Self-Admitted GenAI Usage in Open-Source Software

Tao Xiao, Youmei Fan, Fabio Calefato, Christoph Treude,
Raula Gaikovina Kula, Hideaki Hata, Sebastian Baltes ✉

Abstract—The widespread adoption of generative AI (GenAI) tools such as GitHub Copilot and ChatGPT is transforming software development. Since generated source code is virtually impossible to distinguish from manually written code, their real-world usage and impact on open-source software (OSS) development remain poorly understood. In this paper, we introduce the concept of *self-admitted GenAI usage*, that is, developers explicitly referring to the use of GenAI tools for content creation in software artifacts. Using this concept as a lens to study how GenAI tools are integrated into OSS projects, we analyze a curated sample of more than 200,000 GitHub repositories, identifying 1,292 such self-admissions across 156 repositories in commit messages, code comments, and project documentation. Using a mixed methods approach, we derive a taxonomy of 32 tasks, 10 content types, and 11 purposes associated with GenAI usage based on 1,292 qualitatively coded mentions. We then analyze 13 documents with policies and usage guidelines for GenAI tools and conduct a developer survey to uncover the ethical, legal, and practical concerns behind them. Our findings reveal that developers actively manage how GenAI is used in their projects, highlighting the need for project-level transparency, attribution, and quality control practices in AI-assisted software development. Finally, we examine the longitudinal impact of GenAI adoption on *code churn* in 151 repositories with self-admitted GenAI usage and find no general increase, contradicting popular narratives on the impact of GenAI on software development.

Index Terms—Software Engineering, Generative Artificial Intelligence, Large Language Models, Software Maintenance and Evolution

1 Introduction

The emergence of generative artificial intelligence (GenAI) tools such as ChatGPT and GitHub Copilot has redefined software development, as documented by literature reviews [1], developer studies [2, 3], and productivity studies [4]. These tools assist developers in writing and reviewing code, refining documentation, and automating various aspects of the software development lifecycle. Although prior research has evaluated the technical capabilities of GenAI tools [5] and surveyed their usability [3], only a few studies have systematically investigated their real-world adoption and usage patterns, for example, by mining GenAI mentions in repositories [6], analyzing AI-generated pull request descriptions [7], or compiling datasets of developer-ChatGPT conversations [8]. One reason is that only the tool vendors have access to fine-grained usage data [4] that allows them to determine which code suggestions were accepted and hence which code was co-authored by GenAI tools. Without additional context, generated code is virtually impossible to distinguish from human-authored code.

As a result, much of what we know about GenAI usage in software development is inferred indirectly, either from vendor-controlled telemetry, laboratory studies, or analyses of repository activity that rely on aggregated metrics rather than direct evidence of GenAI use. This makes it difficult to understand how GenAI tools are actually integrated into collaborative development workflows, how their use is governed in practice, and how their impact should be interpreted at the project level. In real-world settings, developers must decide how much to rely on or revise AI-generated content, and maintainers must determine whether to prohibit, restrict, or encourage GenAI use. Researchers increasingly rely on aggregate metrics such as code churn to assess claims about software quality degradation. Without observable, project-level signals of GenAI usage embedded in software artifacts, these decisions risk being shaped by assumptions, anecdotal evidence, or broad industry narratives rather than by empirical data. Studying explicit references to GenAI usage offers a way to ground these discussions in real development practices, making it possible to examine not only what GenAI is used for, but also how it is acknowledged, regulated, and followed by human action in open-source software (OSS) projects.

These projects, with their collaborative nature and publicly accessible repositories [9], offer a unique context to study the adoption of GenAI tools. Although such tools promise to support OSS projects by automating development tasks, there are also reports of "AI slop" wasting valuable time of maintainers [10] or GenAI-generated contributions leading to more rework [11].

We introduce the concept of **self-admitted GenAI usage**, inspired by the notion of self-admitted technical debt [12]. Just as developers acknowledge technical debt through

Sebastian Baltes is the corresponding author.

- *T. Xiao is with Kyushu University, Japan. E-mail: xiao@ait.kyushu-u.ac.jp*
- *Y. Fan is with Nara Institute of Science and Technology, Japan. E-mail: fan.youmei.fs2@is.naist.jp*
- *F. Calefato is with University of Bari, Italy. E-mail: fabio.calefato@uniba.it*
- *C. Treude is with Singapore Management University, Singapore. E-mail: ctreude@smu.edu.sg*
- *R. G. Kula is with University of Osaka, Japan. E-mail: raula-k@ist.osaka-u.ac.jp*
- *H. Hata is with Shinshu University, Japan. E-mail: hata@shinshu-u.ac.jp*
- *S. Baltes is with Heidelberg University, Germany. E-mail: sebastian.baltes@uni-heidelberg.de*

comments and commits, they sometimes explicitly refer to using GenAI tools. These self-admissions can highlight tasks delegated to GenAI tools, challenges encountered, or changes made due to AI-generated content. Identifying such usage enabled us to explore three research questions (RQs). First, to understand the practical applications of GenAI tools in software development, we examined which *tasks* (e.g., writing test cases) are supported or automated by these tools, which *contents* (e.g., methods in source files) are referenced in GenAI-related mentions, and which *purposes* (e.g., acknowledging GenAI use) such mentions serve. We then investigated the *regulation and recommendation* of GenAI usage, and concluded with an analysis of *code churn* following the first GenAI mention in a project.

RQ1 *For which tasks, contents, and purposes do open-source developers mention GenAI tools?*

One finding that emerged was that project maintainers have begun to establish policies and usage guidelines regarding their use (see Table 7). These regulations provide insights into emerging best practices, ethical considerations, and potential concerns surrounding GenAI adoption. Understanding project-level policies is crucial for the responsible integration of GenAI tools in collaborative software development, leading to our second RQ:

RQ2 *How do open-source projects regulate or recommend the usage of GenAI tools?*

In addition to understanding how developers use GenAI tools and how projects regulate their usage, it is important to understand their impact on software quality and maintenance. The 2024 GitClear report [11], which received considerable attention in the developer community, claimed that increased code churn after GenAI adoption indicates "*downward pressure on code quality.*" The report defines code churn as "*the percentage of lines that are reverted or updated less than two weeks after being authored,*" interpreting such changes as "*either incomplete or erroneous when the author initially wrote, committed, and pushed them*" to the repository. To investigate this claim, we formulate a third RQ:

RQ3 *Does the code churn change after open-source projects start using GenAI tools?*

We conducted a large-scale empirical study of more than 200,000 OSS repositories hosted on GitHub. Our investigation focused on identifying explicit mentions of GenAI tools in various artifacts and analyzing how these mentions relate to development activities. We followed a mixed methods approach, combining a qualitative analysis of GenAI-related mentions with a quantitative examination of code churn over time, resulting in four main contributions:

1) We introduce self-admitted GenAI usage as an empirical lens for studying GenAI adoption in open-source software and curate a dataset of 1,292 self-admitted GenAI usages across 156 GitHub repositories.
2) Using a mixed-methods approach, we derive a taxonomy of GenAI usage comprising 32 development tasks, 10 content types, and 11 purposes, grounded in a qualitative analysis of the identified usages.
3) We empirically analyze how open-source projects govern GenAI usage by examining 13 policies and usage guidelines, contextualized through a developer survey.
4) We assess the longitudinal impact of GenAI adoption on software evolution using a repository-level analysis of code churn in 151 projects, showing that GenAI adoption does not lead to a general increase in churn and that effects are stronger for generation tasks.

2 Methodology

We followed a mixed-methods research design. Our data collection process is visualized in Figure 1. After retrieving instances of self-admitted GenAI usage from open-source GitHub repositories, we conducted a qualitative analysis to answer **RQ1**. Through multiple iterative coding phases, we labeled these instances to classify supported tasks and generated content. Since this qualitative analysis yielded a considerable number of statements that focused on the regulation or recommendation of GenAI practices, we followed up with a closer analysis of these aspects as part of **RQ2**. For **RQ3**, we used self-admitted GenAI usages to approximate the time when the projects started using GenAI tools, to analyze the effect of GenAI usage on code churn using a Regression Discontinuity Design (RDD).

2.1 Repository Sampling

The foundation of our research is a large sample of open-source GitHub repositories. We selected GitHub as our study platform because it is the largest and most widely used open-source hosting service, with over 500 million repositories according to the 2024 Octoverse report [13], making it the most suitable environment for analyzing trends of GenAI usage in open-source software development. Using the GitHub search tool provided by Dabic et al. [14], we selected repositories primarily written in the five most popular programming languages as of the above-mentioned report [13]: Python, JavaScript, TypeScript, Java, and C#. This focus on the most popular languages ensures that our study is both manageable and relevant to the most commonly used development ecosystems. Since **RQ3** aims at a comparison of code churn before and after projects started using GenAI tools, we only selected repositories that: (1) were created before the ChatGPT launch date (30 November 2022) and (2) had at least one commit on or after this date. Moreover, to eliminate duplicates, we excluded forks. Our initial sample of GitHub projects contained 207,062 repositories distributed across Python (77,542), JavaScript (48,500), TypeScript (37,424), Java (25,160), and C# (18,436).

Since our interest is to study "engineered" software projects [15], we applied three additional filtering criteria. First, we excluded repositories not declaring a license or using non-standard licenses (marked as *Other* in the GitHub search tool). For the remaining repositories, we labeled all 38 distinct licenses we found and then removed projects declaring licenses not commonly used for software projects. These licenses included *Creative Commons Attribution 4.0 International*, *Creative Commons Zero v1.0 Universal*, *Creative Commons Attribution Share Alike 4.0 International*, and the *SIL Open Font License 1.1*. Second, we excluded repositories without any release on GitHub, fewer than two contributors, and those marked as *archived*. Third, we filtered the repositories based on an analysis of various descriptive statistics.

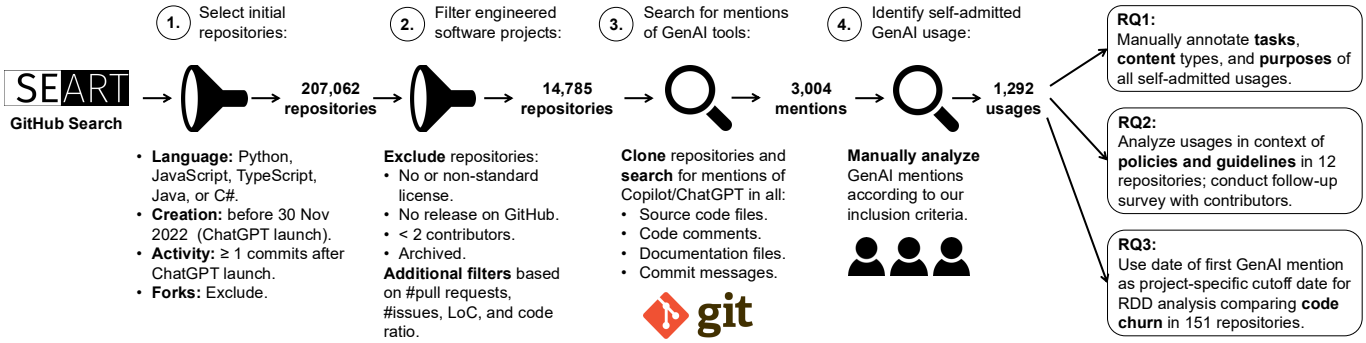

Figure 1. Overview of the data collection process used to answer our three research questions, from the selection (1) and filtering (2) of GitHub repositories to the extraction of GenAI mentions (3) and the identification of self-admitted GenAI usage (4) in these repositories.

Table 1
Descriptive statistics for studied GitHub repositories ($n = 14,785$).

	Min	Mean	Median	Max	Std Dev.
# Issues	9	372	98	171,039	1,880
# Pull Requests	9	454	134	38,421	1,334
# Contributors	2	30	15	475	49
# Lines of Code	1,800	135,223	25,399	47,165,225	734,845
Code Ratio (%)	4.300	78.274	78.721	99.996	12

We analyzed the distribution of central repository properties per programming language. The properties we considered were the number of pull requests, the number of issues, and the repository size measured in lines of code (as provided by the GitHub search tool).

To select engineered software projects with sufficient development data, we excluded repositories in the first quartile (Q_1) for each metric, therefore removing the lowest 25%. Furthermore, we excluded repositories with a code ratio (defined as $lines\_of\_code/(lines\_of\_code + lines\_of\_comments)$) outside the 97% confidence interval. The rationale behind this threshold is that engineered software projects are usually documented using source code comments. Filtering out repositories beyond the 97% confidence interval helps eliminate outliers, that is, repositories with very little code, or codebases dominated by code without comments. A sanity check further confirmed that this ratio serves as a reliable indicator for filtering out non-software or poorly structured projects.

Our final sample of GitHub repositories, obtained in February 2024, contained 14,785 GitHub repositories distributed across Java (5,060), C# (3,544), TypeScript (2,464), Python (1,875), and JavaScript (1,842). Table 1 provides descriptive statistics for the studied GitHub repositories.

2.2 Identifying Self-Admitted GenAI Usages

To identify self-admitted GenAI usage in our filtered sample of GitHub repositories, we retrieved mentions of the two most popular GenAI tools among developers as of the 2023 Stack Overflow Developer Survey [16]. In that survey, ChatGPT was identified as the most popular general AI tool and GitHub Copilot as the most popular AI developer tool. Then, in the second step, we annotated these mentions to identify those related to content generation. We wrote a Python script for the following process:

Table 2
File extensions we included when searching for mentions of GenAI tools in our sample of GitHub repositories.

Type	Language	File Extensions
Code	Python	`.py`, `.ipynb`
Code	Java	`.java`, `.jsp`
Code	TypeScript	`.ts`, `.tsx`, `.vue`
Code	JavaScript	`.js`, `.jsx`, `.vue`, `.mjs`, `.cjs`
Code	C#	`.cs`, `.aspx`, `.cshtml`
Doc.	All	`.md`, `.markdown`, `.mdown`, `.mkdn`, `.mkd`, `.mdwn`, `.mdtxt`, `.mdtext`, `.txt`, `.text`, `.adoc`, `.asciidoc`, `.rst`, `.textile`, `.dbk`

1) **Clone** the default branch of the repository.
2) Search all **source code files** for mentions of ChatGPT or Copilot within **code comments**; save the complete comments along with their language (i.e., the natural language such as English or Chinese).
3) Search all **documentation files** for mentions of ChatGPT or Copilot; save the lines in which the mentions were found, again along with their language.
4) Search all **commit messages** for mentions of ChatGPT or Copilot; save the corresponding commit messages along with their language.

An initial analysis of all files in the repositories revealed a large number of false positive matches, that is, mentions of GenAI tools that were not related to content generation. Therefore, we decided to focus on specific file types when searching for mentions in source code and documentation files. We derived these lists based on common file extensions for the particular programming languages, as well as an analysis of all unique file extensions in which we found mentions during our first data collection run (see Table 2). We further decided to only search mentions of GenAI tools in source code comments, not across the whole source code. This is because, during our initial analysis, we found many false positives that were not related to content generation but to code that calls APIs related to ChatGPT or Copilot. We developed heuristics to reduce these false positives, which we outline in the following.

For identifying mentions of GenAI tools, we employed regular expressions with the following pattern:

```
re.compile(r'(.?)' + llm_tool + r'(.?)',
  re.IGNORECASE | re.DOTALL)
```

where the variable `llm_tool` was assigned the value `r'chat[ \-_]{0,1}gpt'` for ChatGPT and `r'co[ \-_]{0,1}pilot'` for Copilot. These patterns allowed us to capture variations in how these tools were referenced while minimizing false positives. We developed heuristics to further reduce the number of false positives. For example, we noticed that in false positive matches, the mentions of GenAI tools were often surrounded by commas or underscores, e.g., when they were part of URLs for API calls. Our supplementary material contains the full source code that documents our retrieval approach.

Running the above retrieval process on all repositories yielded 3,004 mentions of GenAI tools: 1,572 in commit messages, 397 in source code comments, and 1,035 in documentation files. These mentions were automatically obtained using regular expressions and filtered according to heuristics. However, they still included mentions that were not related to content generation. Thus, we conducted a thorough manual inspection of all mentions to eliminate false positives. This review process was guided by the following instructions:

1) We **include** mentions indicating that content was generated using ChatGPT or Copilot and then copied into the repository. We use a broad definition of "content" that includes not only source code but also comments, translations, and other textual elements.
2) For commits, we also **include** mentions that indicate a modification of previously generated content (e.g., a refactoring or fix for previously generated content) or commits that remove comments indicating the usage of ChatGPT or Copilot to generate content.
3) For documentation files, we **include** mentions that indicate content generation, discuss or regulate the usage of ChatGPT or Copilot in the repositories, and mentions that acknowledge the use of these tools.

To evaluate the coding instructions, two authors independently labeled a sample of mentions, deciding whether they should be included or not. We calculated a sample size of 341 mentions (of 3,004) to achieve estimates with a 95% confidence level and a 5% confidence interval. The inspection resulted in disagreement between the two authors for only 14 cases (4% of the sample). The two authors discussed these cases and tried to reach a consensus. During these discussions, a third author helped resolve each disagreement and suggested possible improvements to the categories. To assess inter-rater reliability, we computed Fleiss' kappa [17] by applying bootstrap resampling methods with 1,000 iterations. The resulting 95% confidence interval was estimated to be (0.87, 0.95), indicating an "*almost perfect*" agreement. Given this high agreement, the first author continued to inspect the remaining mentions alone. In total, we identified 1,292 true-positive mentions of GenAI tools that were aligned with our inclusion criteria. We found true-positive mentions in 156 repositories (11 Python, 12 JavaScript, 37 TypeScript, 47 C#, and 49 Java repositories). We did not merge mentions referring to the same GenAI action (e.g., a commit message and a code comment referring to the same change), as they may indicate distinct usage patterns.

2.3 Data and Code Availability

To enable replication and future research, we have prepared supplementary material that includes the filters we used to sample GitHub repositories, the raw data we retrieved, the manually labeled GenAI tool mentions, the Python scripts we used for data retrieval and analysis, and the questionnaires used for our developer survey. The package is available online [18].

3 Reasons for Mentioning GenAI Tools

To answer **RQ1**, we qualitatively analyzed the GenAI mentions that we collected and curated, categorizing them according to tasks, contents, and purposes.

3.1 Method

We performed an open-coding methodology combined with card sorting to manually analyze our sample of 1,292 GenAI tool mentions (see Section 2.2). The initial coding [19] involved systematically examining and categorizing the data according to emerging conceptual themes. In our study, this involved analyzing individual GenAI tool mentions to identify recurring patterns and assign corresponding codes. Following this initial coding phase, we performed open card sorting to organize low-level codes into higher-level abstract categories, allowing us to recognize broader themes and relationships (focused coding). Three authors of this paper collaborated throughout this process to ensure a rigorous and consistent annotation.

A preliminary analysis identified 1,008 mentions related to Copilot for Pull Requests.[1] Of these, 1,000 instances occurred in a single repository (`pancakeswap/pancake-frontend`) where commit messages directly reused pull request descriptions generated by the tool, while the remaining eight mentions in other repositories explicitly documented that developers used Copilot for Pull Requests to generate pull request descriptions, without reusing those descriptions as commit messages. In addition, we identified one separate case in which contributors were encouraged to use ChatGPT to produce pull request descriptions (P11); this case was not part of the Copilot for Pull Requests group.

Given the overrepresentation of a single, highly repetitive use case, namely the reuse of Copilot-generated pull request descriptions as commit messages, we set aside these 1,008 mentions during the initial round of coding to avoid skewing the development of the coding schema. After establishing a stable set of categories based on the remaining data, we revisited these deferred cases and incorporated them into the analysis.

To build the code book, two authors independently analyzed 284 GenAI mentions. The code book development was guided by the following questions:

- **Task:** *Which task has the GenAI tool supported or automated?* Tasks include, for example, writing a test case, fixing a bug, and refactoring the code base.
- **Content:** *Which content is the GenAI mention referring to?* Content categories include methods in source files, sections in documentation files, and commit messages.
- **Purpose:** *Why has the GenAI tool been mentioned?* Possible purposes include acknowledgment of usage for code generation and regulation of usage within the project.

1. https://githubnext.com/projects/copilot-for-pull-requests

Table 3
GenAI-assisted tasks (**RQ1**): Definition and frequency of categories and codes (n = 284 + 1,000 + 8 = 1,292); the code *PR description* is counted and discussed separately because most of it only occurred in one repository (see Section 3.2.1).

Category	Code	Definition	#
Generation	Code	Understand coding tasks written in natural language and generate corresponding code.	105
	Test data	Create test input/output based on software requirements or the existing codebase.	9
	Comment	Generate code comments that explain the purpose and logic of code blocks.	9
	Test file	Create test files based on software requirements or the existing codebase.	8
	Regex	Craft regular expressions tailored to specific text matching needs.	6
	README	Create README files that provide essential information, e.g., project descriptions.	4
	Dummy text	Produce placeholder text that mimics real content in style, structure, and format.	4
	Test method	Create test methods based on software requirements or the existing codebase.	2
	Code review	Generate reviews that suggest improvements and identify potential issues in code changes.	2
	Commit message	Generate commit messages that summarize code changes.	2
	Tutorial	Produce instructional content on specific topics and step-by-step guidance for projects.	2
	Zod schema	Create Zod schemas in TypeScript and JavaScript for type safety and data validation.	2
	Test class	Create test classes based on software requirements or the existing codebase.	1
	Coding practices	Generate guidelines and best practices for coding in the projects.	1
	Variable	Suggest meaningful variable names that improve code semantics and readability.	1
	Changelog	Compile changelogs that document changes, features, and fixes in new software versions.	1
	Configuration	Generate project-specific configuration files, e.g., performance and security settings.	1
	Text	Generate general text that is not mentioned above.	8
	PR description	*Create explicit PR descriptions to assist understanding the changes in the PRs.*	*1,009*
Translation	Text	Convert text between different languages, e.g., software internationalization (i18n).	49
	Code	Convert code between different programming languages, preserving the original logic and functionality while adapting to the syntax and idiomatic patterns of the target language.	1
Optimization	Code refactoring	Restructure code without altering its functionality, aiming to make the code maintainable.	29
	Code improvement	Improve existing code, mention is accompanied by "improve."	5
Maintenance	Label revision	Analyze, update, and improve text labels, ensuring clarity, accuracy, and consistency.	8
	README revision	Analyze, update, and improve README files, ensuring clarity, accuracy, and consistency.	7
	Document revision	Analyze, update, and improve documents, ensuring clarity, accuracy, and consistency.	4
	Changelog revision	Analyze, update, and improve changelogs, ensuring clarity, accuracy, and consistency.	2
	Prompt refinement	Optimize and clarify the prompts to elicit the most relevant and accurate responses.	1
	Color suggestion	Suggest color schemes for UI/UX design based on best practices and design requirements.	1
	Dependency upgrade	Analyze software dependencies and suggest updates to ensure compatibility and security while minimizing breaking changes.	1
	Version update	Suggest meaningful version numbering for software releases for systematic version control.	1
	Comment revision	Analyze, update, and improve code comments, ensuring clarity, accuracy, and consistency.	1
Other	-	Operate general functionality, like Q&A and blog generation.	12
None	-	There is no specific task for the GenAI tool.	9

Our coding process allowed coders to assign multiple codes per mention. During the iterative refinement of the codes and categories, we observed an interesting pattern in how developers describe their work with GenAI tools. Each mention typically encompasses two distinct but interconnected perspectives: (i) the specific task delegated to the GenAI tool and (ii) the broader development task the human developer aims to accomplish. To capture this pattern, we split the task-related codes into two sub-categories: **GenAI task** and **developer task**. We provide the final code book and code assignment as part of our supplementary material.

Using Fleiss' kappa [17], we assessed the interrater reliability between the two coders. The analysis yielded "substantial" to "almost perfect" agreement levels on task ($k = 0.81 - 0.89$), content ($k = 0.95 - 0.99$), and purpose ($k = 0.79 - 0.92$), according to standard guidelines for interpreting k [20]. Through iterative discussions, the two coders worked to achieve consensus on the categorizations, with a third researcher arbitrating unresolved disagreements and recommending refinements to the categories. The first author then independently checked the 1,008 mentions that we had initially deferred.

3.2 Results

Our analysis of mentions revealed distinct patterns in how developers integrate GenAI tools into their development workflows. In the following, we describe the categories and codes capturing development tasks, content types, and usage purposes, which emerged from our analysis.

3.2.1 GenAI-Assisted Tasks

Our analysis identified 32 distinct task categories in which developers use GenAI tools in their workflows. Table 3 presents these categories along with their definitions and usage frequencies, while Table 4 shows a list of examples of self-admitted GenAI usage. Unsurprisingly, excluding PR-related activities, generation tasks were most common, with code generation being particularly prominent (105 instances). Translation followed with 50 instances, while optimization and maintenance tasks accounted for 34 and 26 instances, respectively.

As mentioned above, we distinguish between developer tasks and GenAI tasks. While Table 3 lists the GenAI tasks, we also want to discuss human tasks related to GenAI tasks. For example, in one commit message that we analyzed (E1) the developer acknowledged that the code was written "*a bit*

Table 4
Examples of self-admitted GenAI usage referenced in this paper.

ID	Artifact	Link
E1	commit	aksio-insurtech/cratis/commit/e97e...
E2	comment	iportalteam/imm.../PortalShape.java#L95
E3	commit	fusion-flux/portal-cubed/commit/0a9d...
E4	commit	vercel/next.js/commit/d210...
E5	commit	pancakeswap/pancake.../commit/4e0f...
E6	doc.	pancakeswap/.../CONTRIBUTING.md
E7	comment	LAMP-Platform/LAMP/.../Format.cs#L171
E8	doc.	ant-des.../github-actions-workflow.en-US.md
E9	doc.	Minecraft-AMS/Carpet-.../README_en.md
E10	comment	BdR76/.../CsvGenerateCode.cs#L733-L735
E11	commit	VelvetToroyashi/Silk/commit/35d9...
E12	commit	deephaven/web-client-ui/commit/d852...
E13	comment	hypar-io/elements/.../Ellipse.cs#L166-L167
E14	comment	dominokit/domino-.../Slider.java#L546-L550
E15	doc.	Anime4000/IFME/.../changelog.txt#L210
E16	commit	dotnet/project-system/commit/3aa2...
E17	commit	ediwang/moonglade/commit/a185...

hasty on previous release" due to "*trust in GitHub Copilot.*" The developer task described in the commit message was *bug fixing*, while the initial task that the GenAI tool supported was *code generation*.

We identified 20 mentions exhibiting this pattern of human actions triggered by an earlier GenAI action. Among them, 13 referred to code that was initially generated using GenAI tools and then changed. The most common follow-up activity was to fix bugs in AI-generated code (9). In other cases, changes were reverted (1), AI-generated comments were deleted (2), or the generated code was commented out (1). For example, one developer commented out code generated by Copilot with the note: "*Note: do not trust GitHub Copilot. It may use z as up axis*" (E2). Another developer reverted a commit that was created with the help of ChatGPT: "*Revert 'ChatGPT' This reverts commit 71e3...*" (E3).

In addition to the 13 human actions that followed AI code generation that we discussed above, we found seven human actions following the generation of configuration and validation files or an unclear role of the GenAI tool. In five cases, developers specified restrictions or exclusions regarding GenAI usage without mentioning a specific task. In two other cases, they removed and rewrote AI-generated configurations or validations. For instance, one pull request superseded another that "*heavily relies on GitHub Copilot (which makes the progress slow and tedious)*" (E4). The developer manually replaced the generated validation schema with a handwritten version.

Recent research has shown that using AI-generated PR descriptions reduces review time and increases PR merge rates [7]. We found that developers reused generated PR descriptions as part of their commit messages. As mentioned in Section 3.1, this approach was very common in one particular project, which contributed 1,000 such mentions to our sample. These generated messages are not limited to this one project—similar patterns appear in popular projects such as `pytorch/pytorch` and `hasura/graphql-engine`. They are added when developers use the PR description as the message for merge or squash commits. This practice represents a form of explicit, self-admitted GenAI usage, embedding a clear marker of AI contribution directly into the software project's official history. Note that, although this single use case contributes a large absolute count, it represents only one entry in our taxonomy of GenAI tasks (Table 3) and does not affect our broader findings. To illustrate this particular use case, we include an excerpt below (E5). Interestingly, the linked contribution guidelines (E6) do not discuss GenAI usage.

```
chore: Remove no used deps (#7349)
<!--
Before opening a pull request, please read the
[contributing guidelines](https://github.com/...]
first
-->
<!--
copilot:all
-->
### <samp>Generated by Copilot at b3683ce</samp>
[...]
```

Of the 1,009 instances of code *PR description* in Table 3, 1,000 originated from a single repository and are discussed separately above. The remaining nine cases include eight instances in which developers explicitly documented the use of Copilot for Pull Requests to generate PR descriptions (Section 3.1), and one instance suggesting that contributors use ChatGPT to produce PR descriptions (P11).

Below, we discuss the most prevalent supported tasks besides generating PR descriptions. As expected, code generation was one of the most common GenAI-supported tasks that we observed. Some self-admitted GenAI usage for code generation was straightforward, such as the following statement that we found in the source code comment documenting a method written in C#: "*This function was written with Chat-GPT*" (E7).

Beyond code generation, developers used GenAI tools to generate other software artifacts, including test data or documentation. Besides generation, GenAI tools were also used to automate code review, for example, as part of GitHub Actions workflows (E8): "*Recently, the team has added ChatGPT to GitHub Actions to perform GenAI-based code review. The specific job can be found in the chatgpt-cr.yml file.*"

After generation, translation emerged as the second most prevalent task in our analysis. Most mentions referred to translation between natural languages. One mention referred to translation between programming languages. An important use case was internationalization, helping developers overcome language barriers (E9): "*Due to my limited proficiency in English, all English document translations are currently provided by ChatGPT, including this sentence.*" The one mention related code translation documented the translation of existing Python code to R (E10): "*The following R code was generated using ChatGPT based on the Python code.*" However, the developer at the same time asked others to support them in improving the code: "*If anyone can refactor it to something more readable or more sensible code, please let me know or submit as a pull request.*"

Code optimization represented the third largest category. Developers not only acknowledged GenAI tools usage but sometimes even thanked the tools in their commit messages (E11): "*Forgot tabs. Thanks, Copilot.*" In addition to code, GenAI tools were also used to improve UI elements (E12): "*I asked chatGPT to help me brainstorm improvements to some of the labels and hint text based on the [...] Interface Guidelines. I then edited them as human to improve them further.*" Interestingly, also in this case, the developer asked other members to review the generated content: "*Review and let me know if you think any are worse or weird.*" This, together with the human corrective actions triggered by GenAI actions we observed, points to the importance of human oversight in GenAI-assisted development.

3.2.2 Content Types

Our analysis identified three main categories of AI-generated content in open-source software projects, organizing ten dis-

Table 5
Content types (**RQ1**): Definition and frequency of categories and codes.

Category	Code	Definition	#
Project metadata	Commit messages	Target commit messages.	1,003
Source files	Whole methods	Target source code files, ranging from entire functions within a file.	47
	Blocks within one source code file	Target source code files, spanning multiple blocks within a single source code file.	45
	One block within one source code file	Target source code files, spanning one block within a single source code file.	39
	Blocks within multiple source code files	Target source code files, spanning multiple source code files.	21
	Whole files	Target source code files, ranging across the entire file.	12
	Whole classes	Target source code files, ranging across the entire class in the file.	12
Project assets	Documentation files	Target documentation files, which include technical documents in software projects.	106
	Configuration files	Target configuration files, which define the operational parameters and settings.	24
	Resource files	Target resource files, e.g., images, localization strings, and other binary data.	5

Table 6
Purposes of GenAI usage (**RQ1**): Definition and frequency of categories and codes.

Category	Code	Definition	#
Documentation and Acknowledgment	Acknowledgement of usage	Recognizing and documenting the use of GenAI tools within the codebase.	1,236
	Acknowledge that the bug fix is related to AI-generated code	Noting in the documentation or comments that a particular bug fix pertains to issues originating from AI-generated code.	13
	Removal of Copilot comment	Deletion of comments initially suggested by GenAI tools that are no longer relevant or correct.	2
Guidance and Best Practices	Set example	Providing usage examples to illustrate how GenAI tools can be used.	25
	Exclusion of usage within the project	Documenting rules or guidelines on how GenAI tools should not be used within the project to maintain consistency and quality.	18
	Regulation of usage within the project	Documenting rules or guidelines on how GenAI tools should be used within the project to maintain consistency and quality.	10
Quality Assurance	Look for refactoring/reviewing/improving	Marking sections of content generated by GenAI tools that need to be refactored, reviewed, or improved for better performance, readability, or maintainability.	11
	Warning	Issuing cautions in the code, e.g., vulnerabilities, deprecated methods, or unstable features.	10
	TODO	Indicating LLM tasks that need to be completed in the future.	2
GenAI Limitations	Blame Copilot	Specifically attributing errors or suboptimal code to suggestions made by a GenAI tool.	3
	Revert	Noting the need to undo LLM changes that have led to issues or did not perform as expected.	1

tinct codes (see Table 5). Although, as mentioned before, commit messages related to Copilot PR activities dominated our dataset with over 1,000 mentions from a single repository, examining the remaining data revealed important patterns. Developers frequently use GenAI tools to modify source files (176 mentions). However, other file types, such as documentation and configuration files, were also targeted (135 mentions).

When working with source files, developers usually focus on smaller elements such as individual functions or code blocks instead of complete files. For example, we found blocks of code implementing geometrical transformation, for which the developers added a comment indicating ChatGPT usage. Interestingly, they even documented the prompt in the source code comment (E13): "*Code generated from chatgpt with the following prompt:[...].*" In another example, a developer added an interface for UI elements, mentioning ChatGPT as the author in the comment (E14): "*A functional interface to handle slider slide events. [...] @author ChatGPT.*"

For project assets other than source code, GenAI was also used to generate changelogs (E15): "*Note: This changelog is improved by OpenAI ChatGPT from my broken English input.*" Another use case we observed was adding comments explaining options in a configuration file (E16): "*These strings were provided by GitHub Copilot. I checked the first few, and they were correct.*"

Self-admission of GenAI usage, as illustrated by the previously mentioned comment for the generated C# method (E13), appeared consistently across projects. Besides generation, code refactoring is another use case for mentioning GenAI usage: "*code refact by github copilot*" (E17).

Quality assurance emerged as another key purpose, with developers often requesting peer review of AI-generated content. More examples of this can be found in Section 3.2.1.

Summary RQ1:

For the 1,292 GenAI mentions we analyzed, developers mainly used GenAI tools for code generation, natural language translation, and code refactoring. Source code and documentation files were the dominant generation targets. Acknowledgment of GenAI usage was a common purpose, sometimes combined with warnings about possible negative implications. Another important purpose was regulation (see **RQ2**). Our analysis revealed patterns of corrective actions following code generation. Our findings show that GenAI tools are actively used in open-source software and that developers are working on guiding their usage.

3.2.3 Purposes of GenAI Usage

Our analysis identified 11 different purposes for GenAI mentions in software projects, grouped into four main categories (see Table 6). Documentation and acknowledgment of GenAI usage emerged as the most frequent purpose. This manifested itself in several ways, such as offering guidance (53 mentions), flagging areas needing attention (23 mentions), and addressing GenAI limitations (4 mentions).

4 Existing Guidelines for GenAI Usage

One topic that emerged while answering **RQ1** is that some open-source projects have specific policies and guidelines around GenAI usage. Therefore, as part of **RQ2**, we investigated how projects prohibit, restrict, or support the usage of GenAI tools. In addition to analyzing the policies and guidelines, we conducted a survey with open-source developers to understand their views on GenAI regulation.

4.1 Method

Using our sample of GenAI mentions, we found 28 mentions related to policies and usage guidelines around GenAI tool usage. We grouped them into three groups: (1) prohibitive, (2) restrictive, and (3) supportive usage. Table 7 presents detailed examples drawn from 13 documentation files and commit messages in 12 GitHub repositories, where the last column indicates the number of mentions identified in the software artifact.

First, we closely examined these policies and usage guidelines to understand how exactly projects regulate GenAI usage. Second, we conducted a developer survey that included excerpts from the policies and guidelines we found. The primary goals of the survey were to: (1) collect developer perceptions on the need for GenAI tool guidance (e.g., documenting prompts or annotating generated content) and understand the actions taken on this content before integration or publication; and (2) investigate the rationale behind policies and usage guidelines. To investigate the second part, we asked participants if they contributed to one of the repositories from which we extracted policies and guidelines (see Table 7) and then showed the corresponding guidelines, asking them to elaborate on the rationale behind them. In this way, we received feedback on P1 and P8. For developers who did not identify as contributors to any of the repositories, we showed P3, P4, and P7, deliberately selected as examples of prohibitive and restrictive usage, and asked for their feedback on these policies. Supportive policies were not included because they are less likely to raise questions about regulation or project governance. To keep the survey focused and concise, we prioritized policies that reflect the tensions and risks surrounding GenAI usage in OSS projects. Future work could complement this by investigating developer perceptions of supportive policies and the conditions under which projects actively promote GenAI use.

Our target population was the contributors of the 12 GitHub repositories in our dataset that contained explicit GenAI usage policies (see Table 7). Of these, seven had the GitHub Discussions feature enabled, which we used as our primary outreach channel to gather direct developer feedback. For the remaining five repositories where this feature was not available, as well as two repositories where our discussion posts received no response, we identified contributors on GitHub and then, to comply with GitHub's terms of service, looked for contact details outside of GitHub (e.g., personal websites or social media profiles). We were able to determine the email addresses of 30 contributors. In total, we received eight survey responses, which we analyzed using a combination of open coding and card sorting. Informed consent was obtained. The survey questionnaire is available in our supplementary material.

4.2 Results

In the following, we present the results of our analysis of policies and usage guidelines and our developer survey.

4.2.1 Policies and Usage Guidelines of GenAI Tools

As mentioned above, Table 7 lists 13 software artifacts from 12 GitHub repositories that presented policies or usage guidelines for GenAI usage. We classified them into prohibitive, restrictive, and supportive policies.

Prohibitive: Policies P1–P6 illustrate community decisions that exclude GenAI usage in the projects. Maintainers of `jqwik-team/jqwik` raised concerns related to the copyright situation around GenAI-generated content (P1 and P2). Similarly, maintainers of `shoelace-style/shoelace` addressed ethical and licensing issues arising from the inclusion of GenAI-generated code (P3). Regarding code reviews, maintainers of `katsutedev/mal4j` (P5) and `shred/acme4j` (P6) explicitly stated that contributions generated by GenAI are not acceptable. The project `turms-im/turms` (P4) discouraged the use of GenAI-generated responses in discussions, citing concerns over the lack of critical thinking and responsibility. In addition, the maintainers proposed to incorporate indicators for identifying possible GenAI usage and suggested tool support, for example, a ChatGPT detector published on HuggingFace [21]. These regulations demonstrate how open-source communities are beginning to establish boundaries and safeguards regarding GenAI tools in collaborative open-source software development.

Restrictive: Policies P7–P10 restrict the use of GenAI tools in software development workflows without completely banning it. For example, the maintainers of `graycoreio/daffodil` (P7) require developers to disclose any use of GenAI as a prerequisite for submitting a pull request. Meanwhile, maintainers of `owasp/wrong-secrets`, `sitespeedio/sitespeed.io`, and `theokanning/openai-java` (P8, P9, and P10) advised caution when using GenAI, warning of potential inaccuracies and security risks, such as inadvertent secret leakage due to the fact that tool vendors use prompts for reinforcement learning.

Supportive: Policies P11–P13 outline supportive guidelines for the use of GenAI tools. The maintainers of `avaloniaui/avalonia` (P11) encouraged the use of GenAI to help draft pull request descriptions to support the code review process. In `hardisgroupcom/sfdx-hardis` (P12), maintainers recommended using GenAI for Q&A support, particularly for troubleshooting deployment issues. The project `spring-projects/spring-cli` (P13) promoted the use of GenAI to generate `README.md` files and has even developed GenAI tooling to support automated documentation rewriting.

Overall, these policies and usage guidelines reflect a growing awareness of both the opportunities and risks of GenAI tools in open-source software projects and the willingness of the maintainers to guide their usage. An interesting direction for future work is extending the analysis to cover more policies and guidelines and correlating them with other project characteristics. For example, one could hypothesize that GPL-licensed projects are more likely to have restrictive regulations.

4.2.2 Developer Survey on GenAI Governance

Based on the analysis of the before-mentioned policies and usage guidelines, we designed ten questions regarding (i) the necessity of GenAI tool guidance; (ii) the necessity of documenting prompts and their generated contents; (iii) actions on generated content before integrating; and (iv) the rationale behind policies and usage guidelines of real-world GenAI tools. In the following, we will use D to refer to individual developers who participated in our survey.

General GenAI Usage Guidance: Five developers highlighted the necessity of regulating the usage of GenAI tools in software projects. They cited concerns such as copyright issues, license violations, and ethical considerations as key reasons for establishing guidelines. For instance, respondent D_3 remarked that "*using GenAI is a highly ethical question. With a regulation, one can take a stance.*" The motivation for guidelines and regulations varied, with D_6 stating that "*it [GenAI tool usage] is convenient, but can be detrimental to the codebase if used fully unregulated,*" while D_2 noted that "*it largely depends on the risk appetite and sensitivity of the project/organization.*" Interestingly, D_5 expressed a negative view of regulating GenAI tool usage, arguing that it could hinder productivity. They stated: "*No, instead, humanity must fully harness the potential of AI to unleash productivity. Regulating its usage too tightly would hinder innovation and slow down progress. Instead of imposing external regulations on AI usage, human society should develop autogenous forms of regulation, driven by shared values, ethical guidelines, and adaptive practices.*" When asked about specific aspects of software projects that should be regulated, developers expressed concerns primarily about unlicensed training datasets and potential licensing conflicts associated with AI-generated code. For example, D_1 observed "*It's unclear what the license of AI-generated code is. AIs have been trained on all kinds of licenses, so what license is the generated code?*"

Table 7
Policies and guidelines for GenAI usage in software projects (**RQ2**): **pro**hibitive, **res**trictive, and **sup**portive policies (P); the last column (#) shows the number of mentions in the corresponding documentation file or commit message.

Goal	ID	Repository	Excerpt	#
pro	P1	jqwik-team/ jqwik	"*jqwik Contributor Agreement - You **have authored 100% of the contents of your contribution**. Among other things that means that you **have not used GitHub Copilot or a similar LLM** to create all or parts of your contribution! The reason is that the copyright consequences of training an LLM with mostly public code repositories have not been clarified.*" (CONTRIBUTING.md)	1
	P2	jqwik-team/ jqwik	"*Including GH Copilot clause in CONTRIBUTING.md*" (commit/6cdc...)	1
	P3	shoelace-style/ shoelace	"***AI-generated Code** As an open-source maintainer, I respectfully ask that you **refrain from using AI-generated code** when contributing to this project. This includes code generated by tools such as GitHub Copilot, even if you make alterations to it afterwards. While some of Copilot's features are indeed convenient, **the ethics surrounding which codebases the AI has been trained on and their corresponding software licenses remain very questionable and have yet to be tested in a legal context**. I realize that one cannot reasonably enforce this any more than one can enforce not copying licensed code from other codebases, nor do I wish to expend energy policing contributors. I would, however, like to **avoid all ethical and legal challenges that result from using AI-generated code**. As such, I respectfully ask that you refrain from using such tools when contributing to this project. At this time, I will **not knowingly accept any code that has been generated in such a manner**.*" (contributing.md)	1
	P4	turms-im/turms	"***Can Responses Generated by a Model Similar to ChatGPT be Used for Discussion? ChatGPT is an excellent memorizer, but its analysis of various technical solutions is quite naive**. Engaging in discussions with ChatGPT responses only **reflects a lack of critical thinking and a lack of responsibility towards the projects**. Therefore, whether we should answer such responses depends on the proportion of responses after removing ChatGPT answers. [...] **How to Identify Responses Generated by a Model Similar to ChatGPT** [...]*" (index.md)	9
	P5	katsutedev/ mal4j	"*PLEASE READ BEFORE SUBMITTING PR **Does not include AI generated code**, such as GitHub Copilot or ChatGPT.*" (pull_request_template.md)	1
	P6	shred/acme4j	"*Acceptance Criteria These criteria must be met for a successful pull request: . . . You confirm that you **did not use AI based code generators** like GitHub Copilot for **your contribution**.*" (CONTRIBUTING.md)	1
res	P7	graycoreio/ daffodil	"*Submitting a Pull Request (PR) Before you submit your Pull Request (PR) consider the following guidelines: Please note: If your PR contains **code that was generated by an AI tool such as ChatGPT or Copilot**, **you must disclose this** in the description of your PR.*" (CONTRIBUTING.md)	1
	P8	owasp/ wrongsecrets	"***Why you should be careful with AI (or ML) and secrets** Any AI/ML solution that relies on **your input** might use that input for further improvement. This is sometimes referred to as 'Reinforcement learning from human feedback' . . . This means that when you use those and give them feedback or agree on sending them data to be more effective in helping you, then this data resides with them and might be queryable by others.*" (challenge32_reason.adoc)	1
	P9	sitespeedio/ sitespeed.io	"*We **don't use ChatGPT to code** sitespeed.io but we **prompt it to write a blog post** about sitespeed.io as it was Steve Jobs writing it and it turned out quite good.*" (CONTRIBUTING.md)	4
	P10	theokanning/ openai-java	"*How to Contribute Add POJOs to API library I usually have **ChatGPT write them for me by copying and pasting** from the OpenAI API reference (example chat [link]), but double-check everything because **Chat always makes mistakes**, especially around adding '@JsonProperty' annotations.*" (CONTRIBUTING.md)	1
sup	P11	avaloniaui/ avalonia	"*Please provide a good description of the PR. Not doing so will delay review of the PR at a minimum, or may cause it to be closed. **If English isn't your first language, consider using ChatGPT or another tool to write the description.** If you're looking for a good example of a PR description see [PR link] for example.*" (CONTRIBUTING.md)	1
	P12	hardisgroupcom/ sfdx-hardis	"*Learn how to solve deployments errors that can happen during merge requests [...] SOS, I'm lost [...] - Call your release manager, he/she's here to help you! **Google / ChatGPT / Bard** the issue*" (salesforce-ci-cd-solve-deployment-errors.md)	1
	P13	spring-projects/ spring-cli	"*Large Language Models such at OpenAI's ChatGPT offer a powerful solution for generating code using AI. ChatGPT is trained not only on Java code but also on various projects within the Spring open-source ecosystem. Using a simple command, you can describe the desired functionality, and ChatGPT **generates a comprehensive 'README.md' file** that provides step-by-step instructions to achieve your goal ... For further improvements and accuracy, you can get **ChatGPT to rewrite the description by using the –rewrite option**: The 'ai add' command lets you add code to your project generated by using OpenAI's ChatGPT.*" (ai-guide.adoc)	5

Documenting Prompts and Generated Content: $D_{5,6,8}$ emphasize the importance of documenting prompts and generated content to ensure accountability in software projects. They suggest two methods to achieve this: (1) associating prompts with their functionality and sharing them under a *CC BY 4.0* license, and (2) embedding prompts as code comments or in project documentation, supplemented by shared conversations, e.g., via ChatGPT links. Despite some developers considering prompt documentation unnecessary, the majority agreed that it is valuable to understand the extent of GenAI's contributions to a project. This documentation is essential for assessing which code is potentially affected by copyright and licensing issues; it might also prove useful for later maintenance activities.

Actions on Generated Content Before Integration: Developers are, compared to manually written code, more likely to perform code reviews and license compliance checks on AI-generated content before integrating it into their projects. Three developers highlighted these practices as crucial steps to ensure the quality and compliance of GenAI-based contributions. Additionally, some developers indicated that they rely on automated tools, e.g., code quality checks or automated testing, to evaluate generated content. One developer noted the importance of adding comments to the document generation context. Interestingly, D_2 explicitly stated that no additional actions are necessary, explaining that "*all content in the PR will be subjected to rigorous review and testing regardless.*" This response reflects the perspective that standard testing and code review are sufficient to ensure the quality of both AI-generated and manually created content.

Project-Specific GenAI Usage Guidance: The feedback we received from open-source developers regarding GenAI tool guidance reflects a combination of ethical, legal, and practical considerations. For example, the project owner of `jqwik-team/jqwik` (D_3) described their decision to disallow the use of GenAI tools as an "*ethical decision due to all its collateral damages.*" This statement suggests a strong position against the potential implications of accepting AI-generated contributions, with a particular focus on copyright and ethics. The regulation in the accompanying contributor agreement (P1) explicitly prohibits contributions created using GenAI tools, citing the unresolved legal implications of training AI models on public code repositories. Similarly, the project owner of `owasp/wrongsecrets` (D_2) focuses on the ethical risks of using GenAI when describing the rationale behind guideline P8. They highlight the importance of vigilance when handling sensitive data, particularly in the `wrongsecrets` project. They reported: "*This is a recommendation meant for people using WrongSecrets, and it applies more broadly than WrongSecrets or even OWASP itself. You should be conscious about what data you share, and be vigilant that you don't input sensitive data, since tenant*

boundaries are murky at best." This raises a broader concern about how user input may be stored or reused by GenAI systems. The associated recommendation emphasizes that GenAI tools often rely on reinforcement learning, which could expose sensitive data to unintended parties.

Guideline P7, which requires the disclosure of AI usage in pull requests, received support from three developers (D_4, D_5, D_6). D_4 emphasized that disclosure depends on whether a "*key idea*" was generated by AI, while D_5 highlighted the importance of transparency for license compliance. D_6 added that disclosing the percentage of GenAI involvement in contributions could reduce the likelihood of generated "*noise PRs*" and improve code review efficiency. This reflects a growing recognition of the need for transparency in collaborative software development, where understanding the role of AI in contributions can improve accountability and ensure compliance.

Opinions diverge considerably for P3, which prohibits AI-generated code. D_4 opposed such restrictions, viewing them as unnecessary limitations that could stifle productivity and innovation. D_6 criticized the policy as being overly cautious, suggesting that asking contributors to "*disclose percentage*" of generated content is sufficient. D_7 supported the regulation, noting its alignment with their own concerns about the ethical and legal implications of using GenAI tools. D_5, pointed to "*ethical and legal ambiguities related to AI-generated code*", describing them as "*maintainer's main concerns.*" They specifically highlighted that "*AI models are likely trained on large datasets that include open-source codebases with various licensing terms.*" D_4 and D_6's feedback on P4, which regulates the use of AI in community discussions, emphasizes concerns about the high false positive rates of AI identification tools and warns against deferring critical decisions to automation. D_5 argued that while LLMs are suitable for repetitive tasks and generic translations, they lack the creativity needed for meaningful contributions. This aligns with the cautionary tone of the regulation, which warns about overreliance on AI-generated content.

Summary RQ2:

We found 13 policies and guidelines on GenAI usage in open-source software projects, including strict policies prohibiting GenAI usage, policies requiring attribution, but also guidelines encouraging contributors to use GenAI, for example, for translating natural language text. The results of our developer survey reflect the tension between anticipated productivity gains of GenAI tools and legal and ethical implications of their usage.

5 Impact of GenAI Usage on Code Churn

The goal of **RQ3** was to examine the impact of GenAI usage on open-source software projects.

5.1 Method

The GenAI mentions we identified as part of **RQ1** allow us to approximate the point in time when the 156 open-source projects in our sample started using the GenAI tools. We included 151 repositories with true positive GenAI mentions that did not prohibit the use of GenAI tools. That is, we excluded five repositories prohibiting GenAI usage according to our **RQ2** analysis. Hence, we use self-admitted GenAI usage as a proxy for GenAI tool adoption.

To assess the impact of GenAI tool usage, we calculated the code churn, as defined in the GitClear report (see Section 1), before and after the first self-admitted GenAI usage. Code churn is a widely recognized indicator of software maintainability [22]. Churn rates can signal challenges such as higher defect density [23]; files affected by technical debt tend to be more defect-prone [24]. Code churn is particularly relevant for understanding the maintainability of LLM-generated source code, which might introduce redundancies or bugs that result in changes soon after adding generated code.

The specific notion of code churn introduced by GitClear measures whether added or modified code is updated again within 14 days of the initial commit. Therefore, it serves as an indicator of the maturity of the code that developers add or modify. The 2024 GitClear report [11] suggested that code churn has been continuously increasing since the adoption of GenAI tools in software projects.

To answer **RQ3**, we selected repositories with at least one self-admitted GenAI usage. The first recorded GenAI mentions in the commit history served as the adoption point (t_{mention}). Code churn was analyzed across two timeframes:

- pre-GenAI adoption: The 360 days preceding t_{mention}
- post-GenAI adoption: The 360 days following t_{mention}

In the following, the term *churned lines* refers to the number of lines that were added or modified within the defined timeframes (pre-GenAI adoption or post-GenAI adoption). For each commit, we track the changes introduced with the commit and whether those changes were modified again within a 14-day window.

Specifically, we defined code churn as the percentage of lines that are reverted or updated within 14 days after they were initially added or modified. We added a second definition that focuses on churned files instead of lines to gain a more comprehensive understanding of the impact of GenAI adoption on the selected repositories.

Line-based churn measures the percentage of lines (1) that the commit added or modified and (2) that were changed again within 14 days after the commit. This metric captures the frequency with which individual lines are churned, indicating potential code maintainability challenges. Line-based churn ch_L for a commit c is defined as:

$$ch_L(c) = \frac{\text{\#lines changed again within 14 days}}{\text{total \#lines changed by } c}$$

File-based churn measures the percentage of files (1) that the commit added or modified and (2) that were changed again within 14 days after the commit. For this definition, we consider all changes to the files, regardless of the specific lines that were changed. File-based churn ch_F for a commit c is defined as:

$$ch_F(c) = \frac{\text{\#files changed again within 14 days}}{\text{total \#files changed by } c}$$

For each granularity level (ch_L, ch_F), to understand trends, we report changes in the average code churn over multiple commits. We calculated:

1) The average churn per repository, comparing pre- and post-GenAI adoption using *Wilcoxon signed-rank test* [25]. We applied the *Wilcoxon Z statistic* r to measure the paired effect and interpreted the effect size as follows [26]: $|r| < 0.1$ as *negligible*, $0.1 \leq |r| < 0.3$ as *small*, $0.3 \leq |r| < 0.5$ as *medium*, and $0.5 \leq |r|$ as *large*;
2) The average churn over all commits in all repositories, comparing pre- and post-GenAI adoption using *Mann-Whitney test* [27]. We applied the Cliff δ [28] to measure the independent effect and interpreted the effect size as follows [29]: $|\delta| < 0.147$ as *negligible*, $0.147 \leq |\delta| < 0.33$ as *small*, $0.33 \leq |\delta| < 0.474$ as *medium*, and $0.474 \leq |\delta|$ as *large*.

We further used a *Regression Discontinuity Design* (RDD) [30, 31] to study the impact of GenAI adoption on code churn. RDD is a quasi-experimental method evaluating the impact of an intervention by comparing outcome data points before and after a cutoff point (in our case, the first GenAI mention in a repository). This method has been applied in software

Table 8
Effect size of significant code churn differences pre- vs. post-GenAI adoption, measured using Wilcoxon signed-rank test ($\alpha = 0.05$) and Wilcoxon Z statistic r ($n = 151$).

Churn Type	Effect size	#Significant	Sum sig.	Not sig.
File-based	negligible	5 | 10	15	19
	small	14 | 30	44	8
	medium	9 | 23	32	1
	large	2 | 26	28	4
	sum	30 | 89	119	32
Line-based	negligible	5 | 5	10	22
	small	13 | 33	46	7
	medium	7 | 24	31	0
	large	8 | 25	33	2
	sum	33 | 87	120	31

Each value corresponds to the number of repositories exhibiting an increasing trend or a decreasing trend, respectively.

Table 9
Distribution of code churn patterns based on RDD ($\alpha = 0.05$, $n = 149$).

Churn Type	Trend	Slope		Sum sig.	Not sig.
		#Positive	#Negative		
File-based	Upward	3 (11.5%)	12 (46.2%)	26	123
	Downward	4 (15.4%)	7 (26.9%)		
Line-based	Upward	5 (16.7%)	10 (33.3%)	30	119
	Downward	3 (10.0%)	12 (40.0%)		

engineering before, for example, to assess the impact of introducing code review bots and GitHub Actions to software repositories [32, 33].

We categorized the patterns that emerged from the RDD analysis based on two characteristics: (1) *trend* and (2) *slope*. The *trend* characteristic captures whether code churn increases or decreases from the pre- to the post-GenAI adoption period. The *slope* characteristic captures whether and how the rate of change in the trend line shifts at the point of GenAI adoption. The Ordinary Least Squares (OLS) model used in the RDD analysis requires a minimum number of observations to estimate its four parameters (intercept, time trend, treatment effect, and interaction) while maintaining positive degrees of freedom [31]. To meet this requirement, we aggregated commit-level churn values into weekly data points and fit the RDD model to the resulting weekly time series. For each repository, we required a minimum time span of five weeks, each containing at least one commit, as well as at least one such week on either side of the GenAI adoption point, ensuring that both pre- and post-adoption trends could be estimated. After applying these thresholds, we excluded two repositories with insufficient data.

For the 149 repositories included, we identified four patterns, that we describe in the following:

a. **Upward trend with positive slope change:** This pattern shows code churn increasing after GenAI adoption with an increasing rate of change, which means that the churn grows progressively faster.
b. **Upward trend with negative slope change:** Here, the code churn increases after GenAI adoption, but the rate of increase decelerates over time, suggesting that the initial churn increase gradually stabilizes over time.
c. **Downward trend with positive slope change:** In this pattern, churn decreases after GenAI adoption, but the change rate slows down over time.
d. **Downward trend with negative slope change:** This pattern exhibits decreasing churn after GenAI adoption with an accelerating rate of decline, which means that the churn reduction progressively increases.

Table 10
Significant RDD churn discontinuities by task category (Generation/Optimization/Maintenance).

Churn Type	Task	#Significant	Sum sig.	Not sig.
File-based	Generation	10 | 6	16	68
	Optimization	1 | 1	2	12
	Maintenance	0 | 1	1	2
Line-based	Generation	9 | 6	15	69
	Optimization	2 | 2	4	10
	Maintenance	0 | 2	2	1

Each value corresponds to the number of repositories exhibiting an increasing trend or a decreasing trend, respectively.

These patterns (with examples illustrated in Figure 2) provide a useful framework for analyzing how code churn metrics change after GenAI adoption in different project contexts. To assess the robustness and causal interpretability of the RDD-based patterns, we conducted robustness checks. First, in addition to each repository's project-specific adoption point (i.e., the first commit explicitly reporting GenAI assistance), we estimated parallel RDD models using two global cutoff dates corresponding to major GenAI releases: GitHub Copilot (June 2021) and ChatGPT (November 2022). Second, we assessed robustness to bandwidth selection by estimating the RDD under multiple symmetric temporal windows around the cutoff (±90, ±180, and ±360 days), while keeping all other model specifications constant. Third, we conducted placebo tests by shifting each repository's cutoff date forward and backward in time and re-estimating the same models. Across all robustness checks, the results remained stable, indicating that our main RDD findings do not depend on arbitrary cutoff choices or temporal window specifications.

Finally, to examine heterogeneity across different forms of GenAI usage, we further disaggregated the RDD analysis by the GenAI-assisted tasks identified in **RQ1** (e.g., generation, optimization, and maintenance). We additionally performed a manual verification of every subcode assigned to each task in Table 3, including those subcodes likely to influence code churn. The full set of GenAI generation task categories includes code, test data, comment, test file, regex, test method, Zod schema, test class, and configuration. GenAI optimization task categories encompass all subcodes of code refactoring and code improvement. GenAI maintenance task categories include comment revision, color suggestions, dependency upgrades, and version updates.

5.2 Results

Table 8 illustrates the variations in code churn of the studied repositories. Of the 151 repositories with self-admitted GenAI usage, 119 had a significant difference in file-based churn, and 120 had a significant difference in line-based churn ($p < 0.05$). Eleven repositories had an increasing file-based churn with a medium-to-large effect size, and 15 had an increasing line-based churn with a medium-to-large effect size. A decreasing churn was more common: 49 repositories had a decreasing file-based churn with a medium-to-large effect size, and 49 repositories had a decreasing line-based churn with a medium-to-large effect size.

Besides the average code churn per repository pre- and post-GenAI adoption, we also compared the average code churn over all commits in our dataset pre- and post-GenAI adoption. The average file-based code churn decreased from 0.17 to 0.06 with a significant difference ($p < 0.05$) and a medium effect ($|\delta| = 0.42$), the average line-based churn decreased from 0.68 to 0.50 with a significant difference ($p < 0.05$) and a negligible effect ($|\delta| = 0.09$). These results are contrary to

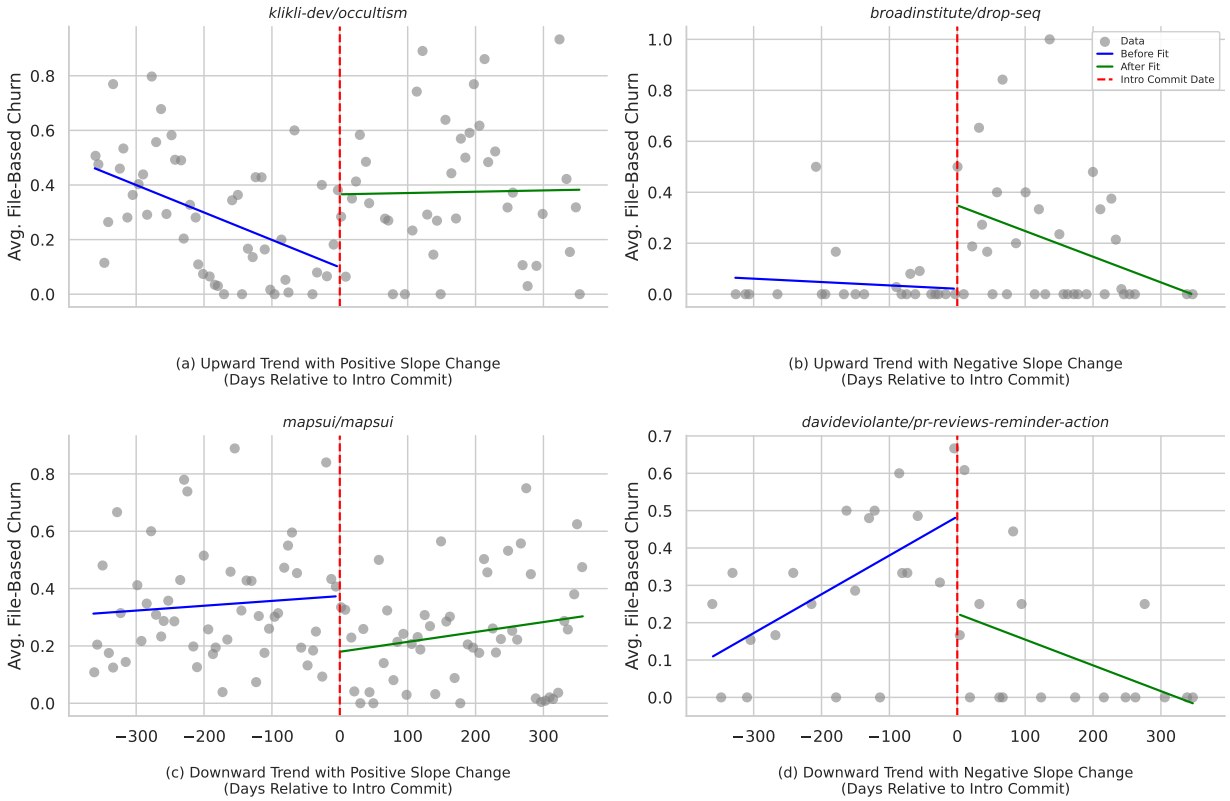

Figure 2. One representative repository for each of the four RDD patterns of file-based churn, showing average file-based churn per week (gray), pre-adoption fit (blue), post-adoption fit (green), and the first GenAI mention (red dashed line).

our expectations because the GitClear report was very bold in claiming that code churn increased for the projects they studied, suggesting a "*downward pressure on code quality*" [11]. While we observed that some repositories have an increasing trend in code churn, both the overall trend and the trend in many individual repositories point to a decreasing code churn over time. Therefore, with our data and methodology, we cannot confirm this claim.

Table 9 summarizes the results of our RDD analysis. We observed that only 26 (file-based) and 30 (line-based) repositories showed significant code churn trends ($p < 0.05$). For file-based churn, an overall upward trend with a negative slope after the cutoff date was most common (12 repositories). For line-based churn, an overall downward trend with a negative slope was most common (12 repositories). However, there were almost as many repositories (10) with an overall downward trend, but a positive slope after the cutoff date. We found 15 repositories with a significant upward trend in file-based churn and 15 with a significant upward trend in line-based churn. However, most of them had a negative slope. In addition, 11 projects had a significant downward trend in file-based churn and 15 had a significant downward trend in line-based churn. We cannot conclude that there is a general trend toward increasing code churn.

To deepen our understanding of which GenAI-assisted tasks (**RQ1**) are most strongly associated with churn changes, we have further disaggregated the RDD results by task (see Table 10). Across all models, significant discontinuities are concentrated in the generation category, including the generation of code, test data, test methods, regular expressions, and comments. In contrast, optimization and maintenance tasks do not show consistent patterns. In summary, this task-level analysis reveals that GenAI-assisted generation tasks appear more rework-prone than other forms of assistance.

Summary RQ3:

Our results revealed that for most of the repositories analyzed, there was no significant change in code churn after GenAI adoption. We did find 15 repositories with an overall upward trend in line-based code churn after the first GenAI mention. However, for most of them, the slope was negative. In addition, we also found 15 repositories with a downward trend. Among the GenAI tasks identified in **RQ1**, generation tasks show a stronger impact on code churn than other tasks. These results indicate that more research is required to understand why certain projects or GenAI tasks are affected and others are not, and how higher (or lower) code churn relates to the long-term maintainability of software projects.

6 Discussion

In this section, we discuss and contextualize the results of our three research questions and summarize the implications for software developers and researchers.

6.1 RQ1: Reasons for Mentioning GenAI Tools

By focusing on self-admitted GenAI usage, that is, explicit mentions of GenAI tools in source code comments, commit messages, and documentation files, we gained a thorough understanding of how and why developers acknowledge GenAI tools in open-source projects. One central contribution of this paper is our taxonomy of assisted tasks, content types, and usage purposes (see Tables 3, 5, and 6).

Our taxonomy provides a multi-dimensional characterization (task, content type, purpose), where prior studies

only categorize tasks assigned to GenAI tools. Our analysis revealed that developers primarily use GenAI tools for code generation, natural language translation, and code refactoring. Tufano et al. [6] explored mentions of ChatGPT in commits, PRs, and issues. They identified that the three most common task categories were feature implementation and enhancement, software quality, and documentation. In our study, we present a more fine-grained and comprehensive categorization of tasks automated by both ChatGPT and GitHub Copilot. For studies targeting GitHub repositories, it is crucial to consider GitHub Copilot as well, because (1) unlike ChatGPT, it is a tool tailored to software development, and (2) it is more deeply embedded in developers' workflows (their local editors, but also into the GitHub platform as a whole). Moreover, we complement the task categories by specifically discussing content and usage purposes. In addition, we identified patterns of human intervention. Hou et al. [1] reviewed literature on LLMs for software engineering. They found that software engineering research has a strong focus on code generation and program repair. We complement this observation with a detailed taxonomy of how open-source developers use LLM-based tools in their projects. In addition to code generation, we found that internationalization and natural language translation are common use cases for LLMs in open-source software projects (49 instances, see Table 3). This finding is aligned with Tufano et al. [6], who identified 12 instances of ChatGPT usage for internationalization. Such use cases highlight the need to support not only programming tasks but also broader software development activities. We further found instances of projects regulating the usage of GenAI tools, which we analyzed in more detail as part of **RQ2**. While our study partially confirms previous studies on software development tasks being automated using GenAI tools, we contribute three novel perspectives: (1) some developers deeply care about acknowledging GenAI usage in open-source software, (2) open-source maintainers try to actively guide and regulate GenAI usage, and (3) issues with generated code can trigger human interventions in open-source software projects.

Based on our findings, we recommend that **researchers** investigate developers' rationale behind self-admitted (vs. hidden) GenAI usage. Our notion of self-admitted GenAI usage, inspired by self-admitted technical debt [12], can be a valuable lens for studying GenAI usage in practice. Of course, only a fraction of the generated software artifacts contain GenAI mentions, and the artifacts that are documented might not be representative of the overall GenAI usage. Better understanding when and why developers decide to self-admit GenAI usage is one potential direction for future work. Moreover, our annotated dataset of self-admitted GenAI usage can serve as a starting point to build a tool to automatically identify true positive GenAI mentions according to the definition presented in Section 2.2. An improved and scaled detection of self-admitted GenAI usage would allow researchers to build larger datasets that could then enable more comprehensive studies on code quality and maintainability of generated code.

Software developers can browse our taxonomy of tasks, content types, and purposes to identify potential applications of GenAI tools in their projects. One central aspect is whether to establish guidelines clarifying in which cases project maintainers require contributors to disclose and acknowledge GenAI usage (see also Section 6.2).

We found that some acknowledgments were combined with warnings about the potential negative implications of GenAI tool usage. Sometimes, GenAI tools were also blamed for issues. However, acknowledgment can also serve a positive purpose, e.g., documenting prompts. The question arises not only when to acknowledge GenAI usage, but also which context to document beyond the tool name (which we focused on). In which cases does it make sense to document complete prompts and where and how should one document the generation context? Such questions can be addressed both from a **scientific** and from a **practical** perspective. A more standardized approach for documenting GenAI contributions is required, since most self-admitted GenAI usages did not document the generation context beyond brief summaries. As discussed above, we found cases documenting prompts behind GenAI usage. **GenAI tool builders** (e.g., of IDE plugins) could offer an option to record these contexts, enabling transparent acknowledgment of GenAI usage and prompt reuse.

6.2 RQ2: Existing Guidelines for GenAI Usage

Motivated by the purpose categories *Guidance and Best Practices* that we identified while answering **RQ1** (see Table 6), we further explored the policies and usage guidelines for GenAI tools that we found (see Table 7). Their content ranged from encouraging developers to use GenAI tools to prohibiting their usage entirely.

Our developer survey suggests a broad spectrum of positions that cover ethical, legal, and practical considerations. Mentioned aspects include the unclear copyright situation of the training data, the unclear implications for generated content, data privacy risks when sharing inputs with GenAI systems, and concerns regarding code quality and maintainability. Moreover, a majority of our survey participants agreed that the regulation of GenAI usage is necessary in open-source projects. In relation to that, participants argued for transparent disclosure of GenAI usage and also for documenting the generation context. It is unclear how much transparency is required and what purposes it can serve: Is a binary flag sufficient? Or is it better to document the percentage of generated content, as suggested by a participant? Or the whole prompt? Do only manually written prompts need to be disclosed, or also system prompts? This aspect is aligned with the questions raised in the discussion for **RQ1** about prompt context.

Our results suggest that **software developers**, especially those maintaining open-source software projects, should **articulate a clear position** regarding GenAI usage in their projects. The spectrum of possible positions ranges from a general recommendation to use GenAI tools, over recommendations for specific tools and use cases, to more restrictive policies requiring an extensive peer review of generated content, or policies prohibiting GenAI usage completely. Open-source projects should **clearly communicate expectations** regarding GenAI usage to their contributors. For downstream consumers of open-source dependencies, explicit GenAI policies serve as a signal of due diligence that may influence their dependency selection.

Our analysis of policies, guidelines, and developers' positions regarding GenAI regulation provides a solid foundation for **researchers** to design and conduct further **studies on how software projects regulate** GenAI usage and how such regulations impact development activity. An idea worth exploring is whether existing GenAI tools could be augmented to **capture provenance information during generation** that could be automatically added to source code comments, commit messages, or other artifacts such as Software Bills of Materials (SBOMs) [34] or Software Bill of Materials for AI (SBOM for AI) [35]. There are already open-source projects that extensively document prompts in commit messages.[2] This provenance information is essential to study the **long-term impact of code generation on maintainability**, but also to support software supply chain transparency and vulnerability management. Researchers can contribute to the **development of standardized metadata formats** to capture provenance and traceability information of code and other software artifacts.

6.3 RQ3: Impact of GenAI Usage on Code Churn

Our results for RQ3 challenge popular narratives about the impact of GenAI on software development. Contrary to claims

2. github.com/cloudflare/workers-oauth-provider/commit/adcb...

in the GitClear report, which was extensively discussed in the software development community [36, 37], we did not find an increasing code churn after GenAI adoption. The overall trend we observed pointed in the opposite direction, i.e., we noticed a decreasing average code churn. This is in line with a study by Grewal et al. [38] which examined how ChatGPT-generated code is integrated into GitHub projects. They found that approximately 54% of the generated code lines were integrated and only 2.5% of the generated snippets were later modified. However, our RDD analysis revealed that three repositories (3/26, 11.5%) had a significant upward file-based code churn trend with a positive slope ($p < 0.05$). This indicates that a subset of repositories exhibited a progressively faster increase in code churn after GenAI adoption. To contextualize this finding, we note that a larger proportion of repositories (12/26, 46.2%) showed an upward trend with a negative slope, i.e., churn increased, but the increase decelerated over time. In addition, 26.9% (7/26) showed downward trends. This heterogeneity suggests that GenAI's impact on code churn is context-dependent rather than uniformly negative, as suggested by the GitClear report.

Our **RQ3** results motivate us to recommend that **researchers** explore the factors that contribute to increased code churn. The patterns we identified using our RDD analysis are a **valuable lens for clustering projects**. This clustering can inform a detailed qualitative study of projects that exhibit similar patterns. The difference between our results and the GitClear report can be partially attributed to the methodological differences between the studies. While GitClear used a global cutoff date, we used the first GenAI mention in a repository as a proxy for GenAI adoption, thus following a more fine-grained approach. Moreover, we introduced **file-level and line-level code churn** and calculated churn at the project-level and globally. Our definitions and the code in our supplementary material enable other researchers to consider code churn in their own studies.

For **software developers**, we further suggest **monitoring the impact of GenAI** on the software projects they maintain or contribute to. Our results suggest that the impact of GenAI adoption on the development activity in software projects might not be as clear as suggested by the GitClear report. Considering that we did notice an increasing code churn in several projects, it is nevertheless important for project maintainers to **monitor the development activity** and the quality of contributions. Going forward, we might develop our code churn implementation into a tool that project maintainers can easily integrate into their repositories.

7 Related Work

To situate our work, we organize related work into three themes that align with the dimensions explored in our study: (i) studies examining GenAI tasks and purposes, (ii) studies on risks and integration concerns around GenAI adoption, and (iii) studies on the impact of GenAI on software development processes.

7.1 GenAI Tasks and Purposes

Many researchers have focused on understanding how developers use GenAI tools across different software engineering activities and the types of content these tools generate.

Besides our work and that of Tufano et al. [6], a few other studies have also established taxonomies of GenAI tasks in software development. Sagdic et al. [39] used semantic modeling and expert analysis to understand the topics developers discuss when interacting with ChatGPT, revealing 17 topics in seven categories, with over one-quarter of prompts focused on seeking programming guidance. Champa et al. [40] defined 12 categories of software development tasks based on a literature review and applied these categories to analyze developers' interaction with ChatGPT. They found that code quality management and commit issue resolution represent the most frequent assistance requests. These additional taxonomies provide further evidence of the breadth of software engineering activities in which developers rely on GenAI assistance.

Research examining the purposes and contexts of GenAI usage has revealed several patterns in the ways developers integrate AI tools into their workflow. Using the DevGPT dataset [8], Jin et al. [41] found that LLM-generated code was rarely used as production-ready code, providing concrete evidence of the gap between GenAI capabilities demonstrated in research settings and their practical application in real-world development scenarios. Their analysis revealed distinct purposes for AI-generated content: nearly one-third of the generated code was not integrated at all, whereas approximately one-quarter was incorporated into auxiliary files, such as README documentation files and test cases, rather than production codebases. This pattern suggests that developers may primarily use GenAI for explanatory and educational purposes rather than direct code production. Xiao et al. [7] studied GenAI-developer collaboration through the analysis of over 18K pull requests where descriptions were crafted by GitHub Copilot. They found that developers complement AI-generated content with manual input, underlining the collaborative nature of human-AI interaction in producing development artifacts that require iterative refinement and enhancement. Our analysis complements these studies by focusing on self-admitted GenAI usage, examining how and why developers explicitly acknowledge AI assistance in their development artifacts across different tasks and content types.

Despite the increasing amount of research studying GenAI assistance in software development, an important gap remains in our understanding of self-admitted GenAI usage patterns in the wild, particularly regarding how developers openly acknowledge and document GenAI assistance across different software engineering tasks and purposes.

7.2 GenAI Risks and Integration Concerns

The integration of GenAI tools into software development workflows has raised serious concerns regarding security risks and responsible adoption practices. Research in this area has focused on understanding the varied challenges developers face when incorporating these tools, ranging from immediate security and quality concerns to broader organizational and workflow integration issues.

Regarding security concerns, Sandoval et al. [42] examined the security implications of using AI-written code assistants and found that LLMs may inadvertently introduce vulnerabilities into codebases, highlighting the need for careful screening when integrating AI-generated code. Asare et al. [43] compared the performance of GitHub Copilot with human developers in secure coding tasks. They found that the GenAI tool exhibits patterns of security weaknesses similar to those of human programmers, raising questions about code review practices and security governance.

Code quality issues have emerged as another major risk factor closely related to security concerns. Siddiq et al. [44] used the DevGPT dataset to assess the quality of ChatGPT-generated code and found that such code suffers from issues including undefined variables, improper documentation, and security vulnerabilities related to resource management. These quality concerns extend across different programming contexts, as demonstrated by Moratis et al. [45], who analyzed 144 JavaScript code blocks generated by ChatGPT and found that approximately one-quarter of AI-written code blocks contained one or more violations. They observed that approximately 50% of the violations related to best practices, 37% related to code style issues, and 12% were classified as error-prone violations. Quality concerns increase when considering code modification versus creation. Rabbi et al. [46] analyzed 1,756 AI-generated

Python code snippets, systematically distinguishing between code created from scratch and modified code. They found that code modified using ChatGPT more frequently suffers from quality issues compared to ChatGPT-generated code. This pattern suggests that different types of AI assistance may require different governance approaches. Zhang et al. [47] identified code smells in Kubernetes manifest files generated by AI tools, showing that quality concerns extend beyond traditional programming tasks to infrastructure-as-code artifacts.

The successful adoption of GenAI tools requires substantial organizational changes that address both technical and human factors. Sauvola et al. [48] analyzed the potential of GenAI for future software development and identified skill-gap challenges where developers lack necessary AI expertise, highlighting the need for investment in training programs to develop competencies in prompt engineering, AI output validation, and human-AI collaboration. These organizational challenges have also led researchers to investigate GenAI adoption patterns. Russo et al. [49] developed the Human-AI Collaboration and Adaptation Framework, a theoretical model designed to understand and predict GenAI tool adoption in software engineering. They found that compatibility factors—particularly, how well AI tools integrate within existing development workflows—serve as the primary driver of organizational adoption decisions. This finding challenges conventional technology acceptance theories [50], as traditional factors, such as perceived usefulness, social influence, and personal innovativeness, proved less influential than expected.

The integration of GenAI tools into complex software development workflows and ecosystems also involves legal considerations. Wintersgill et al. [51] examined OSS license compliance from the perspectives of legal practitioners, identifying challenges in managing compliance for traditional software components. As AI-generated code becomes more and more prevalent in open-source projects, OSS compliance frameworks may need to be adapted to address questions of attribution, licensing obligations, and intellectual property considerations for AI-generated content.

The limited analysis of current GenAI adoption policies represents an important research opportunity. Our work contributes to filling this gap by examining how open-source projects are developing governance approaches to manage GenAI adoption and the specific risks and concerns (technical, ethical, and legal) that drive these policy decisions.

7.3 GenAI Impact on Software Development

A substantial amount of research has been conducted on quantifying the impacts of GenAI tools on software development processes and outcomes, moving beyond anecdotal evidence and developer perceptions.

Ziegler et al. conducted a large-scale empirical study examining GitHub Copilot's effect on developer productivity [4]. Based on surveys and usage telemetry, they found that developers perceived productivity improvements when using Copilot, particularly for repetitive and routine coding activities, with the perceived magnitude of improvement varying considerably based on task complexity and context.

In 2024, GitClear analyzed over 150 million lines of code across GitHub repositories from 2020 to 2023 to assess the impact of AI-assisted development on code quality [11], attributing rising code churn to GenAI adoption and interpreting it as indicative of code that was incomplete or erroneous when initially committed. The 2025 follow-up report [52], based on an expanded dataset of 211 million lines through 2024, reported a rise in code churn from 4.5% in 2023 to 5.7% in 2024, a 39.9% drop in refactoring, and a 17.1% increase in copy-pasted code. The same report documented an eight-fold increase in duplicated code blocks during 2024 and reported that for the first time, copy-pasted lines exceeded moved lines within commits, indicating a fundamental shift away from code refactoring toward code duplication and raising concerns about growing technical debt and the long-term sustainability of AI-assisted coding. However, our analysis of code churn in select GitHub repositories in which developers acknowledged GenAI usage reveals different patterns, suggesting that the relationship between AI assistance and code quality may be more nuanced than these industry reports indicate.

Pearce et al. [53] conducted a security assessment of code contributions generated by GitHub Copilot across multiple programming languages and contexts. They found systematic security weaknesses in AI-generated code, arguing that security issues introduced by GenAI tools stem from the models' training on publicly available code repositories, which inherently contain security flaws. Asare et al. [43] compared vulnerability rates between human-written and Copilot-generated code and found that, while the GenAI tool introduced security vulnerabilities, the rates were not higher than those introduced by human developers. These findings suggest that security concerns with AI-generated code may reflect broader challenges in secure coding practices rather than AI-specific problems.

Our study adds to the existing body of knowledge by analyzing self-admitted GenAI usage across more than 200,000 OSS repositories and conducting a study of code churn, showing how OSS projects use GenAI tools and how their usage impacts development activity.

8 Threats to Validity

In this section, we discuss the threats to the construct, internal, and external validity of our study.

8.1 Construct Validity

Our reliance on self-admitted GenAI usage introduces two main threats. First, we only captured the visible part of GenAI adoption in OSS projects. Developers who use GenAI tools without leaving a trace remain outside of our analysis scope, meaning our findings represent a lower bound on actual GenAI adoption. Therefore, the observed patterns must be interpreted within this context, as they may not apply to all instances of GenAI-assisted software development. Second, some self-admitted mentions introduce ambiguity in determining which portions of code were generated by GenAI tools. When a developer comments that the code was "generated by ChatGPT," this may refer to complete classes, functions/methods, code blocks, or merely an initial structure that was subsequently modified. Although we always examined the whole context around a GenAI mention, we might have misclassified its scope and purpose in some instances.

The focus on self-admitted usage has implications for our answers to the research questions. Developers may be more likely to acknowledge GenAI usage for mundane tasks such as translation rather than for core development tasks such as implementing complex business logic. Therefore, the observed frequencies might reflect what developers feel comfortable disclosing rather than their actual usage (**RQ1**). Strict usage policies might lead to fewer self-admitted usages, as contributors might avoid disclosure (**RQ2**). Finally, unacknowledged usage before the first explicit mention could distort the pre-adoption baseline for our code churn analysis (**RQ3**). However, at the same time, we consider self-admitted GenAI usage a useful analytical lens for studying the impact of GenAI adoption in open-source software projects. Given our manual validation, the precision of the identified GenAI usages is high, even though the overall recall of all GenAI usages is low.

When calculating code churn, we used the first explicit mention of GenAI tools as a proxy for adoption timing, which may not accurately reflect when projects actually began using GenAI. However, our generous analysis window of 360 days before and after this point and robustness checks (e.g., placebo

tests) help accommodate potential discrepancies in adoption dates. Although we focused on only one measure of code quality, the relevance of code churn as a metric was motivated by industry research. The GitClear 2025 report [52] documented a rise in churn from 4.5% in 2023 to 5.7% in 2024, coinciding with the proliferation of GenAI-assisted development. This increase correlates with two related trends: a 39.9% decline in "moved" code (indicating reduced refactoring) and a 17.1% rise in "copy/pasted" code. Previous research links reduced refactoring to higher defect rates [54] and code clones to increased technical debt [55, 56]. Future work could expand our analysis by considering additional metrics.

8.2 Internal Validity

Our heuristic-based approach for detecting GenAI mentions may have produced false negatives, particularly for mentions using non-standard terminology or abbreviations. We addressed this by developing comprehensive regular expressions, covering common naming variations, and conducting a thorough manual validation of the identified mentions. We rely on manually annotated data, which may be miscoded due to the subjective nature of understanding the coding book. To mitigate this threat and ensure consistency in our qualitative analysis, we implemented a rigorous manual review process with multiple raters in several rounds of independent coding, achieving high inter-rater reliability.

The number of policies and guidelines we analyzed and the number of survey responses we received were relatively low. However, even this limited data revealed diverse regulation approaches and opinions, motivating future research.

8.3 External Validity

We restricted our analysis to public repositories hosted on GitHub, focusing on five popular programming languages. The self-admitted GenAI usage we studied might not reflect general GenAI usage—self-admitted or hidden—in other repositories, programming languages, or industrial software projects. However, the selected languages represent the most commonly used languages according to the 2024 GitHub Octoverse report [13]. Furthermore, our filtering criteria for engineered software projects ensured that our findings reflect practices in actively maintained software projects. Moreover, our results may not generalize to other open-source platforms such as GitLab, which may have different norms and adoption patterns.

The developer survey that we conducted as part of **RQ2** received eight responses from contributors of one of the 12 projects that we identified to have explicit GenAI usage policies and guidelines. Their responses might not reflect the views of a broader developer population. Future work should extend this analysis, for example, by analyzing GitHub Discussion threads on AI regulation. Finally, our focus on ChatGPT and GitHub Copilot might not capture the usage patterns of other GenAI tools that were released more recently, e.g., Claude Code.

9 Conclusion

This study introduced *self-admitted GenAI usage*—explicit references to LLM-based tools such as ChatGPT and GitHub Copilot—as a novel lens for examining how generative AI is used in open-source software development. In our mixed-methods study design, we first mined more than 200,000 GitHub repositories, isolating 1,292 true-positive GenAI mentions across 156 projects. Qualitative open coding of these instances and subsequent card sorting yielded taxonomies of 32 assisted tasks, 10 content types, and 11 usage purposes. We complemented this content analysis with a survey of project contributors and a systematic review of 13 project-level policies and guidelines. In addition, we performed a regression-discontinuity (RDD) analysis of code churn in the 149 repositories that contained sufficient data to study the impact of GenAI adoption on open source software projects. Based on our findings, we derive several actionable implications.

Implications for software developers. The **RQ1** results show that developers most often use GenAI for code generation, natural-language translation, and refactoring—with explicit acknowledgment as the dominant purpose. We also observed recurring follow-up actions, including bug fixes, refactorings, reversions, and deletions, triggered by earlier GenAI-generated content. Together with our **RQ3** task-level analysis, which shows that generation tasks are more rework-prone than optimization or maintenance, this suggests that GenAI output has a provisional nature. Developers should therefore plan explicit validation and revision steps after GenAI-assisted generation, particularly for code. While acknowledgment of GenAI usage is common (**RQ1**), contextual metadata such as prompts, model versions, or scope of generation is rarely recorded. This gap points to a concrete opportunity for developers to adopt provenance conventions (e.g., structured comments or commit tags) that better support the corrective practices already observed in the data. The **RQ2** findings show substantial variation in project-level GenAI governance. Developers contributing to different projects should therefore avoid assuming uniform norms and instead adapt their GenAI usage and disclosure practices to project-specific guidelines.

Implications for project maintainers. RQ2 shows that maintainers are already actively regulating GenAI usage, but through heterogeneous approaches ranging from bans to selective encouragement. This diversity suggests that generic policies are unlikely to work. Instead, maintainers should align GenAI guidelines with project-specific factors such as contribution patterns, review capacity, and risk tolerance. The **RQ3** results show no systematic increase in code churn after GenAI adoption, contradicting prominent industry claims. While some repositories exhibit increased churn, many show stable or decreasing trends, and effects vary widely across projects. This suggests that restrictive policies motivated solely by assumed quality degradation are not well supported by the evidence. Project-level monitoring, e.g., tracking churn or related indicators before and after GenAI adoption, offers a project-specific and evidence-based alternative. The findings for **RQ1** highlight that GenAI is frequently used for PR descriptions and documentation, which are comparatively low-risk artifacts. Maintainers can act on this by explicitly encouraging GenAI usage in these areas to improve communication and review efficiency while limiting exposure to higher rework costs.

Implications for tool builders and platform providers. RQ1 shows that developers already engage in voluntary disclosure of GenAI usage when the cost is low, indicating that the main limitation to transparency lies in the absence of supporting mechanisms rather than in developer opposition. Tool builders can act on this by offering built-in, optional disclosure mechanisms such as automatic annotations or commit templates that integrate with existing workflows. **RQ3** indicates that GenAI-assisted code generation is more rework-prone than other uses. Tools could respond by explicitly supporting post-generation validation, for example, through prompts or workflow affordances that encourage human review of generated code. At the platform level, support for project-specific GenAI policies, such as configurable disclosure requirements in pull requests, could help operationalize the heterogeneous governance approaches observed in **RQ2**.

Implications for researchers. Our curated dataset of 1,292 self-admitted GenAI mentions and the taxonomy derived in **RQ1** provide a foundation for scaling empirical studies via automated detection and large-scale mining. Researchers can build on this work by developing detectors for self-admitted

GenAI usage and using them to study adoption and disclosure practices at scale. **RQ3** highlights the importance of methodological granularity. Analyses that ignore project-specific adoption points or governance contexts risk drawing misleading conclusions. Future studies should therefore favor repository- and task-level designs when assessing the impact of GenAI usage, rather than relying on aggregate trends. Because this study focuses on version-controlled artifacts, future research should extend the analysis to adjacent artifacts such as pull-request discussions and review comments. Such extensions would naturally build on our dataset, taxonomy, and churn analysis, helping to complete the picture of how self-admitted GenAI usage shapes collaborative software development.

ACKNOWLEDGMENTS

We thank all survey participants for providing valuable insights for our research. The research contribution of Fabio Calefato was partially supported by the European Union, NextGenerationEU through the Italian Ministry of University and Research, Projects PRIN 2022 ("QualAI: Continuous Quality Improvement of AI-based Systems", grant n. 2022B3BP5S, CUP: H53D23003510006). Tao Xiao is supported in part by JSPS Grant-in-Aid for JSPS Fellows 23KJ1589 and the Kayamori Foundation of Informational Science Advancement.

REFERENCES

[1] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. C. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 8, pp. 220:1–220:79, 2024.

[2] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *CHI Extended Abstracts '22*, 2022.

[3] J. T. Liang, C. Yang, and B. A. Myers, "A large-scale survey on the usability of AI programming assistants: Successes and challenges," in *ICSE '24*, 2024, pp. 52:1–52:13.

[4] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian, "Measuring GitHub Copilot's impact on productivity," *Commun. ACM*, vol. 67, no. 3, pp. 54–63, 2024.

[5] N. Nguyen and S. Nadi, "An empirical evaluation of GitHub Copilot's code suggestions," in *MSR '22*, 2022, pp. 1–5.

[6] R. Tufano, A. Mastropaolo, F. Pepe, O. Dabic, M. Di Penta, and G. Bavota, "Unveiling ChatGPT's usage in open source projects: A mining-based study," in *MSR '24*, 2024, pp. 571–583.

[7] T. Xiao, H. Hata, C. Treude, and K. Matsumoto, "Generative AI for pull request descriptions: Adoption, impact, and developer interventions," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 1043–1065, 2024.

[8] T. Xiao, C. Treude, H. Hata, and K. Matsumoto, "DevGPT: Studying developer-chatgpt conversations," in *MSR '24*, 2024, pp. 227–230.

[9] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social coding in GitHub: transparency and collaboration in an open software repository," in *CSCW '12*, 2012, pp. 1277–1286.

[10] D. Stenberg, "Death by a thousand slops," 2025, accessed 2026-01-05. [Online]. Available: https://daniel.haxx.se/blog/2025/07/14/death-by-a-thousand-slops/

[11] GitClear, "Coding on Copilot: 2023 data suggests downward pressure on code quality," 2024, accessed 2026-01-05. [Online]. Available: https://gitclear.com/coding_on_copilot_data_shows_ais_downward_pressure_on_code_quality

[12] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *ICSME '14*, 2014, pp. 91–100.

[13] GitHub, "Octoverse: AI leads Python to top language as the number of global developers surges," 2024, accessed 2025-07-01. [Online]. Available: https://github.blog/news-insights/octoverse/octoverse-2024/

[14] O. Dabic, E. Aghajani, and G. Bavota, "Sampling projects in GitHub for MSR studies," in *MSR '21*, 2021, pp. 560–564.

[15] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating GitHub for engineered software projects," *Empir. Softw. Eng.*, vol. 22, no. 6, pp. 3219–3253, 2017.

[16] Stack Overflow, "Stack Overflow Developer Survey 2023," 2023, accessed 2025-12-25. [Online]. Available: https://survey.stackoverflow.co/2023/

[17] J. L. Fleiss, "Measuring nominal scale agreement among many raters," *Psychol. Bull.*, vol. 76, no. 5, pp. 378–382, 1971.

[18] T. Xiao, Y. Fan, F. Calefato, C. Treude, R. G. Kula, H. Hata, and S. Baltes, "Self-admitted GenAI usage in open-source software," Jan. 2026. [Online]. Available: https://doi.org/10.5281/zenodo.15871467

[19] K. Charmaz, *Constructing grounded theory*. SAGE, 2014.

[20] A. J. Viera, J. M. Garrett *et al.*, "Understanding interobserver agreement: the kappa statistic," *Fam. Med.*, vol. 37, no. 5, pp. 360–363, 2005.

[21] B. Guo, X. Zhang, Z. Wang, M. Jiang, J. Nie, Y. Ding, J. Yue, and Y. Wu, "How close is chatgpt to human experts? comparison corpus, evaluation, and detection," *arXiv preprint 2301.07597*, 2023.

[22] J. C. Munson and S. G. Elbaum, "Code churn: A measure for estimating the impact of code change," in *ICSM '98*, 1998, pp. 24–31.

[23] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *ICSE '05*, 2005, pp. 284–292.

[24] S. Wehaibi, E. Shihab, and L. Guerrouj, "Examining the impact of self-admitted technical debt on software quality," in *SANER '16*, 2016, pp. 179–188.

[25] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[26] J. Cohen, *Statistical power analysis for the behavioral sciences*. Routledge, 2013.

[27] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *Ann. Math. Stat.*, vol. 18, no. 1, pp. 50–60, 1947.

[28] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions," *Psychol. Bull.*, vol. 114, no. 3, pp. 494–509, 1993.

[29] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, and L. Devine, "Exploring methods for evaluating group differences on the nsse and other surveys: Are the t-test and cohen's d indices the most appropriate choices," in *Annual Meeting of SAIR*, vol. 14, 2006.

[30] D. L. Thistlethwaite and D. T. Campbell, "Regression-discontinuity analysis: An alternative to the ex post facto experiment," *J. Educ. Psychol.*, vol. 51, no. 6, pp. 309–317, 1960.

[31] G. W. Imbens and T. Lemieux, "Regression discontinuity designs: A guide to practice," *J. Econom.*, vol. 142, no. 2, pp. 615–635, 2008.

[32] M. Wessel, A. Serebrenik, I. Wiese, I. Steinmacher, and M. A. Gerosa, "Effects of adopting code review bots on pull requests to OSS projects," in *ICSME '20*, 2020, pp. 1–11.

[33] M. Wessel, J. Vargovich, M. A. Gerosa, and C. Treude, "GitHub actions: The impact on the pull request process," *Empir. Softw. Eng.*, vol. 28, no. 6, p. 131, 2023.

[34] D. Riehle, "The software bill of materials," *Computer*, vol. 58, no. 4, pp. 115–120, 2025.

[35] B. Xia, T. Bi, Z. Xing, Q. Lu, and L. Zhu, "An empirical study on software bill of materials: Where we stand and the road ahead," in *ICSE '23*, 2023, pp. 2630–2642.

[36] Hacker News, "Coding on Copilot: Data suggests downward pressure on code quality," 2024, accessed 2026-01-05. [Online]. Available: https://news.ycombinator.com/item?id=39177008

[37] Reddit, "New GitHub Copilot Research Finds 'Downward Pressure on Code Quality'," 2024, accessed 2026-01-05. [Online]. Available: https://www.reddit.com/r/programming/comments/1ac7cb2/new_github_copilot_research_finds_downward/

[38] B. Grewal, W. Lu, S. Nadi, and C.-P. Bezemer, "Analyzing developer use of ChatGPT generated code in open source github projects," in *MSR '24*, 2024, pp. 157–161.

[39] E. Sagdic, A. Bayram, and M. R. Islam, "On the taxonomy of developers' discussion topics with ChatGPT," in *MSR '24*, 2024, pp. 197–201.

[40] A. I. Champa, M. F. Rabbi, C. Nachuma, and M. F. Zibran, "ChatGPT in action: Analyzing its use in software development," in *MSR '24*, 2024, pp. 182–186.

[41] K. Jin, C.-Y. Wang, H. V. Pham, and H. Hemmati, "Can ChatGPT support developers? an empirical evaluation of large language models for code generation," in *MSR '24*, 2024, pp. 167–171.

[42] G. Sandoval, H. Pearce, T. Nys, R. Karri, S. Garg, and B. Dolan-Gavitt, "Lost at C: A user study on the security implications of large language model code assistants," in *USENIX Security Symposium '23*, J. A. Calandrino and C. Troncoso, Eds., 2023, pp. 2205–2222.

[43] O. Asare, M. Nagappan, and N. Asokan, "Is GitHub's Copilot as bad as humans at introducing vulnerabilities in code?" *Empir. Softw. Eng.*, vol. 28, no. 6, p. 129, 2023.

[44] M. L. Siddiq, L. Roney, J. Zhang, and J. C. D. S. Santos, "Quality assessment of chatgpt generated code and their use by developers," in *MSR '24*, 2024, pp. 152–156.

[45] K. Moratis, T. Diamantopoulos, D.-N. Nastos, and A. Symeonidis, "Write me this code: An analysis of ChatGPT quality for producing source code," in *MSR '24*, 2024, pp. 147–151.

[46] M. F. Rabbi, A. I. Champa, M. F. Zibran, and M. R. Islam, "Ai writes, we analyze: The ChatGPT python code saga," in *MSR '24*, 2024, pp. 177–181.

[47] Y. Zhang, R. Meredith, W. Reeves, J. Coriolano, M. A. Babar, and A. Rahman, "Does generative ai generate smells related to container orchestration?: An exploratory study with kubernetes manifests," in *MSR '24*, 2024, pp. 192–196.

[48] J. Sauvola, S. Tarkoma, M. Klemettinen, J. Riekki, and D. Doermann, "Future of software development with generative AI," *Autom. Softw. Eng.*, vol. 31, no. 1, p. 26, 2024.

[49] D. Russo, "Navigating the complexity of generative AI adoption in software engineering," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 5, pp. 135:1–135:50, 2024.

[50] N. Marangunić and A. Granić, "Technology acceptance model: a literature review from 1986 to 2013," *Univers. Access Inf. Soc.*, vol. 14, no. 1, pp. 81–95, 2015.

[51] N. Wintersgill, T. Stalnaker, L. A. Heymann, O. Chaparro, and D. Poshyvanyk, ""the law doesn't work like a computer"': Exploring software licensing issues faced by legal practitioners," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 882–905, 2024.

[52] GitClear, "AI Copilot Code Quality: 2025 Data Suggests 4x Growth in Code Clones," 2025, accessed 2026-01-05. [Online]. Available: https://gitclear.com/ai_assistant_code_quality_2025_research

[53] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of GitHub copilot's code contributions," *Commun. ACM*, vol. 68, no. 2, pp. 96–105, 2025.

[54] P. Mohagheghi, R. Conradi, O. M. Killi, and H. Schwarz, "An empirical study of software reuse vs. defect-density and stability," in *ICSE '04*, 2004, pp. 282–292.

[55] L. Aversano and L. Nardi, "Investigating on the impact of software clones on technical debt," in *TechDebt@ICSE '19*, 2019, pp. 108–112.

[56] D. Feitosa, A. Ampatzoglou, A. Gkortzis, S. Bibi, and A. Chatzigeorgiou, "Code reuse in practice: Benefiting or harming technical debt," *J. Syst. Softw.*, vol. 167, p. 110618, 2020.