Unveiling ChatGPT's Usage in Open Source Projects: A Mining-based Study

Rosalia Tufano*
SEART @ Software Institute
Università della Svizzera italiana
Switzerland

Antonio Mastropaolo*
SEART @ Software Institute
Università della Svizzera italiana
Switzerland

Federica Pepe
Department of Engineering
University of Sannio
Italy

Ozren Dabić
SEART @ Software Institute
Università della Svizzera italiana
Switzerland

Massimiliano Di Penta
Department of Engineering
University of Sannio
Italy

Gabriele Bavota
SEART @ Software Institute
Università della Svizzera italiana
Switzerland

ABSTRACT

Large Language Models (LLMs) have gained significant attention in the software engineering community. Nowadays developers have the possibility to exploit these models through industrial-grade tools providing a handy interface toward LLMs, such as OpenAI's ChatGPT. While the potential of LLMs in assisting developers across several tasks has been documented in the literature, there is a lack of empirical evidence mapping the actual usage of LLMs in software projects. In this work, we aim at filling such a gap. First, we mine 1,501 commits, pull requests (PRs), and issues from open-source projects by matching regular expressions likely to indicate the usage of ChatGPT to accomplish the task. Then, we manually analyze these instances, discarding false positives (*i.e.,* instances in which ChatGPT was mentioned but not actually used) and categorizing the task automated in the 467 true positive instances (165 commits, 159 PRs, 143 issues). This resulted in a taxonomy of 45 tasks which developers automate via ChatGPT. The taxonomy, accompanied with representative examples, provides (i) developers with valuable insights on how to exploit LLMs in their workflow and (ii) researchers with a clear overview of tasks that, according to developers, could benefit from automated solutions.

CCS CONCEPTS

• **Software and its engineering → Software tools**.

KEYWORDS

ChatGPT, Empirical study

ACM Reference Format:
Rosalia Tufano, Antonio Mastropaolo, Federica Pepe, Ozren Dabić, Massimiliano Di Penta, and Gabriele Bavota. 2024. Unveiling ChatGPT's Usage in Open Source Projects: A Mining-based Study. In *21st International Conference on Mining Software Repositories (MSR '24), April 15–16, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3643991.3644918

*Both authors contributed equally to the paper.

MSR '24, April 15–16, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0587-8/24/04.
https://doi.org/10.1145/3643991.3644918

1 INTRODUCTION

Recommender systems for software engineers have been defined by Robillard *et al.* [93] as:

> *"Software applications that provide information items estimated to be valuable for a software engineering task in a given context"*

Over the years, researchers have developed various forms of recommenders, aimed at suggesting relevant code elements for a given task [44, 85, 111], helping to fix bugs [47, 80, 105, 114] and vulnerabilities [53, 67, 69], and even to automatically document software systems [43, 82, 99].

In the last decade, the increasing gain of maturity and improvement of deep learning architectures, the availability of hardware infrastructures, and of data from forges such as GitHub has opened the road towards the development of recommenders able not only to better solve the aforementioned problems, but also to perform tasks for which no recommender was previously thought in the past, including generating entire code blocks [55, 102], automatically reviewing source code [75, 103, 106, 108], or generating scenarios for automatically reproducing issues [49, 65, 116].

The advent of Large Language Models (LLMs) and, lately, of LLM-based chat bots such as ChatGPT [1] has opened new development landscapes. In such a context, a developer can, from a single tool, receive help for a wide number of tasks: one can ask ChatGPT to design an architecture, write or complete source code to achieve a given task, review and possibly refactor/optimize existing code, repair bugs, generate tests, and so on. In other words, today's ChatGPT and tomorrow's similar tools from other providers will gradually become the main source of help for developers, essentially replacing what a colleague, user manuals, the Web (including forums such as Stack Overflows) and, recently, code completion specialized tools such as GitHub Copilot [2], have done so far.

Given such a scenario, it would be worthwhile understanding how developers have leveraged ChatGPT so far to achieve different goals. Certainly, this goal could possibly be achieved through interviews and survey questionnaires. However, we have decided to follow a radically different approach, *i.e.,* by mining traces of ChatGPT usages in GitHub: commit messages, issues, and pull requests (PRs).

This is because on the one hand, we could observe how developers "admit" the use of ChatGPT in their (open-source) projects, but, also, how such code is being reviewed before being merged.

To conduct our study, we first mined all commits, issues, and PRs from GitHub that match the keyword "ChatGPT ". Then, we extracted n-grams surrounding the word ChatGPT and manually reviewed them for further filtering. This allowed us to filter the initial sample to reduce the chance of false positives, *e.g.,* an issue mentioning ChatGPT but not using it for the automation of a task. Then, we performed an open coding based on card sorting [97] on all 1,501 candidate instances (*i.e.,* commit/issue/PR) we identified, classifying the ChatGPT purpose for each instance (*i.e.,* why it was used) or discarding the instance as a false positive.

As a result, we obtained a taxonomy of purposes for using ChatGPT in the automation of a software-related task. The taxonomy features seven root categories and 52 categories in total.

For each category, we discuss typical use cases, as well as implications for practitioners and researchers. Also, we highlight and discuss scenarios for which the use of ChatGPT turned out to be failing, counterproductive, or risky for the given activity.

All data used in our study is publicly available [107].

2 STUDY DESIGN

The *goal* of the study is to unveil the purposes for which LLM recommenders are used to support the development of open-source projects. The *context* consists of ChatGPT, as a representative of state-of-the-art LLMs, and of 1,501 manually inspected commits, PRs, and issues sampled from open source projects hosted on GitHub. Our study aims at answering the following research question:

> *What are the software-related tasks for which developers document the support received by ChatGPT?*

We answer this research question by mining, from development artifacts, traces of ChatGPT usages. We focus on artifacts for which it is possible to perform keyword-matching queries on GitHub. As such, we search for commit messages, PRs, and issues mentioning ChatGPT in their textual content (Section 2.1). We do not consider GitHub discussions, as we are interested to analyze text directly traceable to software artifacts. Then, we manually inspect 1,501 instances with the goal of categorizing the task(s) supported by ChatGPT (if any) in each of them (*e.g., generate tests*, *code review*) — see Section 2.2. The obtained categories of tasks have then been used to derive a taxonomy of tasks supported by ChatGPT. Such a taxonomy provides (i) developers with a comprehensive catalog of usage scenarios in which LLM recommenders can be leveraged; and (ii) researchers with software-related tasks which could benefit from automation, possibly through specialized solutions to be developed rather than via generic LLM recommenders such as ChatGPT.

In the following, we detail the steps behind our study design.

2.1 Mining Candidate Instances

The goal of this step is to identify commits, PRs, and issues in which ChatGPT has been *likely* used to support one or more tasks. False positive instances (*e.g.,* instances in which ChatGPT was mentioned but not actually leveraged) will be discarded in a later stage.

We started by querying—on June 12, 2023—the GitHub APIs to identify all commits, PRs, and issues containing the word "ChatGPT". For commits, the search was performed on the commit message/body, while for PRs and issues the target was their title and description. The output of this step were 186,425 commits, 15,629 PRs, and 31,934 issues (233,988 overall instances). By inspecting the retrieved instances, we noticed a predominance of false positives, mostly due to projects which integrate ChatGPT (*i.e.,* use the ChatGPT APIs) to offer features to their users (*e.g.,* a chat bot) rather than using it for automating software-related tasks.

We then performed a first filtering to automatically discard as many false positives as possible. To this aim, we extracted from the collected instances all forward/backward 2-grams and 3-grams containing the word "ChatGPT". For example, let us assume that a commit message features the sentence: "*Implemented matrix transposition with the help of ChatGPT*". In this case, we extract the following backward *n*-grams: "*of ChatGPT*", and "*help of ChatGPT*". Instead, a PRs titled *Used ChatGPT to implement tests* will result in the backward 2-gram "*used ChatGPT*" and in the forward *n*-grams "*ChatGPT to*" and "*ChatGPT to implement*".

We then sorted all extracted *n*-grams in ascending order of frequency, and inspected those appearing in at least 0.02% of the instances (>1k instances). We classified each *n*-gram as likely indicating the ChatGPT support in a task (*e.g.,* "*ChatGPT to generate*") or as likely indicating false positives (*e.g.,* "*ChatGPT API integration*"). This resulted in a set of 34 relevant *n*-grams available in our replication package [107]. We excluded all instances not containing at least one of such *n*-grams, and all those belonging to GitHub repositories having less than 10 stars in an attempt to filter out toy projects. Finally, we removed duplicates (*e.g.,* duplicated commits due to forked repositories) obtaining a final set of 1,501 instances, distributed as follows: 527 commits, 327 PRs, and 647 issues. The 1,501 manually analyzed issues, commits, and PRs belong to 732 different projects.

2.2 Manual Analysis and Taxonomy Definition

The goal of the manual analysis was to characterize within each of the 1,501 instances the task(s) (partially) automated using ChatGPT. Five authors (from now on evaluators) were involved in the manual inspection. Each instance has been independently inspected by two evaluators. The whole process was supported by a web app we developed that implemented the required logic and provided a handy interface to categorize the instance. For each instance, the evaluator was presented with: (i) the metadata as returned by the GitHub APIs (*e.g.,* for a commit: its author, message, body, date, etc.); (ii) the *n*-gram that was matched in that specific instance (*e.g.,* "*ChatGPT to generate*"); and (iii) the link to the instance on GitHub for an easier inspection.

The categorization required the assignment of one or more lebels to an instance, describing the automated task(s) (*e.g., refactoring code*, *write documentation*). In case the manual inspection revealed that ChatGPT was not actually used to automate software-related tasks, the instance was discarded.

Since there are no documented taxonomies of software-related tasks automated with the support of LLM recommenders, we followed an open coding strategy [97].

Specifically, each evaluator could introduce a new label, as they felt it was needed to properly describe the automated task(s). After the label was added, it became available, through the web app, to the other evaluators. While this goes against the notion of open coding, in a scenario in which there are no pre-defined categories this helps to reduce the chance of multiple evaluators defining similar labels to describe the same task while not introducing a substantial bias in the process.

It is important to mention that the labeling process has not been performed in a single shot, but rather in three rounds each involving roughly $\frac{1}{3}$ of the instances to inspect. At the end of each round the authors met to revise the set of labels defined up to that moment by (i) renaming unclear labels; (ii) merging similar labels, *i.e.,* labels describing the same automated task but with different wordings; and (iii) agreeing on irrelevant labels, actually indicating instances to discard (*i.e.,* unrelated to tasks supported by ChatGPT).

Once all 1,501 instances have been inspected by two evaluators, we solved conflicts. As for the relevance labeling, we found conflicts in 17% of the cases, with Cohen's $k = 0.64$, which is considered a strong agreement [57].

For what concerns the (open-coded) categories, we found differences in 380 cases (~25% of instances). While such a percentage may look high, this can be easily explained by two design choices. The first is the already mentioned lack of pre-defined categories. This implies that two evaluators defining semantically equivalent but different labels to describe an automated task (*e.g., create tests vs test writing*) would generate a conflict. The second concerns our conservative definition of conflicts: We considered an instance as a conflict if two evaluators assigned a different set of labels to the instance, *even if the two sets partially overlapped*. Conflicts also arose if one of the two evaluators discarded the instance as a false positive while the other labeled it. Each conflict has been inspected in pairs by two additional evaluators, who discussed and solved it. In the end, 467 instances were kept and classified, distributed as follows: 165 commits, 159 PRs, and 143 issues. The 467 classified instances belong to 358 projects, having [min=8, 1Q=60, median=444, 3Q=3,075, max=179,567] stars, and [min=0, 1Q=15, median=77.5, 3Q=535, max=89,252] forks.

The 45 labels defined through the above-described process have been used to build a hierarchical taxonomy of software-related tasks for which ChatGPT provided (partial) automation. Two of the authors created a preliminary version of the taxonomy which has then been refined in two rounds by collecting the feedback of all five authors involved in the labeling.

3 RESULTS DISCUSSION

Fig. 1 depicts the taxonomy of tasks automated via ChatGPT. The taxonomy is composed of seven trees, each grouping together related tasks: *feature implementation/enhancement*, *process*, *learning*, *generating/manipulating data*, *development environment*, *software quality*, and *documentation*. The numbers attached to each task T_i indicate, from the right to the left, the number of commits, issues, and PRs in which we found evidence of T_i's automation using ChatGPT. For example, we found a total of 110 instances (43 commits, 29 issues, and 38 PRs) in which ChatGPT has been used to automate the implementation or enhancement of a feature.

Note that the sum of the number of instances in all tasks is greater than the total number of valid instances we inspected (467), since one instance may have required the support of ChatGPT for multiple tasks, *e.g., generating tests* and their related *code comments*. Also, note that the number of instances in a parent category is not always the sum of the instances in its child categories. For example, consider the *software quality* → *fixing* → *supporting debugging* category: Such a task has been automated in 12 instances (11 issues and 1 PRs) and has one child category named *writing code to reproduce a bug*, automated in 3 issues. The reason for such a discrepancy is that in 12 instances it was clear that ChatGPT has been used to support the debugging process, but only in three of those cases the classification could be even more precise and refer to the specific task of helping to reproduce a bug.

In the following, we discuss the seven main categories of automated tasks by reporting qualitative examples and discussing implications for practitioners (see 💻 icon) and researchers (💡). We also showcase ChatGPT's limitations when used for the automation of the related tasks (⚠). Due to the lack of space, we do not discuss all 52 categories in our taxonomy, but only the main ones. However, our replication package [107] provides the complete dataset reporting, for each category, the instances assigned to it.

3.1 Feature implementation/enhancement

This category features tasks related to the usage of ChatGPT as a support for implementing and enhancing software features. We start by commenting two of its related, but differing, subcategories: *implementing a new feature* and *prototyping*. The former refers to the usage of ChatGPT as a support to implement a specific *part* of a feature that the developer is working on. This means that the developer delegates the implementation of a specific functionality, which is then manually integrated with the rest of the code needed for the feature. An example is the PR #37233 from the `woocommerce` project [3], in which the PR author states: "*I wrote a Python script with ChatGPT to parse csv files since we need to update this payment list quarterly*".

The latter, instead, refers to the usage of ChatGPT as a way to quickly implement either (i) a complete feature that can be used as a starting prototype for reasoning about the addition of the new feature in the project, or (ii) the whole starting version of a project, on which developers can work and build on top. An example of the first scenario is the PR #73 from the `nix-gaming` project [4] in which the contributor proposes the addition of an autoupdater script commenting "*I don't know if this is the right way to do it [. . .] Credits to ChatGPT for the script*". While this PR has been approved and merged, it is representative of those instances in which the contributor explicitly states to be unsure about what was accomplished using ChatGPT, or even declared that they were completely unfamiliar with coding while contributing PRs or issues — see *e.g.,* [5]: "*I started to edit the files with the support by ChatGPT— as said, I have no idea about coding*". 💻 Similarly to what happens when defining onboarding and contribution guidelines in open source projects [45, 61], it may be desirable to define guidelines about contributing with AI-generated code, *i.e.,* a project may decide to only welcome contributions from ChatGPT by users that are confident in assessing the correctness of the generated code.

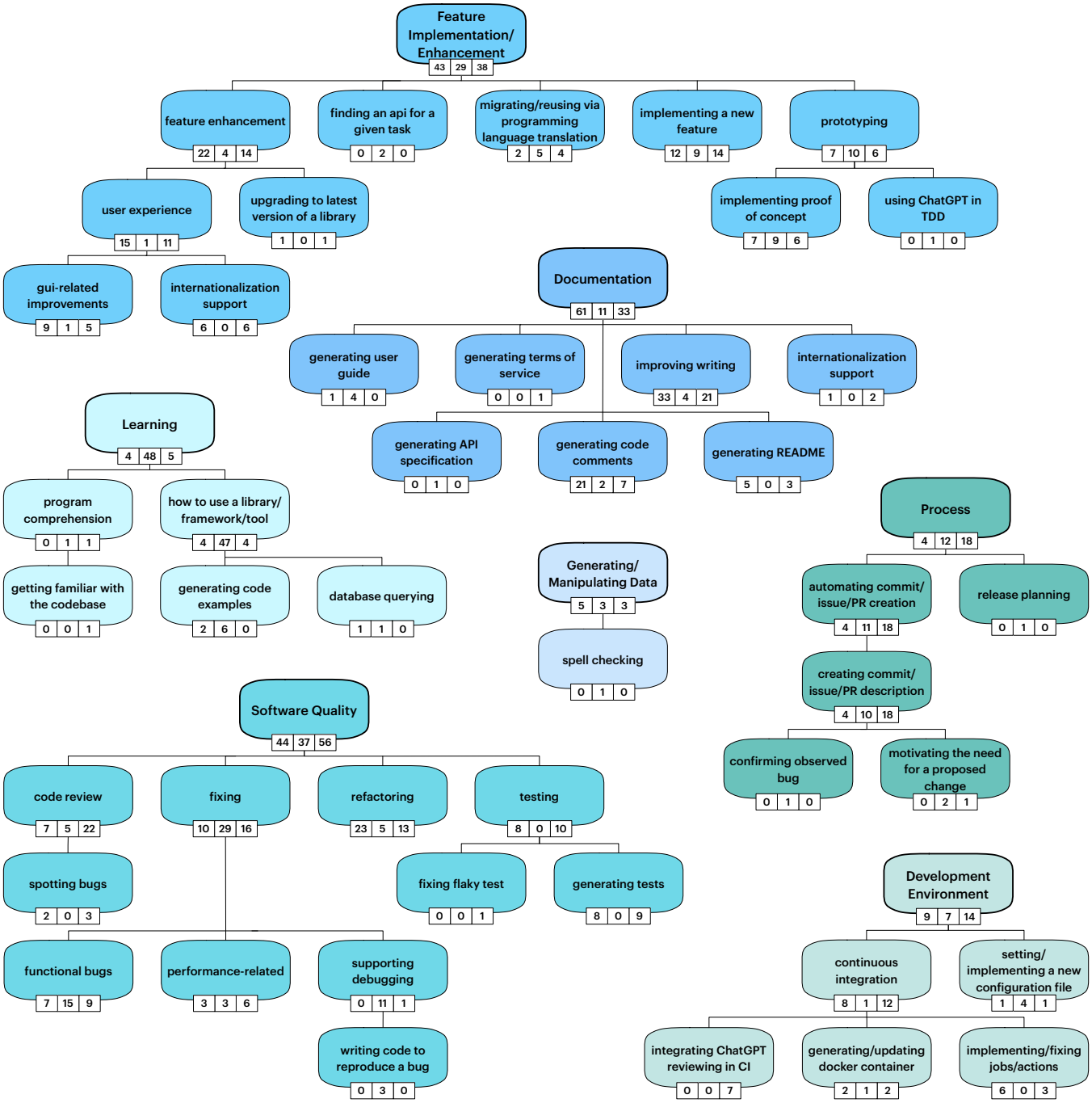

Figure 1: Taxonomy of types of tasks automated via ChatGPT

Also, projects may need to adapt the code review process, *e.g.,* relying less on (semi-) automated code quality check (*e.g.,* a linter to check code quality) when AI-generated contributions come from users having little or no programming expertise.

This finding is also relevant for research, as, for example, it may impact studies involving contributors of OSS (*e.g.,* studies on newcomers similar to those of Steinmacher *et al.* [100] or Zhou and Mockus [117]). Related studies in the future should be careful about surveying developers that have only submitted AI-generated contributions, as they may not be representative of the target population of OSS developers. Clearly, the development landscape may also significantly change, as contributors mainly relying on AI when submitting code could become the norm.

⚠ In general, there is a clear risk related to the ownership and understanding of code contributed via ChatGPT, especially when it is used to contribute complete features. Such a problem has been well-summarized in a comment of a PR we inspected [6]: "*[. . .] I want to make something clear about code suggestions done by ChatGPT: deferring to an AI bot is not the same as code ownership [. . .] the idea that an author puts some code into a commit and sends it means they should have an intellectual understanding of it. PR authors should own the code they send – ownership in the sense of being able to advocate for the code*".

The consequence is that AI-generated code may require a more thorough quality assurance, but also may lead to issue triaging problems and in general maintenance issues in the absence of a real owner. Concerning the second usage scenario for prototyping (*i.e.,* using ChatGPT to draft the whole starting version of a project), a concrete example is the issue #1 from the `apple-notes-to-sqlite` project [7] titled "Initial proof of concept with ChatGPT".

The issue documents the conversation between the repository's owner and ChatGPT, and resulted in the implementation of the first prototype of an application exporting Apple notes to SQLite. 💻 This project counts 120 stars on GitHub at the date of writing and is a concrete example of how ChatGPT can provide a jumpstart in software development.

Still related to prototyping, we also found one issue [8] in which developers discuss the possibility of using "*ChatGPT to generate reliable code through a semi-automated Test-Driven Development (TDD) process that incorporates feedback loops*" (*using ChatGPT in TDD* category in our taxonomy). 💻 From a practitioner's perspective, this would lead to a different metaphor, in which the developer is mainly in charge of writing tests letting the LLM generate code.

💡 From a research perspective, this requires empirical investigations, as it has been done in the past for conventional TDD (*e.g.,* [48, 66]) thus defining a suitable process with AI in-the-loop.

Another popular sub-category is the *feature enhancement* task, in which ChatGPT is used to enhance an already existing feature (as opposed to help contributing with a new feature). This includes generic enhancements such as writing CSS for existing web pages [42] or adding options to a feature [9], as well as more specific improvements that can be seen in the category's subtree. For example, we found 12 instances (6 commits + 6 PRs) in which ChatGPT has been used for internationalization purposes [10], mostly related to translating elements in the GUI, including error messages. ⚠ In some of these cases the reviewers asked whether the contributor was actually familiar with the target language or if, instead, they were just running ChatGPT and reporting the translation, with the risk of introducing internationalization issues. 💡 Such a finding is relevant for researchers working on detecting and fixing internationalization issues [62]. For example, ChatGPT could produce mistakes different from those typically committed by developers who manually implement internationalization.

The last sub-category we discuss is the one related to *migrating/reusing via programming language translation*, a task supported by ChatGPT in 11 of the inspected instances. 💻 Practitioners used ChatGPT to automatically translate code snippets across languages, allowing for possibilities of reuse that were unimaginable before (*e.g.,* reusing code across projects written in different languages).

An interesting example is the PR # 4559 from the `garden` project [11] in which the contributor used ChatGPT to translate from Javascript to Typescript the code of a third-party project which has not been updated in the last six years and was known to be affected by vulnerabilities. As documented by the contributor: "*I didn't find an easy fix . . . so I just created a fork of [third-party project] and asked ChatGPT to convert it to Typescript, removed the dependency on [third-party project] and later tweaked it to make sure that everything works*". 💡 Our findings confirm the relevance of research targeting the automated translation of software across programming languages [84, 86], but it also highlights the very strong performance of what should be considered the state-of-the-practice and, therefore, a baseline for comparing new approaches in this research thread. ChatGPT seems to be able to generalize across several languages, even by translating hundreds of lines of code, see *e.g.,* the Javascript to Python translation in the `textual-paint` project [12].

⚠ At the same time, recent research has pointed out perils of ML-based language translation [79], especially because the translation may not take into account that different programming languages may follow different programming paradigms (*e.g.,* object oriented *vs* functional), and the result could just be "Java with a Python syntax" or something similar. As Malyala *et al.* suggest [79], a combination of ML-based translation with static analysis and rule-based translation could be a pragmatic road to follow.

3.2 Process

The *process* category groups instances mentioning the usage of ChatGPT to support activities related to the development process, *e.g., release planning*, or the automation of steps needed to *create commits, PRs, issues, e.g.,* generating a PR description.

We found a single issue in which ChatGPT has been used to come up with ideas on how to improve a software project (*release planning* label in our taxonomy) [13]. While this is a single data point and should be taken as such, we found this usage of ChatGPT extremely interesting, since it goes substantially further than what state-of-the-art tools supporting release planning are able to do. The latter usually mine data (*e.g.,* app reviews [56, 90, 94]) to help developers summarizing the customers' feedback and come up with aspects to improve in the software. 💻 ChatGPT does not require any sort of data mining on the developer's side and, as visible from the issue we inspected [13], can be queried for general ideas about what to improve in a software or even on how to improve a specific feature/quality aspect of the software (*e.g.,* "*Can you suggest improvements to make the help system more useful for data scientists?*", "*Suggest ways to make the help system more useful for developers*"). The contributor confirmed that "*ChatGPT came up with [. . .] pretty good ideas*". ⚠ One clear limitation is that requirement crowdsourcing may require up-to-date sources of information (*e.g.,* recent information about the features that competitive software implements) which ChatGPT may not have. Also, practitioners may be afraid to prompt ChatGPT with sensitive, market-competition-related information. 💡 To cope with both issues, researchers may develop approaches based on Retrieval Augmentation Generation (RAG) [74] to combine LLMs with private or up-to-date resources.

In 32 instances developers used ChatGPT to automatically generate a commit message (*e.g.,* [14]) or a PR/issue description (*e.g.,* [15]). The automatic generation of commit messages [59, 73, 76, 110] and of PR title/descriptions [64, 71, 78] has been tackled by several researchers, especially after the wide adoption of deep learning in software engineering. While we are not aware of empirical comparisons performed between these tools and ChatGPT two observations can be made. 💡 First, the instances in our dataset show an impressive capability of ChatGPT in summarizing even complex changes spanning several files, while studies in the literature documented strong limitations of techniques for the automated generation of commit messages, mostly targeting the low-hanging fruits (*e.g., Add README*) [77]. Second, empirical comparisons should carefully consider which test instances to use, considering that the dataset on which ChatGPT has been trained is not publicly available. While this does not make it possible to ensure a lack of overlap between training and test set, an easy solution is to use very recent commits/PRs/issues as test set, since those are unlikely to have been seen by the model behind ChatGPT.

Our taxonomy also features two specializations of the *creating commit/PR/issue description* category. The first concerns a scenario in which ChatGPT has been used to confirm an observed failure as a bug: "*[failure description] asking ChatGPT suggested it to be a bug*" [16]. The second represents instances in which ChatGPT was used to better motivate a proposed change. An example of this second usage scenario is the issue #1648 from the `js-libp2p` project [17], being a feature request including a question posed to ChatGPT about the usefulness of the proposed feature ("*Can't we already do this with just a libp2p stream? Why do we need HTTP?*") with the LLM providing four disadvantages of not having such a feature.

In other cases falling in this category, the contributor just describes a chat they had with ChatGPT that helped them in coming up with the issue/PR (*e.g.,* "*this is a PR to improve the way we store thumbnails in the data folder; after a nice chat with ChatGPT I discovered why most apps do this*" [18]).

⚠ Despite the very successful applications of ChatGPT to *process*-related tasks, we observed cases of what has been recently defined as artificial hallucination [72], namely confident responses provided by an AI such as ChatGPT which look plausible to the human interacting with it but that are clearly wrong. This is reflected in negative reactions to suggestions given by ChatGPT about features to improve/implement (*e.g.,* "*I can't really do anything about that, and it would defeat the entire purpose of [project]*" [19]).

3.3 Learning

The *learning* category tree mostly features issues opened by users of a software project as a result of problems they are experiencing in using the related library/framework/tool. In doing so, they mention their attempt to solve the faced problem by asking ChatGPT (*e.g.,* [20, 21]). ⚠ This is a category of task for which ChatGPT showed clear limitations related to the previously mentioned artificial hallucination issue. In 36 out of the 47 opened issues, the indications provided by ChatGPT on how to solve the problem faced by the user were wrong, resulting in negative comments either by the user itself when opening the issue (*e.g.,* "*I even asked ChatGPT who made up some configuration options that do not exist*" [21]) or by the developers replying to the issue (*e.g.,* "*just stop asking ChatGPT about this thing, because the data that was used to train it only spans until 2021, which means everything created from 2022 (including [project]) onwards is outside of its knowledge domain; If you ask about something it doesn't know, it will make up fake answers that don't work at all, and fake libraries that don't exist*" [22]). ⚠ Similarly to release planning, leveraging outdated knowledge of LLMs can be risky, therefore they need to be complemented with alternative approaches.

The *learning* tree also features two instances in which ChatGPT has been used to understand code. In these two cases, the LLM provided useful support to the developer, even in understanding code automatically generated by a framework by reverse engineering it: "*This code is very difficult for people to read because it is compiled by webpack. However, ChatGPT completed reverse-engineering in a short time*" [23]. 💻 This shows the potential of ChatGPT in supporting program comprehension and on the research side 💡 suggest investigations aimed at assessing the impact of ChatGPT in program comprehension, as well as approaches to support the use of third-party LLMs on private code or other software artifacts.

3.4 Generating/manipulating data

Developers use ChatGPT to easily *generate/manipulate data*. The variety of data involved in this category includes strings appearing in the UI (*e.g.,* "*add ChatGPT suggestions to bully messages*" [24], "*Add extra motivational messages generated by ChatGPT*" [25]), fake data needed to fill templates (*e.g.,* "*add more fake data in the sample using ChatGPT*" [26]), or more intellectual content such as math problems for an educational project (*e.g.,* "*use ChatGPT to generate these problems and create an initial solution*" [27]).

💡 ChatGPT seems to be particularly suited in the generation of data for which correctness is not a strong requirement (*e.g.,* fake data, augmenting UI-related strings handling a dialog with the user). This makes it a suitable tool to automatically generate test inputs since even implausibly generated inputs could represent a good opportunity to assess the robustness to wrong inputs. ⚠ However, in this scenario, several risks may arise. First, LLMs can be subject to bias [51, 52] and may generate unwanted discriminatory or offensive text. Moreover, it cannot be excluded that LLMs could be subject to adversarial attacks, leading to the generation of unwanted outputs as it has been shown for other recommender systems [88].

3.5 Development environment

This tree groups together instances in which ChatGPT has been used to support and (partially) automate activities related to the *development environment*. The most popular application of ChatGPT is its *integration as reviewer in the continuous integration and delivery (CI/CD) pipeline*. In this scenario, ChatGPT is used to comment about contributed code and identify bugs and/or suboptimal implementation choices (*e.g.,* "*added bot reviewer powered with ChatGPT to help us with PR reviews*" [28]). 💻 The ChatGPT-based review is usually integrated in the continuous integration pipeline and aims at providing a first quick feedback to the contributor, without replacing (but supporting) the human reviewer. Furthermore, ChatGPT is usually combined with classic lint tools looking for issues and assessing test coverage.

💡 Such an application confirms the relevance of the recent line of research related to the automation of code review tasks [106], for which ChatGPT could become a baseline for comparison. Future research should also consider how to properly leverage LLM-recommenders such as ChatGPT to obtain code reviews in line with an organization/project's own coding styles and guidelines. ⚠ A clear issue is the need for passing code to ChatGPT, which may not be acceptable (and even forbidden) in industrial environments. In such cases, approaches leveraging local LLMs are to be preferred.

Other tasks automated via ChatGPT concern the *implementation/fixing of jobs/actions* in continuous integration scripts and the *generation/updating of docker containers*. While ChatGPT can be a good aid to draft CI/CD scripts, this is one of those tasks in which the long training time needed for LLMs and, as a consequence, their inability to be continuously retrained to be updated, represents a strong limitation. Indeed, technologies such as GitHub actions which are used to achieve CI/CD are relatively new and rapidly evolving. This resulted in PRs contributing with CI/CD scripts created with the help of ChatGPT which, accordingly to the reviewers, were using outdated actions and commands (*e.g.,* "*the actions used are outdated*" [29]). ⚠ This highlights one strong limitation of LLMs: They might not be suitable in rapidly evolving contexts such as young technologies, programming languages, etc.

💡 In these cases, it is possible that smaller, specialized models that can be quickly retrained might be more suitable and reliable.

3.6 Software quality

Software quality is the largest tree in our taxonomy in terms of number of instances (137). This indicates that developers largely leverage ChatGPT for automating tasks related to software quality improvement.

Refactoring operations recommended by ChatGPT are widely implemented by developers. This may include simple renaming [30] as well as more complex code transformations such as converting a recursive function into an iterative one: "*Thanks to ChatGPT for doing the recursion to iterative conversion*" [31]. 💡 While we found a wide variety of refactoring actions automated via ChatGPT, we observed a lack of code transformations involving multiple files, such as extract class refactoring. This is likely due to the limited view that ChatGPT has of the software systems, given its (current) lack of integration in the IDE. For such cases, approaches to generate suitable prompts helping ChatGPT to produce responses for more complex refactoring scenarios may be desirable.

Functional bugs have been fixed with the help of ChatGPT (31 instances in our taxonomy). ⚠ In these instances we observed two of the previously discussed issues affecting the usage of ChatGPT in open-source projects. First, ChatGPT has been used by inexperienced programmers to submit patches, possibly with little understanding of the contributed code: "*Was getting the error . . . so I used ChatGPT to fix it, not sure how GitHub works . . . so I gonna put it here*" [32]. Second, the inability of ChatGPT to cope with complex code, similar to what we inferred looking at the automated refactorings: *e.g.,* "*Do not trust ChatGPT to fix complex code depending on multiple files; ChatGPT has no idea of the scope nor the current state of the codebase, so it will not be able to give a valid answer . . .*" [32].

Besides the explicit bug fixes suggested by ChatGPT, the LLM is also used to *support debugging*. 💻 This mostly comes in two fashions. The first is the expected one, with a user observing a failure in a code and asking ChatGPT what was causing it: *e.g.,* "*I asked ChatGPT what this error could be, and the AI gave me an important clue . . .*" [33]. The second is the usage of ChatGPT to generate a minimal, reproducible example [34]. 💡 As of today, there are state-of-the-art approaches supporting the automated reproduction of bugs [49, 65, 116]. This is mainly possible because such approaches are (i) tailored for specific categories of applications, *e.g.,* mobile apps, and (ii) can access the whole application code base. In the future, LLM-based approaches should be therefore able to use such information to support the automated reproduction of bugs.

The *code review* subtree collects the usage of ChatGPT as a reviewer, mostly for incoming PRs. It is worth commenting on the difference between this subtree and the previously discussed *development environment → continuous integration → integrating ChatGPT reviewing in CI*. The latter concerns instances in which developers manifested their interest in integrating ChatGPT in the CI/CD pipeline as a reviewer. The former, instead, groups instances in which the outcome of a code review performed by ChatGPT was discussed, even when ChatGPT was not integrated into the workflow but queried through its user interface.

Similarly, the *code review → spotting bugs* differs from *support debugging* since in the former the developers were not aware of the bugs found by ChatGPT, while in the latter they used ChatGPT to help debugging after observing a failure.

Another, very relevant subtree of *software quality* is the one related to the automation of *testing* activities. While we found an instance in which ChatGPT was used to fix flaky tests [35], in the rest of cases (17 instances) the automated task is the generation of tests, *e.g.,* "*includes tests, which were coded with care by ChatGPT*" [36], "*tests were generated by ChatGPT and while it was not perfect it did a decent job at creating the unit tests code*" [37]. ⚠ The latter is only one of the comments we found in PRs which confirm the usefulness of ChatGPT as an aid to write tests rather than as a completely automated solution: "*I have generated the tests with the help of ChatGPT and manually checked all of them — it got a few of them wrong or was testing impossible cases, but it did find that one edge case*". 💡 On this line, research approaches could aim at integrating LLM-based test generation with approaches aimed at identifying and repairing broken tests [54, 58, 89, 101].

3.7 Documentation

The last popular application of ChatGPT we discuss (105 instances in our sample) concerns the automated generation of software *documentation*. ChatGPT is used both to write documentation from scratch (38 instances) as well as to improve existing documentation (see *improving writing* category with 58 instances). In some cases, projects' users suggest to improve parts of the documentation since they found it difficult to read: "*I found the README.md a bit difficult to digest, so I utilized ChatGPT to help me simplify the content. This allowed me to better understand the library's core features and functionality. It might be worth considering a shorter, more concise version of the README for easier comprehension*" [38].

💻 It could be useful for projects' owners, especially for non-native speakers of the language used in the documentation, to consider the usage of ChatGPT to improve documentation quality.

For what concerns the generation of documentation from scratch, ChatGPT is mostly used for *commenting code*, but also for drafting *terms of service* [39], *user guides* [40], and *README* files [41]. Differently from what we found for tasks related to code generation, we did not observe negative reactions of projects' owners/reviewers. This is likely due to: (i) the excellent performance of the LLM when dealing with natural language; (ii) the higher likelihood that the contributor posting ChatGPT-generated content has the actual competencies to assess whether the generated output is correct (*i.e.,* less coding skills required); (iii) the fact that, as ChatGPT generates natural language, projects' contributors see pretty obvious (and relatively straightforward) ways to improve/adapt it when necessary, and therefore there is less evidence of complaints; and (iv) the lower risk related to errors in the generation task (*e.g.,* typos *vs* bugs) except, of course, for terms-of-service. 💡 An empirical investigation aimed at studying the sentiment of reviewers when inspecting different types of AI-generated contributions (*e.g.,* code *vs* documentation) could help in better characterizing and backing up our observation. Last, but not least, also in this case, a proper (in some cases large) prompt may be needed by ChatGPT to generate exhaustive and correct documentation.

This, in turn, may stimulate research on how to combine LLMs with software reverse engineering approaches for that purpose.

4 THREATS TO VALIDITY

Threats to *construct validity* concern the relationship between the theory and observation. Studying the purpose of the use of ChatGPT in software development by mining software repositories has an intrinsic limitation. This is because we observe only cases where developers mention ChatGPT explicitly in a commit message, issue, or PR description. There could be other changes in which developers silently leveraged ChatGPT.

Moreover, as explained in Section 2, we analyzed the textual content of commit messages, issues, and PRs, as they could be queried by GitHub. However, there may be other places where ChatGPT could have been mentioned, *e.g.,* code comments. These would require analyzing all projects' source code and could be considered in future work.

A further threat is due to our interpretation of ChatGPT purposes of usage, by reading and labeling commits and developers' discussions. This classification could have been affected by subjectiveness and imprecision. As explained in Section 2, we mitigated this threat by having two annotators labeling each instance independently, and, after that, having a cooperative conflict resolution.

Threats to *internal validity* concern confounding factors internal to our study that could affect our results. During the manual analysis, we explicitly excluded cases in which the contribution of ChatGPT to a given development activity was unclear. Also, we used multiple labels where ChatGPT was used for multiple purposes.

Threats to *external validity* concern the generalizabiity of our findings. Within the construct validity threats stated above, the observed findings limit to open-source projects hosted on GitHub only.

Therefore, our study needs to be complemented by other types of studies (*e.g.,* interviews, survey questionnaires, ethnographic studies) conducted in closed-source scenarios, such as industrial environments. Moreover, we are only observing the first six months of ChatGPT usage, and it is possible that its variety of use will largely increase in the future. Last, this study is only limited to ChatGPT, and should be, in the future, extended to other general-purpose chat bots that could be used in software development, *e.g.,* including the recently-released Google Bard [68]. It is possible that some of them could adopt techniques to circumvent limitations/risks we found for ChatGPT or, on the other hand, have limitations that ChatGPT does not have, including the ability to access up-to-date content.

5 RELATED WORK

This section describes related work about the use of LLMs-recommenders in software development activities. We focus on studies (i) investigating the usage of ChatGPT when employed for software-related tasks; and (ii) other state-of-the-art AI-based recommenders such as GitHub Copilot [2].

5.1 ChatGPT for Software-related Tasks

White *et al.* [112] present a catalog of prompt patterns aimed at exploiting ChatGPT in three families of tasks, namely *requirements elicitation*, *system design and simulation*, *code quality*, *refactoring*. These families include a total of 14 sub-tasks (*e.g., requirements simulator*, *code clustering*).

While experimenting with different prompts, the authors show that, to date, a large human contribution is still needed to take advantage of LLMs for the automation of software-related tasks.

Nascimento *et al.* [83] compare the performance[1] of software engineers with that of ChatGPT as representative of an AI-based developer. They compared the code written by humans and by AI, finding that when it comes to simple programming tasks, ChatGPT often outperforms novice programmers. Conversely, when the complexity of the tasks increases and more experienced programmers are needed to solve them, humans perform better than the AI.

Tian *et al.* [104] investigate the potential of ChatGPT when used to automate code generation, program repair, and code summarization. They compare ChatGPT against state-of-the-art techniques showing that: (i) for code generation, ChatGPT outperforms competitive techniques, while still struggling to generalize for new problems; (ii) in program repair, ChatGPT achieves competitive performance when compared to the state-of-the-art technique; and (iii) when summarizing code, ChatGPT does not always accurately explain the intention of a given code. Some of the limitations highlighted by Tian *et al.* (*e.g.,* the difficulties experienced by ChatGPT when dealing with new likely unseen code elements) have been confirmed in our study.

Sridhara *et al.* [98] evaluate the ability of ChatGPT to solve software-related tasks such as method name suggestion, code review, and log summarization. Overall, fifteen different tasks have been considered, with ChatGPT also in this case compared against the state-of-the-art and/or against human expert ground truths.

[1]With "performance" the authors mean the ability to properly accomplish a development task.

Their results show that for most tasks the answers provided by ChatGPT are credible and sometimes better than the state-of-the-art or human expert output. However, ChatGPT poorly performs in the tasks of vulnerability detection and test prioritization.

Dong *et al.* [60] set up an experimental design aimed at mimicking human teamwork via ChatGPT: The authors assemble a virtual team composed of three ChatGPT instances instructed to cover different roles, namely analyst, programmer, and tester. The goal of the team is to carry out software analysis, coding, and testing. The experimental results show that the code generated by this "virtual" collaboration outperforms direct code generation. Moreover, the self-collaboration allows ChatGPT to address more complex tasks as compared to those achievable via direct code generation.

Our study complements the work discussed above and provides empirical knowledge about the use of ChatGPT in the open source. This, on the one hand, stimulates further empirical research focused on specific tasks that have been found relevant for developers. On the other hand, stimulates research on approaches to circumvent the clear limitations experienced by developers when using ChatGPT.

Several researchers investigated the usage of ChatGPT for automated program repair (APR). Cao *et al.* [50] focus on program repair in the context of deep learning programs, which are known to pose specific challenges for their debugging and testing. In particular, the authors analyze the capability of ChatGPT to identify faulty programs, localize the faults, and automatically repair it. Also, they show how different prompts/dialogues can have a major impact on the ChatGPT capabilities of addressing the tackled tasks.

Sobania *et al.* [95] assess the capability of ChatGPT to fix bugs in the QuixBugs benchmark. ChatGPT performances have been contrasted against the state-of-the-art techniques, showing its competitiveness against both traditional and deep learning-based APR techniques.

Fan *et al.* [63] present a study related to the previously discussed ones, but assess whether APR techniques can be used to fix erroneous solutions (code) generated by LLMs. They found that the code automatically generated by LLMs share common mistakes also present in human-written code, suggesting that APR techniques can be suitable for automated code as well.

These pieces of research confirm the potential of ChatGPT for APR that, accordingly to our taxonomy, is a task for which it has been largely leveraged by developers.

5.2 Empirical Studies on GitHub Copilot

Several studies have analyzed the use of GitHub Copilot [2]. These investigations include studies on the impact of the recommender on developers [70, 92, 109, 118], robustness assessment [81, 113, 115], empirical evaluation of correctness [87, 115], and scrutiny of security aspects [46, 91, 96].

Imai *et al.* [70] investigate the extent to which Copilot is a valid alternative to a human pair programmer. The authors involved 21 participants each of which performed coding tasks under three different treatments: (i) pair-programming with Copilot; (ii) human pair-programming as a driver; and (iii); human pair-programming as a navigator. They observed that Copilot results in increased productivity (*i.e.,* number of added lines of code), but decreased quality in the produced code.

Peng *et al.* [92] achieved outcomes consistent with the findings of Imai *et al.* [70]. In particular, they found that the group of developers with access to the AI pair programmer (*i.e.,* treatment group), completed the task 55.8% faster than the control group, *i.e.,* those who did not use the code recommender.

Vaithilingam *et al.* [109] observed instead that Copilot does not improve the task completion time and success rate. However, developers report that they value Copilot's support since it can recommend code that can be used as a starting point for the task, thus saving the effort of searching online.

Ziegler *et al.* [118] examine the relationship between Copilot usage and developers' productivity. Their findings indicate that the acceptance rate of suggested solutions can act as a reliable proxy for perceived productivity.

Nguyen and Nadi [87] provide LeetCode questions as input to Copilot to assess its ability to provide correct solutions. The study revealed significant variations in correctness among the questions related to different programming languages, ranging from 57% for Java down to 27% for JavaScript.

Yetistiren *et al.* [115] set out to evaluate the quality of code generated by GitHub Copilot. The assessment included aspects such as validity, correctness, and efficiency of the generated code. In the reported study GitHub Copilot achieved a remarkable 91.5% success rate in generating syntactically valid code. Further analyses revealed that 28.7% of the completed tasks were actually correct, 51.2% were partially correct, and 20.1% were wrong.

Wong *et al.* [113] present an investigation exploring the extent to which Copilot is capable of generating formally verifiable code. Their findings reveal that the deep learning-based code recommender successfully synthesized formally verifiable code for four out of the six problems subject of the experiment.

Mastropaolo *et al.* [81] study the robustness of GitHub Copilot when used for the automated generation of Java methods. They provided Copilot with different but semantically equivalent descriptions of the methods to generate, showing that the provided prompt can substantially change the generated recommendation. In particular, this was observed for ∼46% of methods involved in the empirical study.

Hammond *et al.* [91] assess the likelihood of receiving code recommendations from Copilot that feature security flaws. Their findings indicate that 40% of the experimented completion scenarios led to the injection of vulnerable code. In a related study, Asare *et al.* [46] investigate whether the Copilot increases the likelihood of introducing vulnerabilities as compared to manual coding. To this aim, the authors prompted Copilot to generate code recommendations in contexts in which human developers had introduced vulnerabilities in the past. The study revealed that Copilot also produced original vulnerable code in ∼33% of cases.

Sobania *et al.* [96] compare the program synthesis capabilities of GitHub Copilot with those of genetic programming techniques. They conclude that the two approaches have similar performance.

The studies discussed above mostly concern code generation, yet recently Copilot is also being used to generate other artifacts such as commit messages.

Our study, conducted with a more generic tool such as ChatGPT, indicates that code generation is only one task where developers may leverage AI-based tools.

Table 1: Summary of implications for practitioners and researchers derived from our study

Insights for Practitioners

Contributions including AI-generated content

- Define guidelines for projects' contributions including AI-generated code, *e.g.,* a project may decide to only welcome AI-generated code from users that are confident in assessing the correctness of the contributed code.
- Clear risk related to the ownership and understanding of code contributed via ChatGPT, especially when it is used to contribute with a complete feature: The (human) contributor is not always able to explain or advocate for the submitted code.
- As with any AI-based solution, the usage of ChatGPT for software-related tasks may result in artificial hallucination: AI responses that look plausible to the user can be clearly wrong. The hard skills of developers remain essential in the era of AI-assisted coding.

Automation possibilities offered by ChatGPT

ChatGPT can be leveraged to support very complex tasks, for which its usage has not been documented/experimented in the literature. These include:

- Prototyping the complete first version of a project, providing a substantial jumpstart in software development.
- TDD collaboration, where the developer is mostly in charge of writing tests and delegating to LLM the code writing task.
- Translating source code across different programming languages, thus improving code reusability.
- Release planning, suggesting ideas on how to improve a software project based on what was observed in the wild.
- Data generation, *e.g.,* augmenting UI-related strings handling dialogs with the user.
- Debugging, from several different perspectives, including helping in locating the bug as well as in reproducing it.

Software-related tasks involving natural language

Due to its extensive training on natural language artifacts, ChatGPT is well-suited to support software-related tasks strongly characterized by natural language, such as the generation of software documentation.

Risks related to sensible/private information

Some of the tasks automated via ChatGPT (*e.g.,* code review) require to pass it sensible information, such as the code base itself, which may not be acceptable in industrial environments. Practitioners must carefully consider the tradeoff of using a publicly available LLM *vs* training a local LLM.

Unsuitability of ChatGPT for tasks dealing with recent technologies

ChatGPT may not be suitable for tasks requiring up-to-date technology appeared after its last retraining. LLMs leveraging up-to-date knowledge available in the wild may obtain better results.

Insights for Researchers

Implications for the design of empirical studies

- Empirical investigations studying OSS contributors may or may not consider representative developers that only submitted AI-generated code.
- ChatGPT must be considered as a baseline in works proposing novel recommenders for tasks where it was found to be useful. However, as the dataset on which ChatGPT has been trained is not publicly available, it is hard to make a fair comparison ensuring the lack of overlap between training and test set. A possible solution is to use recent data points as test set, since those are unlikely to have been seen by the model behind ChatGPT.

Studying and enhancing AI-aided development processes

Practitioners are already leveraging ChatGPT for a variety of tasks. Nevertheless, it may be useful to (empirically) devise AI-enabled development processes, with suitable guidelines. These include using ChatGPT (or similar tools):

- In TDD, with the developer being mostly in charge of writing tests and delegating to the LLM the production code.
- To support program comprehension, especially when newcomers onboard a project and must become familiar with its code base.
- To generate tests.
- To automate code review.

Moreover, researchers should focus on approaches aimed at better integrating ChatGPT or similar tools in development contexts where there is a need for:

- Understand, refactor, complete very specific code or other artifacts.
- Avoid exposing internal artifacts to the outside, *e.g.,* using Retrieval Augmentation Generation or similar approaches.
- Repair ChatGPT-generated code and tests, or adapt them in own code base.

Questioning the suitability of existing recommenders for AI-generated code

The effectiveness of recommender systems for software engineers proposed in the literature (*e.g.,* tools identifying and fixing internationalization issues, APR techniques) may need to be reassessed on AI-generated code, since the latter may have characteristics different from those of human-written code.

6 CONCLUSIONS

In this paper, we manually analyzed, through an open coding process, 1,501 commits, issues and PRs from open source projects in which there was documented usage of ChatGPT for the automation of software-related tasks. The goal was to categorize the type of support ChatGPT provided. The result of this analysis is a taxonomy of 45 tasks (partially) automated via ChatGPT, which we discussed highlighting ChatGPT's strengths and weaknesses and distilling implications for practitioners and researchers. The latter, together with our taxonomy, represent the main outcome of our study, and have been abstracted and summarized in Table 1 for easier reference. Our future work will focus on validating our findings by (i) interviewing developers, and (ii) generalizing them to other general-purpose LLMs.

In addition, we plan to obtain evidence of the degree to which ChatGPT has proven beneficial for developers in the context of software-related task. Achieving this, however, requires the implementation of a meticulous study design with the explicit aim of mitigating biases and addressing significant issues that may arise during the analysis. We deliberately opted against undertaking this investigation by purely mining software repositories, as developers may be less prone to report cases in which usage attempts of ChatGPT turned out to be a failure.

ACKNOWLEDGMENT

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 851720). Massimiliano Di Penta acknowledges the Italian PRIN 2020 Project EMELIOT "Engineered MachinE Learning-intensive IoT system", ID 2020W3A5FY. Federica Pepe is partially funded by the PNRR DM 352/2022 Italian Grant for Ph.D. scholarships.

Tufano thanks CHOOSE for sponsoring her trip to the conference.

REFERENCES

[1] [n. d.]. ChatGPT. https://openai.com/blog/chatgpt. Accessed: 2023-03-27.
[2] [n. d.]. Copilot Website. https://copilot.github.com. Accessed: 2022-11-10.
[3] 2023. https://github.com/woocommerce/woocommerce/pull/37233.
[4] 2023. https://github.com/fufexan/nix-gaming/pull/73.
[5] 2023. https://github.com/danielgross/whatsapp-gpt/issues/68.
[6] 2023. https://github.com/typescript-eslint/typescript-eslint/pull/6915.
[7] 2023. https://github.com/dogsheep/apple-notes-to-sqlite/issues/1.
[8] 2023. https://github.com/pfusik/cito/issues/80.
[9] 2023. https://github.com/greenshot/greenshot/pull/484.
[10] 2023. https://github.com/vcmi/vcmi/pull/1659.
[11] 2023. https://github.com/garden-io/garden/pull/4553.
[12] 2023. https://github.com/1j01/textual-paint/commit/e9494ddf.
[13] 2023. https://github.com/go-go-golems/glazed/issues/50.
[14] 2023. https://github.com/fluxninja/aperture/commit/70a68635.
[15] 2023. https://github.com/talent-connect/connect/issues/658.
[16] 2023. https://github.com/gofiber/fiber/issues/2301.
[17] 2023. https://github.com/libp2p/js-libp2p/issues/1648.
[18] 2023. https://github.com/spacedriveapp/spacedrive/pull/925.
[19] 2023. https://github.com/pizzaboxer/bloxstrap/issues/224.
[20] 2023. https://github.com/spring-cloud/spring-cloud-stream/issues/2643.
[21] 2023. https://github.com/shaka-project/shaka-player/issues/5015.
[22] 2023. https://github.com/module-federation/module-federation-examples/issues/2942.
[23] 2023. https://github.com/prosyslab-classroom/cs348-information-security/issues/365.
[24] 2023. https://github.com/pbui/bobbit/commit/089fc145.
[25] 2023. https://github.com/dodona-edu/dodona/commit/9efb97f8.
[26] 2023. https://github.com/reorx/jsoncv/commit/1d5f8f1d.
[27] 2023. https://github.com/sirupsen/napkin-math/issues/26.
[28] 2023. https://github.com/hubtype/botonic/pull/2491.
[29] 2023. https://github.com/kvas-it/pytest-console-scripts/pull/76.
[30] 2023. https://github.com/kkdai/chatgpt/pull/4.
[31] 2023. https://github.com/spotlightpa/almanack/commit/955fc76b.
[32] 2023. https://github.com/kohya-ss/sd-webui-additional-networks/issues/43.
[33] 2023. https://github.com/rootzoll/raspiblitz/issues/3640.
[34] 2023. https://github.com/puppeteer/puppeteer/issues/9959.
[35] 2023. https://github.com/kiali/kiali/pull/5973.
[36] 2023. https://github.com/shilomagen/passport-extension/pull/16.
[37] 2023. https://github.com/kyverno/kyverno/pull/5834.
[38] 2023. https://github.com/failfa-st/hyv/pull/1.
[39] 2023. https://github.com/robherley/snips.sh/pull/17.
[40] 2023. https://github.com/pwncollege/dojo/issues/132.
[41] 2023. https://github.com/az-digital/az_quickstart/pull/2226.
[42] 2023. n. https://github.com/igrigorik/videospeed/issues/1035.
[43] E. Aghajani, G. Bavota, M. Linares-Vásquez, and M. Lanza. 2021. Automated Documentation of Android Apps. *IEEE Transactions on Software Engineering, TSE* 47, 1 (2021), 204–220.
[44] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting Accurate Method and Class Names. In *10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*. 38–49.
[45] Apache Software Foundation. [n. d.]. Guide for new project contributors https://community.apache.org/contributors/. Accessed: 2023-07-08.
[46] Owura Asare, Meiyappan Nagappan, and N Asokan. 2022. Is github's copilot as bad as humans at introducing vulnerabilities in code? *arXiv preprint arXiv:2204.04741* (2022).
[47] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: learning to fix bugs automatically. *ACM Program. Lang., Object-oriented Programming, Systems, Languages, and Applications OOPSLA* (2019), 159:1–159:27.
[48] Maria Teresa Baldassarre, Danilo Caivano, Davide Fucci, Natalia Juristo, Simone Romano, Giuseppe Scanniello, and Burak Turhan. 2021. Studying test-driven development and its retainment over a six-month time span. *J. Syst. Softw.* 176 (2021), 110937.
[49] Carlos Bernal-Cárdenas, Nathan Cooper, Madeleine Havranek, Kevin Moran, Oscar Chaparro, Denys Poshyvanyk, and Andrian Marcus. 2023. Translating Video Recordings of Complex Mobile App UI Gestures into Replayable Scenarios. *IEEE Trans. Software Eng.* 49, 4 (2023), 1782–1803.
[50] Jialun Cao, Meiziniu Li, Ming Wen, and Shing-chi Cheung. 2023. A study on prompt design, advantages and limitations of chatgpt for deep learning program repair. *arXiv preprint arXiv:2304.08191* (2023).
[51] Joymallya Chakraborty, Suvodeep Majumder, and Tim Menzies. 2021. Bias in machine learning software: why? how? what to do?. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 429–440.
[52] Joymallya Chakraborty, Suvodeep Majumder, Zhe Yu, and Tim Menzies. 2020. Fairway: a way to build fair ML software. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 654–665.
[53] Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2023. Neural Transfer Learning for Repairing Security Vulnerabilities in C Code. *IEEE Transactions on Software Engineering* 49, 1 (2023), 147–165.
[54] Shauvik Roy Choudhary, Dan Zhao, Husayn Versee, and Alessandro Orso. 2011. WATER: Web Application TEst Repair. In *Proceedings of the First International Workshop on End-to-End Test Script Engineering (ETSE '11)*. Association for Computing Machinery, 24–29.
[55] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2021. An Empirical Study on the Usage of Transformer Models for Code Completion. *IEEE Transactions on Software Engineering, TSE* 48, 12 (2021), 4818–4837.
[56] Adelina Ciurumelea, Andreas Schaufelbühl, Sebastiano Panichella, and Harald C. Gall. 2017. Analyzing reviews and code of mobile apps for better release planning. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER*. IEEE Computer Society, 91–102.
[57] J Cohen. 1960. A coefficient of agreement for nominal scales. *Educ Psychol Meas.* (1960).
[58] Brett Daniel, Tihomir Gvero, and Darko Marinov. 2010. On Test Repair Using Symbolic Execution. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA '10)*. Association for Computing Machinery, 207–218.
[59] Jinhao Dong, Yiling Lou, Qihao Zhu, Zeyu Sun, Zhilin Li, Wenjie Zhang, and Dan Hao. 2022. FIRA: Fine-Grained Graph-Based Code Change Representation for Automated Commit Message Generation. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Association for Computing Machinery, 970–981.
[60] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2023. Self-collaboration Code Generation via ChatGPT. *arXiv preprint arXiv:2304.07590* (2023).

[61] Eclipse Foundation. [n. d.]. Platform/How to contribute https://wiki.eclipse.org/Platform/How_to_Contribute. Accessed: 2023-07-08.

[62] Camilo Escobar-Velásquez, Michael Osorio-Riaño, Juan Dominguez-Osorio, Maria Arevalo, and Mario Linares-Vásquez. 2020. An Empirical Study of i18n Collateral Changes and Bugs in GUIs of Android apps. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 581–592.

[63] Zhiyu Fan, Xiang Gao, Abhik Roychoudhury, and Shin Hwei Tan. 2022. Automated Repair of Programs from Large Language Models. *arXiv preprint arXiv:2205.10583* (2022).

[64] Sen Fang, Tao Zhang, You-Shuai Tan, Zhou Xu, Zhi-Xin Yuan, and Ling-Ze Meng. 2022. PRHAN: Automated Pull Request Description Generation Based on Hybrid Attention Network. *Journal of Systems and Software* 185 (2022), 111160.

[65] Mattia Fazzini, Kevin Moran, Carlos Bernal-Cárdenas, Tyler Wendland, Alessandro Orso, and Denys Poshyvanyk. 2023. Enhancing Mobile App Bug Reporting via Real-Time Understanding of Reproduction Steps. *IEEE Trans. Software Eng.* 49, 3 (2023), 1246–1272.

[66] Davide Fucci, Hakan Erdogmus, Burak Turhan, Markku Oivo, and Natalia Juristo. 2017. A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last? *IEEE Trans. Software Eng.* 43, 7 (2017), 597–614.

[67] Xiang Gao, Bo Wang, Gregory J. Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction. *ACM Trans. Softw. Eng. Methodol.* 30, 2 (2021).

[68] Inc. Google. 2023. Try Bard, an AI expertiment by Google https://bard.google.com.

[69] Z. Han, X. Li, Z. Xing, H. Liu, and Z. Feng. 2017. Learning to Predict Severity of Software Vulnerability Using Only Vulnerability Description. In *33th IEEE International Conference on Software Maintenance and Evolution ICSME*. 125–136.

[70] Saki Imai. 2022. Is GitHub Copilot a Substitute for Human Pair-programming? An Empirical Study. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 319–321.

[71] Ivana Clairine Irsan, Ting Zhang, Ferdian Thung, David Lo, and Lingxiao Jiang. 2022. AutoPRTitle: A Tool for Automatic Pull Request Title Generation. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 454–458.

[72] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of Hallucination in Natural Language Generation. *ACM Comput. Surv.* (2023).

[73] Siyuan Jiang, Ameer Armaly, and Collin McMillan. 2017. Automatically generating commit messages from diffs using neural machine translation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 135–146.

[74] Patrick S. H. Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems*.

[75] Zhiyu Li, Shuai Lu, Daya Guo, Nan Duan, Shailesh Jannu, Grant Jenks, Deep Majumder, Jared Green, Alexey Svyatkovskiy, Shengyu Fu, and Neel Sundaresan. 2022. Automating code review activities by large-scale pre-training. In *30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE*. 1035–1047.

[76] Zhongxin Liu, Xin Xia, Ahmed E. Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018. Neural-Machine-Translation-Based Commit Message Generation: How Far Are We?. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*. Association for Computing Machinery, 373–384.

[77] Zhongxin Liu, Xin Xia, Ahmed E. Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 2018. Neural-machine-translation-based commit message generation: how far are we?. In *33rd IEEE/ACM International Conference on Automated Software Engineering, ASE*. 373–384.

[78] Zhongxin Liu, Xin Xia, Christoph Treude, David Lo, and Shanping Li. 2019. Automatic Generation of Pull Request Descriptions. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 176–188.

[79] Aniketh Malyala, Katelyn Zhou, Baishakhi Ray, and Saikat Chakraborty. 2023. On ML-Based Program Translation: Perils and Promises. In *45th International Conference on Software Engineering, ICSE '23, Companion Proceedings, 2023*.

[80] Ehsan Mashhadi and Hadi Hemmati. 2021. Applying CodeBERT for Automated Program Repair of Java Simple Bugs. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR*. 505–509.

[81] Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. 2023. On the robustness of code generation techniques: An empirical study on github copilot. *arXiv preprint arXiv:2302.00438* (2023).

[82] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. 2015. How Can I Use This Method?. In *37th IEEE/ACM International Conference on Software Engineering, ICSE*. 880–890.

[83] Nathalia Nascimento, Paulo Alencar, and Donald Cowan. 2023. Comparing Software Developers with ChatGPT: An Empirical Investigation. *arXiv preprint arXiv:2305.11837* (2023).

[84] Anh Tuan Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2014. Statistical Learning Approach for Mining API Usage Mappings for Code Migration. In *29th IEEE/ACM International Conference on Automated Software Engineering, ASE*. 457–468.

[85] Anh Tuan Nguyen, Tung Thanh Nguyen, Hoan Anh Nguyen, Ahmed Tamrawi, Hung Viet Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2012. Graph-based pattern-oriented, context-sensitive source code completion. In *34th IEEE/ACM International Conference on Software Engineering, ICSE*. 69–79.

[86] Anh Tuan Nguyen, Tung Thanh Nguyen, and Tien N. Nguyen. 2014. Migrating Code with Statistical Machine Translation. In *36th IEEE/ACM International Conference on Software Engineering, ICSE*. 544–547.

[87] Nhan Nguyen and Sarah Nadi. 2022. An Empirical Evaluation of GitHub Copilot's Code Suggestions. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. IEEE, 1–5.

[88] Phuong T. Nguyen, Claudio Di Sipio, Juri Di Rocco, Massimiliano Di Penta, and Davide Di Ruscio. 2021. Adversarial Attacks to API Recommender Systems: Time to Wake Up and Smell the Coffee*!*. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021*. 253–265.

[89] Minxue Pan, Tongtong Xu, Yu Pei, Zhong Li, Tian Zhang, and Xuandong Li. 2022. GUI-Guided Test Script Repair for Mobile Apps. *IEEE Transactions on Software Engineering* 48, 3 (2022), 910–929.

[90] Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado Aaron Visaggio, Gerardo Canfora, and Harald C. Gall. 2015. How can i improve my app? Classifying user reviews for software maintenance and evolution. In *IEEE International Conference on Software Maintenance and Evolution, ICSME*. IEEE Computer Society, 281–290.

[91] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2021. An Empirical Cybersecurity Evaluation of GitHub Copilot's Code Contributions. *arXiv preprint arXiv:2108.09293* (2021).

[92] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The impact of ai on developer productivity: Evidence from github copilot. *arXiv preprint arXiv:2302.06590* (2023).

[93] Martin P. Robillard, Robert J. Walker, and Thomas Zimmermann. 2010. Recommendation Systems for Software Engineering. *IEEE Softw.* 27, 4 (2010), 80–86.

[94] Simone Scalabrino, Gabriele Bavota, Barbara Russo, Massimiliano Di Penta, and Rocco Oliveto. 2019. Listening to the Crowd for the Release Planning of Mobile Apps. *IEEE Trans. Software Eng.* 45, 1 (2019), 68–86.

[95] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An analysis of the automatic bug fixing performance of chatgpt. *arXiv preprint arXiv:2301.08653* (2023).

[96] Dominik Sobania, Martin Briesch, and Franz Rothlauf. 2021. Choose Your Programming Copilot: A Comparison of the Program Synthesis Performance of GitHub Copilot and Genetic Programming. *arXiv preprint arXiv:2111.07875* (2021).

[97] Donna Spencer. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media.

[98] Giriprasad Sridhara, Sourav Mazumdar, et al. 2023. ChatGPT: A Study on its Utility for Ubiquitous Software Engineering Tasks. *arXiv preprint arXiv:2305.16837* (2023).

[99] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. 2011. Automatically detecting and describing high level actions within methods. In *33rd IEEE/ACM International Conference on Software Engineering, ICSE*. 101–110.

[100] Igor Steinmacher, Tayana Uchôa Conte, Christoph Treude, and Marco Aurélio Gerosa. 2016. Overcoming open source project entry barriers with a portal for newcomers. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016*. ACM, 273–284.

[101] Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. 2018. Visual Web Test Repair. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. Association for Computing Machinery, 503–514.

[102] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. IntelliCode compose: code generation using transformer. In *28th ACM Joint European Software Engineering Conference and the ACM/SIGSOFT International Symposium on the Foundations of Software Engineering ESEC-FSE*. 1433–1443.

[103] Patanamon Thongtanunam, Chanathip Pornprasit, and Chakkrit Tantithamthavorn. 2022. AutoTransform: Automated Code Transformation to Support Modern Code Review Process. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 237–248.

[104] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. 2023. Is ChatGPT the Ultimate Programming Assistant–How far is it? *arXiv preprint arXiv:2304.11938* (2023).

[105] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Trans. Softw. Eng. Methodol.* 28, 4 (2019), 19:1–19:29.

[106] Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota. 2022. Using Pre-Trained Models to Boost Code Review Automation. In *44th IEEE/ACM International Conference on Software Engineering, ICSE*. 2291–2302.

[107] Rosalia Tufano, Antonio Mastropaolo, Federica Pepe, Ozren Dabić, Massimiliano Di Penta, and Gabriele Bavota. 2023. Replication Package https://github.com/unveilingchatgptsusage/unveilingchatgptsusage.

[108] Rosalia Tufano, Luca Pascarella, Michele Tufano, Denys Poshyvanyk, and Gabriele Bavota. 2021. Towards Automating Code Review Activities. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE*. 163–174.

[109] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts*. 1–7.

[110] Haoye Wang, Xin Xia, David Lo, Qiang He, Xinyu Wang, and John Grundy. 2021. Context-Aware Retrieval-Based Deep Commit Message Generation. *ACM Trans. Softw. Eng. Methodol.* (2021), 30 pages.

[111] Fengcai Wen, Emad Aghajani, Csaba Nagy, Michele Lanza, and Gabriele Bavota. 2021. Siri, Write the Next Method. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE*. 138–149.

[112] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. *arXiv preprint arXiv:2303.07839* (2023).

[113] Dakota Wong, Austin Kothig, and Patrick Lam. 2022. Exploring the Verifiability of Code Generated by GitHub Copilot. *arXiv preprint arXiv:2209.01766* (2022).

[114] Xin Xia, David Lo, Ying Ding, Jafar M. Al-Kofahi, Tien N. Nguyen, and Xinyu Wang. 2017. Improving Automated Bug Triaging with Specialized Topic Model. *IEEE Trans. Software Eng.* 43, 3 (2017), 272–297.

[115] Burak Yetistiren, Isik Ozsoy, and Eray Tuzun. 2022. Assessing the quality of GitHub copilot's code generation. In *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*. 62–71.

[116] Yu Zhao, Ting Su, Yang Liu, Wei Zheng, Xiaoxue Wu, Ramakanth Kavuluru, William G. J. Halfond, and Tingting Yu. 2022. ReCDroid+: Automated End-to-End Crash Reproduction from Bug Reports for Android Apps. *ACM Trans. Softw. Eng. Methodol.* 31, 3 (2022), 36:1–36:33.

[117] Minghui Zhou and Audris Mockus. 2010. Growth of newcomer competence: challenges of globalization. In *Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 443–448.

[118] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. 2022. Productivity assessment of neural code completion. In *International Symposium on Machine Programming*. 21–29.