committing code changes to the VCS, developers must compose a *commit message* to describe the modifications.

In 1976, Swanson [2] categorized commit activities into three fundamental types: perfective, adaptive, and corrective, to support cost-effective management and evolution of software projects [2, 3]. While Swanson's categories provide a dimensional understanding of development maintenance activities, they are not widely used in industrial practices today. Modern software development complexities have led to different commit conventions across different projects [4, 5, 6]. For instance, ember.js [6] mandates a prefix <*type*> filed in their commit messages, asking developers to classify commits into six categories: feature, bug fix, cleanup, documentation, security, and other. The machine-parsable <*type*> prefix in these conventions facilitates automated processes like release note generation [7] and semantic version bump [8]. However, adhering to varying conventions across multiple projects can be challenging for developers. To this end, the Conventional Commits Specification (CCS) [9] offers a uniform commit convention, independent of any specific project, enhancing the readability for both humans and automated tools. Compared to other conventions that are mainly customized for internal project use, CCS aims to be a universal standard. To the best of our knowledge, CCS is currently the most widely used commit convention, which can be evidenced by the fact that mainstream commit lint tools are developed specifically for CCS [7, 10]. Figure 1 illustrates the required format for a conventional commit message and an example. CCS utilizes a mandatory <*type*> prefix to categorize commits into ten distinct types that indicate the essence of the commit. Popular types include "feat" (feature), "fix" (bug fix), "docs" (documentation), "refactor", "ci" (continuous integration), "build" , etc [9]. Aligning with Swanson's taxonomy, "feat" corresponds to adaptive changes and "fix" to corrective changes, while other types represent various perfective modifications.

```
<type>[optional scope]: <description>

[optional body]

[optional footer(s)]
```

```
fix: prevent racing of requests

Introduce a request id and reference to latest
request. Dismiss incoming responses other than
from latest request.

Reviewed-by: Z
Refs: #123
```

Fig. 1: **Conventional commit format and an example**

While commit classification based on Swanson's categories is well explored [3, 11, 12, 13, 14, 15], the classification based on the CCS, to the best of our knowledge, remains an unresolved gap. CCS is a relatively new convention for commit messages

†Corresponding Author

compared to other conventions like semantic versioning [16]. In recent years, we have observed a growing trend in the open-source community where more and more projects and developers are adopting the CCS. Moreover, an interesting phenomenon has been observed: Some commits conform to CCS but are misclassified into incorrect categories, as illustrated in Figure 2, where the commit is derived from InfluxDB[1], a popular Rust database project. This commit removes some unnecessary lines of code, thus falling under the "refactor" type rather than the "feat" type. Therefore, in this study, we first investigate the adoption trend of CCS to gauge its prevalence and application within the open-source community. We then delve deeper into the challenges developers face when classifying commits into CCS categories, aiming to understand the reasons behind apparent misuses of commit types as shown in Figure 2. Based on our findings, we propose an approach for the automated classification of commits into CCS types.

```
Sha: fba4326c72fc22d81aba6976a9fef1e4b6154fd9
Date: Jun 28, 2019

feat(storage): remove unnecessary lines from
verify-wal test
```

Fig. 2: **Example of a commit misusing the <*type*> prefix**

In our **RQ1**: "How common do projects apply Conventional Commits Specification?", we selected the top 500 GitHub projects based on the star counts across seven mainstream programming languages. After removing duplicates, we analyzed a total of 3,058 state-of-the-art open-source projects. Our findings reveal a consistent increase in CCS usage among projects and individual developers. By 2023, 116 of the 3,058 top-starred projects have incorporated CCS as their commit convention, and this number is steadily increasing. Additionally, in projects not yet adopting CCS, nearly 10% of their commits submitted in 2023 followed the CCS format, indicating its rising popularity in open-source communities.

Motivated by the phenomenon where developers may misclassify commits (as shown in Figure 2), we propose **RQ2**: "What challenges do developers face when classifying commits into CCS types?". To unveil these challenges, we conducted a thematic analysis of all issues of CCS in their GitHub repository[2] and the top 100 questions on Stack Overflow related to developers' problems with CCS. Our analysis reveals that developers do encounter challenges when classifying commits into CCS types due to the lack of precise definitions assigned to each type. Furthermore, developers rely on the definitions provided by Angular projects [4], which are often ambiguous and exhibit overlaps in their definitions. Consequently, developers may erroneously use types in their commit messages in certain contexts. To tackle this issue, we propose a clearer definition based on existing definitions and a literature review.

[1]https://github.com/influxdata/influxdb

[2]https://github.com/conventional-commits/conventionalcommits.org/issues

Automatical conventional commit classification helps developers categorize commits while writing conventional commit messages, saving time and reducing errors. It also may facilitate the transformation of older non-CCS-compliant commit history into CCS-compliant, making them compatible with CCS-based automation tools. (See Section II-A for more details). In academia, conventional commit classification can also serve as a tool for empirical studies in software engineering, such as developer skill assessment [17] or characterizing companies' performance in open-source software [18]. Therefore, in our **RQ3**: "Can we automatically classify commits into CCS types?", we first investigate the potential for automated conventional commits classification. Compared to previous studies that focus on three basic categories [3, 11, 12, 13, 14, 15], automatically classifying commits into the ten finer-grained CCS types presents a greater challenge, requiring a deeper understanding of code changes. We harnessed the code comprehension capabilities of Large Language Models (LLMs), fine-tuning an LLM for conventional commit classification. Our Approach demonstrates superior classification performance compared to various baselines and ChatGPT4 [19], the most advanced LLM developed by OpenAI.

The main contributions of this work are as follows:

1) We conducted a thorough investigation into the trends of CCS adoption among projects and developers in the open-source community and identified two distinct patterns of CCS application within projects.
2) We uncover four challenges developers face when classifying commits according to CCS, which is due to the lack of a clear and non-overlapping list of type definitions in CCS. To move this forward, we propose a clearer and less overlapping definition list based on industry practices and literature review.
3) We have developed a model that classifies commits into fine-grained conventional commits types (10 categories). This model surpasses a range of baseline models in classification performance, including ChatGPT-4, achieving state-of-the-art results.

The remainder of the paper is organized as follows. Section II provides the necessary background, and Section III introduces our dataset. Sections IV to VI detail the methods and findings of each of the three research questions. Section VII discusses the practical and academic implications based on our results. Threats to validity are addressed in Section VIII, and the paper concludes with Section IX.

II. Background

In this section, we begin by introducing the Conventional Commits Specification, detailed in Section II-A. Subsequently, we discuss the related work of commit classification, as elaborated in Section II-B. Given that our approach adopts a Large Language Model (LLM) to commit classification, Section II-C is dedicated to presenting the background of LLMs and the techniques for fine-tuning LLMs.

A. Conventional Commits Specification

Conventional Commits Specification (CCS) is a specification for adding human- and machine-readable meaning to commit messages [9]. This specification provides an easy set of rules for creating an explicit commit history. It mandates that each commit message conforms to a specific format, as Figure 1 demonstrates. Within this format, the fields <*type*> and <*description*> are mandatory, while others are optional. The prefix <*type*> field, in particular, is used to denote the essence of code changes. It encompasses a range of ten options, including but not limited to "feat" for introducing a new feature, "fix" for bug fixes, and "docs" for documentation changes. The <*description*> field succinctly summarizes the commit, whereas the optional body field provides a more expansive explanation of the changes. Additionally, the optional scope field specifies the particular area impacted by the commit, and the optional footers section contains pertinent metadata about the commits, such as associated GitHub issues, reviewers, assigners, and more.

On one hand, the CCS assists in communicating the nature of changes to teammates, the public, and other stakeholders. It makes it easier for people to contribute to projects by allowing them to explore a more structured commit history. On the other hand, its machine-readable format paves the way for the development of automated tools. Taking semantic versioning [16] as an example, where version numbers comprise three parts: *major.minor.patch*. Here, the *major* version signals a breaking change, a *minor* version increment indicates the addition of a new backwards-compatible feature, and a *patch* version increment denotes a bug fix. CCS dovetails with SemVer [16] by detailing the features, fixes, and breaking changes in commit messages, thereby facilitating the automatic determination of semantic version bumps[3]. Additionally, it aids in the automatic generation of release notes [20] and changelogs based on the types of commits landed [7]. Therefore, CCS has been widely adopted, especially in open-source projects and agile development teams.

B. Related Work

In the sphere of software engineering, the application of the Conventional Commits Specification for commit classification remains underexplored, with the bulk of existing research anchored in Swanson's categorization of corrective, perfective, and adaptive changes [2]. These studies often leverage commit messages as a primary information source, supplementing them with code change features to enhance classification accuracy [21]. Pioneering works by Mockus et al. [3] and Hindle et al. [22] utilized word frequency analysis to represent commit messages for classification purposes. Levin and Yehudai [12] innovated further by introducing one-hot keyword vectors for representing commit messages. They also leveraged ChangeDistiller [23] to extract features from code changes, thereby enhancing the accuracy of commit classification. Gharbi et al. [24] and Sarwar et al. [15] conceptualized commit classification as a multi-label rather than a multi-class task. Gharbi et al. [24] used Term Frequency-Inverse Document Frequency (TF-IDF) to encode commit messages and employed active learning to reduce the volume of data requiring labeling. Sarwar et al. [15] applied BERT to encode commit messages. Further advancing the field, Ghadhab et al. [11] extracted features related to bug fixes and refactoring from code changes and employed BERT to encode commit messages. Similarly, Lee et al. [13] utilized CodeBERT [25] to extract features from code changes automatically and incorporated these with commit message features for enhanced classification.

While several works [26, 27] focused on commit message generation have mentioned CCS, these studies typically preprocess CCS-compliant commit messages by removing their prefix <*type*> to render them into a format suitable for model training.

To the best of our knowledge, we are the first to investigate the application of the CCS within the open-source community, encompassing trends, challenges, and solutions. We first delve into CCS-based commit classification, classifying commits into ten more detailed categories extensively utilized in the industry.

C. Large Language Models

The development of large language models (LLMs) has undergone four stages, evolving from Statistical Language Models (SLMs) to Neural Language Models (NLMs), then to Pre-trained Language Models (PLMs), and finally to the recent advent of Large Language Models (LLMs) [28]. Compared to their predecessors, a notable distinction of LLMs lies in their scalability. The research found that scaling the size of PLMs (model size or data size) unlocks surprising abilities in solving a series of complex tasks that may not be observed in previous smaller PLMs [29]. LLMs, such as ChatGPT [19], and LLaMa2 [30], are built upon multi-layer Transformer architectures [31] and possess hundreds of billions of parameters, trained using massive text data. In software engineering, LLMs have shown remarkable performances in tasks such as code generation [32], program repair [33], and pull request review generation [34].

Traditionally, fine-tuning large-scale models necessitates updating all or a substantial fraction of their parameters. This process can be both computationally burdensome and time-intensive [35]. Low-Rank Adaptation (LoRA) [36] introduces a novel parameter-efficient fine-tuning approach. It involves retaining the original large model's parameters in a static state while integrating adaptable low-rank matrices into the transformer layers. Specifically, for a pre-trained parameter $W_0 \in \mathbb{R}^{d\times k}$, LoRA facilitates its modification via a low-rank decomposition expressed as $W_0 + \triangle W = W_0 + BA$. Here, $B \in \mathbb{R}^{d\times r}$ and $A \in \mathbb{R}^{r\times k}$, with the rank r, constrained by $r \leq \min(d, k)$, serving as a hyperparameter. Throughout the fine-tuning phase, LoRA keeps W_0 unchanged, updating only the trainable matrices A and B. This methodology effectively reduces the parameters requiring updates from $O(d \times k)$ to $O(r \times (d + k))$, thereby considerably diminishing the computational resources and time needed for model fine-tuning.

[3]https://github.com/googleapis/release-please-action

In this work, we utilize Meta's open-source CodeLlama [37] as our base model and fine-tune it using LoRA [36]. CodeLlama, initialized with Llama2 [30] parameters and trained on a variety of code-related tasks, has achieved state-of-the-art performance in numerous code-related benchmarks [37], making it an ideal choice for our commit classification task.

III. Dataset

We investigate the adoption of Conventional Commits Specification (CCS) in leading open-source projects with significant followings and user bases. These projects typically have frequent updates and commits, providing an ideal environment for examining CCS in real-world applications. We focus on seven prominent programming languages: Go, JavaScript, Python, Rust, Java, C++, and TypeScript. We select the 500 most-starred projects for each language utilizing GitHub's REST API[4]. A deduplication process is necessary because of the multi-language development approach of certain projects. This yielded a final dataset of 3,058 unique projects with a median star count of 4,661, underscoring their prominent role in the open-source community.

CCS offers detailed guidelines on crafting commit messages on its website "conventionalcommits.org". We observed that repositories using CCS typically include this URL in their documentation, directing developers to CCS for a thorough understanding, akin to a citation mechanism. Therefore, in this study, we used the URL "conventionalcommits.org" as a keyword to accurately identify repositories that explicitly adopt CCS. Upon searching this keyword in each project's codebase, we found that 119 repositories mentioned "conventionalcommits.org" in their documentation. To ensure accuracy, we manually reviewed these 119 repositories to confirm their use of CCS as their commit convention. During this step, we discovered that three repositories mentioned CCS but did not adopt it as their commit convention. Consequently, we removed the three repositories, resulting in 116 repositories that definitively use CCS. These 116 repositories encompass a wide range of software domains, including cross-platform application frameworks, JavaScript runtimes, data analysis tools, etc. Table I provides select examples from our dataset, as well as the number of projects for each language represented in our dataset. We provide the full list of these 116 repositories in our online appendix [38].

TABLE I: **Dataset Overview**

Language	#Projects	Example Project	#Example Project Star
Go	19	kratos	21,698
JavaScript	5	ui-grid	5,394
Python	3	flagsmith	3,623
Rust	31	deno	91,452
Java	6	dataease	13,409
C++	4	electron	109,712
TypeScript	48	nuxt	48,531

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

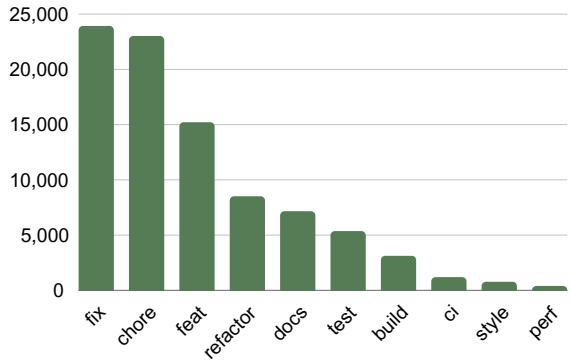

Fig. 3: **Number of commits for each CCS type in the dataset**

The CCS format can be parsed using regular expressions. We utilized *conventional-pre-commit*[5], a widely-used Python-based check hook for CCS, to verify whether commits adhere to the CCS format. Among the 116 repositories, we identified a total of 276,052 commits that comply with CCS. We excluded non-English commit messages and applied the bot list suggested by Tian et al. [39] to filter out commits generated by bots, such as Dependabot [40]. As a result, our dataset included 88,704 developer-crafted conventional commits from 116 state-of-the-art projects that explicitly adopt CCS as their commit message convention. Figure 3 illustrates the distribution of commits across various CCS types within our dataset.

IV. RQ1: How common do projects apply Conventional Commits Specification?

A. Method

We answer this RQ from two aspects: repositories and developers. From the repository perspective, we aim to know how common projects officially declare their usage of Conventional Commits Specification (CCS). When a repository adopts CCS, it will specify this in its documentation, such as contributing guideline [41], mandating developers to adhere to CCS. From the developer perspective, we are interested in exploring how many developers apply CCS as part of their routine development practice, i.e., adhering to CCS in their commit messages, no matter whether their contributed projects mandate CCS or not.

As introduced in Section III, we have collected data from 116 projects that explicitly adopt CCS as their commit message convention within their codebases. By analyzing their git commit history, we traced the timeline of CCS adoption in these projects to investigate the adoption trend. To better understand how projects use CCS as their commit convention, we dive into these 116 projects' codebases and conduct a manual analysis to explore how they incorporate CCS as their commit convention.

To assess the trend from the developers' standpoint, we hypothesize that if a developer uses CCS, they will write CCS-compliant commit messages, regardless of whether the projects they contributed to adopt CCS. Therefore, we measure the

[5]https://github.com/compilerla/conventional-pre-commit

trend of CCS adoption among developers by calculating the proportion of conventional commits within the total annual commits of projects that have not yet adopted CCS. For this, we considered the 3,058 projects mentioned in Section III, excluding the 116 explicitly CCS-using projects, and then randomly sampled 500 projects. To ensure these repositories do not use CCS, we conducted additional keyword searches, such as "conventional commit". Similar to the retrieve method used in Section III, we employed *conventional-pre-commit* to identify conventional commits, thereby calculating the annual proportion of conventional commits.

B. Results

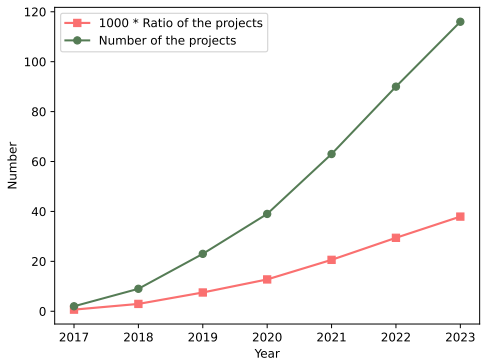

Fig. 4: **Application Trend of Conventional Commits Specification in Projects**

Figure 4 shows the number of projects applying CCS from 2017 to 2023. We also calculated the proportion of projects applying CCS among all the 3,058 projects we selected from seven prominent programming languages. We can see that an increasing number of state-of-the-art open-source projects are adopting the CCS, exhibiting a consistent growth rate annually. Specifically, in 2023, 116 out of 3,058 most popular projects officially announced their adoption of CCS, which is still increasing stably. We took a close look at the 116 projects and categorized two modes of CCS usage within the open-source community: *Document Declaration* and *Integrated with Automatic Tools*. We introduce the two modes as follows:

a) ***Document Declaration*** *(#109):* In this mode, projects explicitly state in their documentation that they use CCS as their commit message convention and require developers to follow CCS when submitting their commits. The most common documentation are contributing guidelines, changelogs, and GitHub PR templates. Of the 109 projects, 66 directly state in their contributing guidelines the necessity to adhere to CCS standards by providing a link to the CCS. For instance, the CONTRIBUTING.md of goreleaser [42] state: *"Commit messages should be well formatted, and to make that "standardized", we are using Conventional Commits. You can follow the documentation on their [website](URL)."* Besides, 43 (out of the 109) projects go further by offering detailed explanations in their contributing guidelines, such as providing explanations for CCS types to aid developers in understanding CCS, as in the case of react-navigation [43].

b) ***Integrated with Automatic Tools*** *(#64):* In this mode, projects utilize various automation tools with CCS. Commonly used tools include GitHub Actions, commit lint [10], and git-cliff [7]. For example, hyperswitch [44] checks if PR titles (which will be the default commit messages after a squash merge [45]) adhere to CCS in their GitHub actions CI/CD. Among the 109 projects that explicitly state the use of CCS in their documentation, 57 projects have adopted some automation tools for checks. Seven projects adopted automation tools without an explicit declaration.

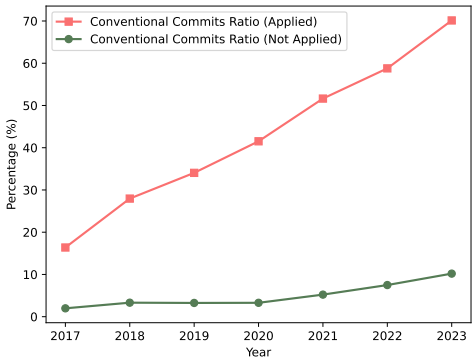

Fig. 5: **Application Trend of Conventional Commits Specification in Developers' Commits**

Figure 5 presents the annual proportion of commits conforming to the CCS format out of all commits produced from 2017 to 2023. As expected, we can see a growing trend in the percentage of CCS-compliant commits within projects adopting the standard (indicated by the red line). By 2023, nearly 70% of the commits in these repositories were in line with the CCS format. Notably, even in projects where developers are not required to follow CCS, we also see an upward trend in the proportion of CCS-compliant commits (represented by the green line). This proportion grew from almost zero in 2017 to nearly 10% in 2023. This suggests that an increasing number of developers are beginning to adopt CCS as part of their development practices.

Summary for RQ1: The Conventional Commits Specification (CCS) has gained increasing popularity among open-source projects and developers. On GitHub, a growing number of popular projects are adopting the CCS as their commit convention. In projects that have not adopted CCS, approximately 10% of the commits submitted by developers in 2023 adhered to CCS. There are two modes of CCS usage in open-source projects: explicitly requiring developers to adhere to CCS in documentation, and using automation tools to enforce CCS compliance.

V. RQ2: What challenges do developers face when classifying commits into CCS types?

A. Method

The Conventional Commits Specification (CCS) hosts a repository on GitHub [9], where users typically report issues when encountering problems with CCS. As of December 15, 2023, the CCS repository comprises 194 issues, with 70 open and 124 closed. These issues are an important source for us to mine challenges developers encountered when classifying commits with CCS. Besides, we also considered another important source, i.e., Stack Overflow, a prominent online question-and-answer site for computer programming. Specifically, we analyze the top 100 questions most relevant to the keyword "conventional commits". We first conducted a manual annotation process on the aforementioned sources to identify the GitHub issues and Stack Overflow questions that are specific to the challenges developers face when classifying commits into CCS types. Subsequently, for the issues or questions identified, we employ thematic analysis, a widely-used qualitative analysis method [46], to categorize the types of challenges developer encountered based on their characteristics.

1) Data Annotation: We conducted the annotation of the 194 CCS-related issues and 100 Stack Overflow questions as follows: The first two authors independently conducted a thorough review of each GitHub issue and Stack Overflow question. The objective was to ascertain whether these issues and questions were pertinent to the challenges developers face in classifying commits into CCS types. For every GitHub issue, we considered the conversations between the issue reporter and other contributors. Similarly, for each Stack Overflow question, both the questions and their respective answers were examined. In instances where there was a discrepancy in the annotations between the two authors, a third author was brought in to serve as an arbitrator.

2) Thematic Analysis: Upon completing the annotation phase, we identified 43 GitHub issues and 9 Stack Overflow questions that are specific to developers' challenges when classifying commits. Specifically, the total views of these identified Stack Overflow questions exceed 73,000[6], suggesting that the challenges are not limited to a small subset of developers. Subsequently, we employed thematic analysis to extract the categories of the challenges. Thematic analysis [46] is a technique for examining semantic data, involving pattern identification to uncover themes. Specifically, our thematic analysis procedure was as follows:

1) Perform an in-depth review of the issues and questions related to developers' problems of classifying commits to comprehend the challenges developers face.
2) Re-examine these issues and questions to generate initial codes.
3) Analyze the initial codes systematically, grouping those that indicate similar challenges. During this stage, we assigned aggregated themes to these grouped codes.
4) Review and refine these initial themes, merging those with overlapping meanings for clarity and conciseness.
5) Establish the final set of themes applicable to all reviewed issues and questions.

To minimize bias, the first two authors independently conducted steps 1 to 4. We also convened multiple meetings to resolve discrepancies and finalize the themes (step 5). Further details are available in our online appendix [38].

[6]Since GitHub issues do not provide data about their page views, we take Stack Overflow questions as a proxy.

B. Results

1) Identified Challenges: Through thematic analysis, we identified four distinct categories of challenges developers have encountered when using the Conventional Commits Specification (CCS), i.e., (1) Requests for changing types, (2) Requests for adding type aliases, (3) Confusion about which type to use, and (4) Request for comprehensive definition.

*a) **Requests for changing types** (#9, 17.3%):* There are 9 instances of 52 (17.3%) identified issues or problems that fall under this category. In this category, developers suggested introducing new types (#8) or removing existing ones (#1), to meet the specific needs of their projects or personal preferences. The requested types can be classified into two categories: management and customization. Management types pertain to actions that help manage the codebase, such as introducing "wip" type for half-finished commits or "revert" type for commits created by git revert operations. Customization types are specific to the nature of the commit, like adding a "security" type to label commits that address security-related bug fixes. While both involve introducing new types, management focuses on workflow control, whereas customization emphasizes the essence of commits. One issue raised was the suggestion to remove the "chore" type as claimed that its definition (e.g. "daily work") is too broad and leads to misuse.

*b) **Requests for adding type aliases** (#10, 19.2%):* This category encompasses challenges developers face regarding the naming of types, often stemming from personal preferences. Developers have requested the provision of aliases for certain types. For instance, some developers proposed using "patch" as an alias for "fix", or "feature" for "feat". There was also confusion and debate over the choice of "docs" instead of "doc" as a type name for documentation changes.

*c) **Confusion about which type to use** (#30, 57.7%):* This category represents the most common challenge, where over half of the issues and Stack Overflow questions are related to developers' confusion about conventional commit types. This confusion is divided into practical and theoretical aspects. Practically, developers are unsure about the appropriate type for certain scenarios not covered or explicitly defined by CCS (#22), such as what types to use when updating a git sub-module or making an initial commit. Theoretically, developers grapple with understanding the definitions of types (#8). Notably, CCS does not provide a definitive list of type definitions but refers users to the Angular project's definitions [9]. This reliance leads to debates and confusion based on Angular's definitions [4]. For instance, there is uncertainty about whether the "feat" type should be used for user-oriented

features or developer-oriented features. Additionally, according to Angular's definition, "refactor" is defined as "A code change that neither fixes a bug nor adds a feature" [4], which is too broad to cause overlap confusion with other types like "style" (code style related changes) and "perf" (performance related changes), as commits conforming to "style" and "perf" might also fit the "refactor" definition.

*d) **Request for comprehensive definition** (#3, 5.8%):* With the prevalent confusion about commit types, some developers have requested a comprehensive list of definitions for each CCS type. The CCS community has recognized this challenge and initiated discussions on July 14, 2020, to introduce a definitive list. However, until the date of this study, these discussions remain open and have not seen significant progress[7].

2) Proposed Solution: The most common challenge identified is CCS's lack of a clear definition list for each type [9], relying instead on the Angular project's definitions [4], which are too broad and have overlaps, leading to confusion. To move this forward, we provide more detailed and less overlapping type definitions, thereby reducing confusion. We use the Angular project's definitions as a foundation [4], acknowledging their extensive use in CCS despite their broad and overlapping aspects. As detailed in Section IV-B, our investigation indicates that out of the projects employing CCS, 43 provide additional elucidations in their contributing guidelines, some including CCS types explanation. To render our definitions more concrete and illustrative, we meticulously reviewed the contributing guidelines of these 43 projects, aiming to integrate more examples and use cases into the Angular framework's existing definitions.

In this study, we will not consider the problems suggesting introducing new types in the first category of challenges identified, nor the challenges of the second category, which propose the incorporation of aliases. (The rationale for this will be discussed in Section VIII). Specifically, our study addresses three types of confusion identified above: (1) whether the "feat" type should target users or developers; (2) the distinction between "refactor" and other types; and (3) the potential removal of the "chore" type due to misuse. While varying in definitions, the concept of a feature generally leans towards a user-centric perspective [47, 48]. For instance, Chen et al. [49] describe features as "a product characteristic from user or customer views". However, determining a commit's user or developer orientation can be challenging for individuals not deeply familiar with the project's context. In this study, to be compatible with automatic commit classification, we include both user and developer-oriented features under "feat", while strongly recommending developers use specific markers to differentiate them (e.g., "feat+" for user-oriented, "feat" for developer-oriented features). Refactoring, theoretically, involves altering a program's structure without changing its functionality to enhance metrics like readability and maintainability [50, 51]. From this perspective, "style" (code style improvements) is a subset of "refactor", and "perf" (performance enhancements) can also be perceived as overlapping with "refactor" since they both do not change a program's functionality. However, according to [51], "perf" is not considered a "refactor". We advocate for specificity in type classification and propose to explicitly exclude "style" and "perf" from "refactor". Regarding the "chore" type, its broad definition (e.g., "daily work"[8]) led to its removal by the Angular team. However, it remains widely used in the community. In our study, we retain the "chore" type as "other", covering commits not encompassed by current types, such as resolving merge conflicts or removing debug code. The definitions of CCS types in this study are as follows:

1) **feature (feat/feat+)**: Code changes aim to introduce new features to the codebase, encompassing both internal and user-oriented features.
2) **fix**: Code changes aim to fix bugs and faults within the codebase.
3) **performance (perf)**: Code changes aim to improve performance, such as enhancing execution speed or reducing memory consumption.
4) **style**: Code changes aim to improve readability without affecting the meaning of the code. This type encompasses aspects like variable naming, indentation, and addressing linting or code analysis warnings.
5) **refactor**: Code changes aim to restructure the program without changing its behavior, aiming to improve maintainability. *To avoid confusion and overlap, we propose the constraint that this category does not include changes classified as "perf" or "style".* Examples include enhancing modularity, refining exception handling, improving scalability, conducting code cleanup, and removing deprecated code.
6) **documents (docs)**: Code changes that modify documentation or text, such as correcting typos, modifying comments, or updating documentation.
7) **test**: Code changes that modify test files, including the addition or updating of tests.
8) **ci**: Code changes to CI (Continuous Integration) configuration files and scripts, such as configuring or updating CI/CD scripts, e.g., ".travis.yml" and ".github/workflows".
9) **build**: Code changes affecting the build system (e.g., Maven, Gradle, Cargo). Change examples include updating dependencies, configuring build configurations, and adding scripts.
10) **chore**: Code changes for other miscellaneous tasks that do not neatly fit into any of the above categories.

It is important to recognize that these CCS types bifurcate commits based on two dimensions: purpose (i.e., the motivation behind making a code change) and object (i.e., the essence of code changes that have been made). Here, "perf", "style", "feat", "refactor", and "fix" are categorized as purpose types, whereas "docs", "test", "ci", and "build" fall under object types. Consequently, there is potential for overlap between purpose and object types. For instance, commits may involve refactoring

[7]github.com/conventional-commits/conventionalcommits.org/issues/283

[8]https://github.com/go-kratos/kratos/

purposes within the test object or fixing issues purpose in the CI object. To mitigate these overlaps, this study prioritizes purpose over object. That is to say, in cases of overlap, the commit type is determined by its purpose. For example, a commit that refactors test code would be categorized as a "refactor" commit. Meanwhile, this is also why we distinguish between "refactor" and "perf": if the intent is to boost performance, the changes are classified as "perf"; if the goal is to make the software easier to understand and modify, then the changes are recognized as "refactor" [51].

Summary for RQ2: When developers adopt the Conventional Commits Specification (CCS), they encounter several challenges. The most common challenges related to types, faced by approximately 60% of users, stem from the lack of a clear and explicit list of definitions for CCS types. This leads to confusion over type selection and understanding. To move this forward, we combined insights from industrial practices and a review of academic literature to provide a more precise, clear, and less-overlapping definition list for CCS.

VI. RQ3: Can we automatically classify commits into CCS types?

A. Method

In this RQ, we aim to automate the classification of commits into the types we defined in Section V-B. This more fine-grained classification task presents a more nuanced challenge compared to earlier studies that categorized commits into three basic classes: perfective, corrective, and adaptive [3, 11, 12, 13, 14, 15]. The categorization into ten distinct CCS types requires a deeper understanding of code changes. Recent advancements in Large Language Models (LLMs) have demonstrated significant potential in code understanding tasks, as detailed in Section II-C. Leveraging these advancements, we utilize CodeLlama [37], a sophisticated LLM tailored for code-related tasks. Developed and open-sourced by Meta, CodeLlama has achieved state-of-the-art performance across multiple benchmarks [37]. To adapt CodeLlama for this task, we first created an instruction dataset and fine-tuned CodeLlama using a LoRA-based methodology (described in Section II-C). The model's performance was evaluated against several baselines, including OpenAI's latest model, ChatGPT4 [19], as well as the previous three-category baseline.

*1) **Dataset Labeling**:* Section III details our collection of a dataset comprising 88,704 conventional commits crafted by developers from 116 popular open-source projects. While these commits contain <*type*> field in which developers self-classified their code changes, we find a prevailing confusion in understanding the CCS types among some developers (see more details in Section V-B). Such confusion may lead to the misclassification of commit types; Figure 2 shows a real example. Besides, we have improved the type definitions to address several identified challenges. Therefore, we manually annotated a dataset in alignment with our definitions from Section V-B for model fine-tuning and testing.

We employed stratified sampling on our dataset in Section III, randomly selecting 200 commits for each of the 10 CCS types, resulting in 2,000 commits. The first and third authors independently annotated these 2,000 commits according to our proposed definition list in Section V-B. Including various modification types within a single commit is typically considered poor practice, compromising code maintainability [52, 39]. Additionally, adhering to the principle that a commit should ideally represent a single CCS type, we excluded commits encompassing multiple types during the annotation process. For categories with fewer than 200 commits after annotation, we conducted additional random sampling and labeling until each category had 200 commits. After labeling, the two annotators have disagreements on 79 commits (about 4%), resulting in a Cohen's kappa coefficient [53] of 0.9, which indicates a high level of agreement. Discrepancies were addressed in several meetings, with a third author serving as an arbitrator when consensus was not reached. Ultimately, we obtained a dataset of 2,000 manually annotated commits, with each CCS type represented by 200 commits. Subsequently, for each commit in our dataset, we removed the <*type*> field from its commit message, converting it into a normal commit message. This adaptation is necessary for our task of classifying all commits into CCS types, especially considering that the majority of commits (approximately 90% in 2023, as detailed in Section IV-B) do not contain the <*type*> field in their messages. This adjustment also ensured that the model learned to classify commits based on the code changes and the content of normal commit messages rather than depending on the <*type*> field.

*2) **Baselines**:* While previous studies have focused on the three-category classification of commits [3, 11, 12, 13, 14, 15], our research pioneers the exploration into a ten-category conventional commits classification. The methods from previous works are not specifically designed for CCS, which could pose challenges when directly applied to CCS classification. The BERT model [54] is widely used in prior studies [11, 13, 15]; Ghadhab et al. [11] employed a BERT-based approach to encode commit messages, achieving state-of-the-art performance with an accuracy of approximately 78% when classifying commit into Swanson's proposed categories [2]. Therefore, we chose BERT as a baseline. Our approach employs the Large Language Model (LLM), i.e., CodeLlama [37], and thus we include comparative analyses with Llama2 [30] and ChatGPT4 [19]. Llama2 is Meta's latest open-source LLM widely used in academic research [30], while ChatGPT4 represents the most advanced LLM developed by OpenAI [19]. Our study utilizes the CodeLlama 7B model, encompassing roughly 7 billion parameters. Llama2, used for comparison, is also based on a 7-billion-parameter framework. While OpenAI has not publicly disclosed the exact parameter count of ChatGPT4, it is significantly higher than ChatGPT3.5, which possesses 175 billion parameters [55].

*3) **Prompt Design**:* Large Language Models (LLMs) are predominantly prompt-driven [56]. In designing an effective prompt, we adhered to the Input Semantics Prompt Pattern proposed by White et al. [56]. As depicted in Figure 6, our prompt incorporates the type definitions provided in Section V-B. These definitions (<*type_definitions*>) guide the LLM in classifying commits based on their messages and code changes. We utilized this prompt to fine-tune both CodeLlama [37] and Llama2 [30]. For ChatGPT4 [19], which is proprietary, we leveraged OpenAI's API[9] to send the test dataset to ChatGPT4 based on this prompt, retrieving the generated labels to assess its performance.

```
You are a commit classifier based on commit
message and code diff. Please classify the given
commit into one of the ten categories: docs, perf,
style, refactor, feat, fix, test, ci, build and
chore. The definitions of each category are as
follows:

<type_definitions>

- The given commit message <commit_message>

- The given commit diff: <commit_diff>
```

Fig. 6: **Prompt Template**

*4) **Implementation Details**:* We implemented our approach using the HuggingFace transformers framework[10]. The dataset was stratified and sampled according to the CCS types, and then divided into training, validation, and test sets in a 7:1:2 ratio. All models were trained on the training set, with the best-performing model on the validation set selected for testing on the test set. The learning rate was set to 1×10^{-4}, with a batch size of 20 and a token length limit of 1,024. We employed the AdamW optimizer [57] and trained our model for five epochs. The hyperparameters of the Low-Rank Adaptation (LoRA) [36] were set with $r = 8$ and $\alpha = 32$. For evaluation purposes, given the nature of this task as a classification task, we employed widely recognized metrics: accuracy, macro precision, macro recall, and macro F1 score [58]. These macro metrics are calculated by computing the metrics for each category and then taking their arithmetic mean. In addition, we report precision, recall, and F1 scores for each category. Due to space constraints, we provide detailed results of these evaluations in our online appendix [38].

All experiments were conducted on a server equipped with two 24G NVIDIA GeForce RTX 3090 GPUs, an Intel(R) Xeon(R) Silver 4310 CPU, and running the Ubuntu 20.04.2 LTS operating system.

B. Results

The results of our comparative analysis are presented in Table II, with the highest scores for each metric emphasized in **bold**. Detailed comparisons of precision, recall, and F1 scores for each of the ten conventional commit types are available in our online appendix [38]. The table shows that our approach surpasses all baselines in both macro metrics and accuracy. When compared to earlier methods that categorized commits into three classes, our approach, in the fine-grained scenario of ten classifications, shows an approximate 22% increase in both accuracy and macro F1 scores. Despite our approach having a parameter count of only 7 billion, which is far less than 4% of that of ChatGPT4 (whose parameter count significantly surpasses ChatGPT3.5's 175 billion [55]), our lightweight model demonstrates a near 7% improvement in accuracy and macro F1 over ChatGPT4. This result suggests that our approach is both cost-efficient and accurate and capable of running on less expensive, energy-efficient devices. Compared to the fine-tuned Llama2, our approach reveals a slight performance improvement, potentially attributable to the additional code-related data pretraining conducted by CodeLlama built upon Llama2's weights [37].

TABLE II: **Classification performance against baselines**

Metrics	BERT	ChatGPT4	Llama2	Our Approach
Macro precision	54.32%	74.86%	75.43%	**77.95%**
Macro Recall	54.25%	69.50%	75.25%	**76.50%**
Macro F1	52.80%	69.90%	74.97%	**76.41%**
Accuracy	54.25%	69.50%	75.25%	**76.50%**

Regarding the specific performance of our approach across various conventional commit types, it is observed that the "chore" and "refactor" categories demonstrate weaker performances, with F1 scores of 53.97% and 60.61%, respectively. In contrast, the remaining categories all exceed 68% in F1 scores, with the "test" category achieving the highest score of 90.91%. Detailed findings can be found in our online appendix [38]. The relatively poor performance of the "chore" and "refactor" categories may be attributed to our definitions, wherein "chore" encompasses miscellaneous types, requiring the model to consider the remaining nine types to determine if a commit falls under the "chore" category. Similarly, "refactor" is defined as refactoring excluding "perf" and "style," necessitating additional assessment by the model. As a result, these two categories may require more detailed evaluation from the model, leading to their lower performance. This is a potential area for improvement in the future.

Summary for RQ3: Our approach demonstrates significant potential in conventional commits classification, surpassing all compared baselines. Compared to traditional three-category classification methods, our approach exhibits an over 22% improvement in accuracy and macro F1 scores. Relative to ChatGPT4, our method is more cost-efficient and exhibits a performance advantage of nearly 7%.

VII. Implications

This section discuss the implication of conventional commit classification for developers and researchers.

[9]https://platform.openai.com/docs/api-reference/chat/create
[10]https://github.com/huggingface/transformers

Implications for Developers. CCS plays a crucial role in supporting numerous automated tools in industry practices, such as automatic semantic version bump[11], changelog generation[12], and release note generation [7]. All these tools depend on the machine-readable <*type*> field present in conventional commit messages to extract categories of commits. For example, git-cliff [7], a widely-used tool for automatic changelog generation, organizes commit messages into different change categories by identifying the <*type*> field in conventional commit messages, thereby enabling the automated creation of changelogs. Furthermore, as revealed in our RQ2, developers may face challenges in selecting the appropriate types due to the absence of detailed type definitions in the CCS. Our automatic classifier, developed in RQ3, has shown promising potential in the classification of conventional commits. On one hand, this classifier can take any commit as input, which may enable the transformation of older, non-CCS-based history commits into CCS-compliant commits, thereby supporting CCS automation toolchains. Table II shows promising results for CCS-based commits, but further experiments are needed on non-CCS-based commits. On the other hand, it can recommend suitable types for developers, enhancing efficiency and reducing error likelihood. For projects adopting CCS as their commit message convention, our classifier can also be integrated into CI/CD pipelines. It automatically checks whether the commits submitted by developers are correctly categorized, thereby improving the quality of conventional commit messages.

Implications for Researchers. In the academic sphere, traditional three-category commit classification has been adopted as a tool for empirical research in software engineering, especially for understanding patterns in commits and developer behavior [59, 17, 39]. For instance, Zhang et al. utilized a commit classifier proposed by dos Santos and Figueiredo [60], which is based on Swanson's definitions, to identify the type of work carried out in commits while comparing paid and volunteer developers in the Rust community [59]. Our study introduces an automatic classifier capable of categorizing commits into ten more fine-grained categories with promising accuracy. This advancement can provide these empirical studies with a more nuanced set of features regarding commits, potentially facilitating more insightful findings. Additionally, many recent studies have focused on commit message generation [61, 62, 26, 27, 63]. However, these studies primarily target normal commit message generation without attempting to adhere to specific conventions like CCS. One of the challenges in generating conventional commit messages, as opposed to normal commit messages, is the generation of the <*type*> field. Our conventional commit classification approach has the potential to improve the generation of CCS-compliant commit messages.

VIII. Threats to validity

This section discusses potential internal and external threats to the validity of our study.

[11]https://github.com/googleapis/release-please-action

[12]https://github.com/cocogitto/cocogitto

Internal Validity concerns the threats to how we perform our study. The first threat lies in our sampling strategy, specifically using 200 commits per type. We found that developers may misclassify commits due to their confusion about CCS types, and our proposed type definitions slightly differ from previous ones (with "chore" types as "other"), necessitating manual dataset annotation for model training and testing. The choice of 200 commits for each type may not be optimal, but it is a practical compromise given the constraints of manual annotation in terms of time and effort. Manual annotation introduces a subjective element, posing a threat to validity. To mitigate this, two authors independently labeled commits, and several meetings were held, with the third author acting as an arbitrator for any inconsistent labeling. The agreement level of 0.9 is high, indicating a high level of consistency in the labeling process. Another threat is that our study does not consider requests to introduce new types (Section V-B). This is because CCS advises developers to create new types tailored to their specific needs. Instead, we focus on the most commonly used types as listed in CCS [9]. Furthermore, we do not consider requests for aliases to types because these requests are limited to individual preferences and are not directly related to the essence of commit types.

External Validity pertains to the threats that might impact the generalizability of our findings. Our study only focuses on the popular open-source projects on GitHub when investigating the trend of CCS adoption (Section IV). Given the time and computational resource constraints, examining all projects on GitHub is not feasible. Consequently, our research begins with these popular projects, underpinned by the hypothesis that such projects, owing to their larger developer base and more frequent updates, will likely have more comprehensive contributing guidelines. This, in turn, increases the probability of these projects adopting CCS to standardize contributors' commits. Our findings suggest a growing adoption of CCS among these projects, offering insights into broader trends on GitHub. Additionally, despite our efforts to develop a clearer and less overlapping type definition list compared to previous versions [4], based on industry practices and a thorough literature review (Section V-B), our proposed type definition list is still relatively incremental and may not encompass all commit activities and could still contain overlaps. This limitation opens a potential avenue for future research to further refine and evolve the conventional commits specification to align with modern software development practices. As seen in Figure 3 (Section III), the commit types in our dataset are not uniformly distributed. However, to train, validate, and test our model in a balanced manner, we constructed a dataset with evenly distributed labels. This approach might lead to slight differences in classification performance in real-world applications compared to the performance reported in this paper.

IX. Conclusion

Conventional Commits [9], as a specification for adding both human and machine readable meaning to commit mes-

sages, is increasingly gaining popularity among open-source projects and developers. The machine-readable type of these commits enables numerous automation tools, such as automatic changelog or release note generation, many of which have been integrated into production applications. However, current studies predominantly focus on the classification framework established by Swanson in 1976 [2], which is not extensively utilized by developers and projects today. To bridge this gap, our study conducts a preliminary study of CCS, encompassing its application status and the challenges developers encounter when using it. We observe a growing popularity of CCS, yet developers do misclassify commits into incorrect CCS types, attributable to the absence of a clear and distinct definition list for each type. We have developed a more precise and less overlapping definition list to address this, grounded in industry practices and literature review. To assist developers in classifying conventional commits, we propose an approach for automated conventional commit classification. Our evaluation demonstrates that our approach outperforms a series of baselines as well as ChatGPT4, showcasing promising potential for both industrial and academic applications.

X. Data Availability

To support future replication and research, we have made the data, scripts, and additional resources utilized in this study available online. These materials can be accessed at the following URL: https://doi.org/10.6084/m9.figshare.26507083.

Acknowledgments

This work is supported by the National Natural Science Foundation of China Grant (62141209, 62202048, and 62232003).

References

[1] N. N. Zolkifli, A. Ngah, and A. Deraman, "Version control system: A review," *Procedia Computer Science*, vol. 135, pp. 408–415, 2018.

[2] E. B. Swanson, "The dimensions of maintenance," in *Proceedings of the 2nd international conference on Software engineering*, 1976, pp. 492–497.

[3] Mockus and Votta, "Identifying reasons for software changes using historic databases," in *Proceedings 2000 International Conference on Software Maintenance*. IEEE, 2000, pp. 120–130.

[4] Angular, "Commit guidelines for angular." [Online]. Available: https://github.com/angular/angular/blob/main/CONTRIBUTING.md#-commit-message-format

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

[6] Ember.js, "Commit guidelines for ember." [Online]. Available: https://github.com/emberjs/ember.js/blob/main/CONTRIBUTING.md

[7] "git-cliff," https://github.com/orhun/git-cliff, accessed: 2024-03-15.

[8] "semantic-release," https://github.com/semantic-release/semantic-release, accessed: 2024-03-15.

[9] "Conventional commit specification." [Online]. Available: https://www.conventionalcommits.org/en/v1.0.0/

[10] "commitlint," https://github.com/conventional-changelog/commitlint.

[11] L. Ghadhab, I. Jenhani, M. W. Mkaouer, and M. B. Messaoud, "Augmenting commit classification by using fine-grained source code changes and a pre-trained deep neural language model," *Information and Software Technology*, vol. 135, p. 106566, 2021.

[12] S. Levin and A. Yehudai, "Boosting automatic commit classification into maintenance activities by utilizing source code changes," in *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2017, pp. 97–106.

[13] 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)*, 2021, pp. 389–395.

[14] S. Hönel, M. Ericsson, W. Löwe, and A. Wingkvist, "Using source code density to improve the accuracy of automatic commit classification into maintenance activities," *Journal of Systems and Software*, vol. 168, p. 110673, 2020.

[15] M. U. Sarwar, S. Zafar, M. W. Mkaouer, G. S. Walia, and M. Z. Malik, "Multi-label classification of commit messages using transfer learning," in *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2020, pp. 37–42.

[16] T. Preston-Werner, "Semantic versioning 2.0.0," 2013. [Online]. Available: https://semver.org/

[17] S. Amreen, A. Karnauch, and A. Mockus, "Developer reputation estimator (dre)," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1082–1085.

[18] Y. Zhang, K.-J. Stol, H. Liu, and M. Zhou, "Corporate dominance in open source ecosystems: a case study of openstack," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1048–1060.

[19] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[20] J. Wu, W. Xu, K. Gao, J. Li, and M. Zhou, "Characterize software release notes of github projects: Structure, writing style, and content," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2023, pp. 473–484.

[21] T. Heričko and B. Šumak, "Commit classification into software maintenance activities: A systematic literature review," in *2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2023, pp. 1646–1651.

[22] A. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt, "Automatic classication of large changes into maintenance

categories," in *2009 IEEE 17th International Conference on Program Comprehension*. IEEE, 2009, pp. 30–39.

[23] B. Fluri, M. Wursch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on software engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[24] S. Gharbi, M. W. Mkaouer, I. Jenhani, and M. B. Messaoud, "On the classification of software change messages using multi-label active learning," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, 2019, pp. 1760–1767.

[25] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, T. Cohn, Y. He, and Y. Liu, Eds. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139

[26] T. H. Jung, "CommitBERT: Commit message generation using pre-trained programming language model," in *Proceedings of the 1st Workshop on Natural Language Processing for Programming*, 2021, pp. 26–33.

[27] M. Miksik, "Fine-tuning transformer models for commit message generation and autocompletion," January 2023. [Online]. Available: http://essay.utwente.nl/94380/

[28] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong *et al.*, "A survey of large language models," *arXiv preprint arXiv:2303.18223*, 2023.

[29] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler, E. H. Chi, T. Hashimoto, O. Vinyals, P. Liang, J. Dean, and W. Fedus, "Emergent abilities of large language models," *Transactions on Machine Learning Research*, 2022, survey Certification.

[30] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[31] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[32] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," in *The Eleventh International Conference on Learning Representations*, 2023.

[33] C. S. Xia and L. Zhang, "Conversational automated program repair," *arXiv preprint arXiv:2301.13246*, 2023.

[34] J. Lu, L. Yu, X. Li, L. Yang, and C. Zuo, "Llama-reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2023, pp. 647–658.

[35] M. Weyssow, X. Zhou, K. Kim, D. Lo, and H. Sahraoui, "Exploring parameter-efficient fine-tuning techniques for code generation with large language models," *arXiv preprint arXiv:2308.10462*, 2023.

[36] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "LoRA: Low-rank adaptation of large language models," in *International Conference on Learning Representations*, 2022.

[37] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[38] "Online appendix." [Online]. Available: https://doi.org/10.6084/m9.figshare.26507083

[39] 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*, 2022, pp. 2389–2401.

[40] R. He, H. He, Y. Zhang, and M. Zhou, "Automating dependency updates in practice: An exploratory study on github dependabot," *IEEE Transactions on Software Engineering*, 2023.

[41] F. Fronchetti, D. C. Shepherd, I. Wiese, C. Treude, M. A. Gerosa, and I. Steinmacher, "Do contributing files provide information about oss newcomers' onboarding barriers?" in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 16–28.

[42] "goreleaser," https://github.com/goreleaser/goreleaser.

[43] "react-navigation," https://github.com/react-navigation/react-navigation.

[44] "hyperswitch," https://github.com/juspay/hyperswitch.

[45] "Squash merge," https://github.blog/changelog/2022-05-11-default-to-pr-titles-for-squash-merge-commit-messages/, accessed: 2024-03-15.

[46] D. S. Cruzes and T. Dyba, "Recommended steps for thematic synthesis in software engineering," in *2011 International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, 2011, pp. 275–284.

[47] K. Lee, K. C. Kang, and J. Lee, "Concepts and guidelines of feature modeling for product line software engineering," in *International Conference on Software Reuse*. Springer, 2002, pp. 62–77.

[48] A. Classen, P. Heymans, and P.-Y. Schobbens, "What's in a feature: A requirements engineering perspective," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2008, pp. 16–30.

[49] K. Chen, W. Zhang, H. Zhao, and H. Mei, "An approach to constructing feature models based on requirements clustering," in *13th IEEE International Conference on Requirements Engineering (RE'05)*. IEEE, 2005, pp. 31–40.

[50] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we

refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2011.

[51] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[52] K. Agrawal, S. Amreen, and A. Mockus, "Commit quality in five high performance computing projects," in *2015 IEEE/ACM 1st International Workshop on Software Engineering for High Performance Computing in Science*. IEEE, 2015, pp. 24–29.

[53] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[54] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *North American Chapter of the Association for Computational Linguistics*, 2019.

[55] J. Ye, X. Chen, N. Xu, C. Zu, Z. Shao, S. Liu, Y. Cui, Z. Zhou, C. Gong, Y. Shen *et al.*, "A comprehensive capability analysis of gpt-3 and gpt-3.5 series models," *arXiv preprint arXiv:2303.10420*, 2023.

[56] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt, "A prompt pattern catalog to enhance prompt engineering with chatgpt," *arXiv preprint arXiv:2302.11382*, 2023.

[57] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," in *International Conference on Learning Representations*, 2017.

[58] M. Hossin and M. N. Sulaiman, "A review on evaluation metrics for data classification evaluations," *International journal of data mining & knowledge management process*, vol. 5, no. 2, p. 1, 2015.

[59] Y. Zhang, M. Qin, K.-J. Stol, M. Zhou, and H. Liu, "How are paid and volunteer open source developers different? a study of the rust project," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3639197

[60] G. E. dos Santos and E. Figueiredo, "Commit classification using natural language processing: Experiments over labeled datasets," in *CIbSE*, 2020, pp. 110–123.

[61] J. Dong, Y. Lou, Q. Zhu, Z. Sun, Z. Li, W. Zhang, and D. Hao, "Fira: fine-grained graph-based code change representation for automated commit message generation," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 970–981.

[62] A. Eliseeva, Y. Sokolov, E. Bogomolov, Y. Golubev, D. Dig, and T. Bryksin, "From commit message generation to history-aware commit message completion," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 723–735.

[63] Y. Zhang, Z. Qiu, K.-J. Stol, W. Zhu, J. Zhu, Y. Tian, and H. Liu, "Automatic commit message generation: A critical review and directions for future work," *IEEE Transactions on Software Engineering*, 2024.