Usability of LLMs for Assisting Software Engineering: a Literature Review

Author:

Dania Hussein

s5dahuss@uni-bonn.de

Examiners:

Prof. Dr. rer. nat. Lucie Flek
flek@bit.uni-bonn.de

Dr. Tobias Hecking
tobias.hecking@dlr.de

September 2024

Abstract

LLMs have been employed in various fields including software engineering. Recently, there has been an increase in studies exploring their application across a broad range of software engineering tasks, including literature reviews focusing primarily on how LLMs can be exploited to optimize processes and outcomes [13][28]. This thesis, however, presents a comprehensive literature review on the usability of Large Language Models (LLMs) for assisting in software engineering tasks. Following Kitchenham's systematic review guidelines presented in [16], we collect and analyze 14 studies from 2017 to 2024 to answer the following research questions: (1) What methods have been employed to evaluate the usability of LLMs in addressing software engineering tasks? (2) Who are the primary users that can benefit from LLMs in software engineering assistance? (3) What difficulties are encountered when utilizing LLMs for software engineering support? In this review, we identify gaps in current usability studies of LLM-based tools and highlights areas needing further development, thereby pinpointing opportunities for future research.

Contents

1 Introduction

Several literature reviews have explored the use of large language models (LLMs) in software engineering [13][28]. These reviews primarily focus on identifying which LLMs are used for specific software engineering tasks. This literature review aims to investigate usability evaluation studies on LLMs in software engineering, with the goal of determining which target group finds these tools useful, identifying the challenges that users face, and providing an overview of existing usability evaluation studies on these tools.

This section outlines the research questions and the motivation behind them, along with a review of existing literature on LLMs in software engineering, highlighting how this approach differs. Section 2 provides background information on automated assistance in software engineering, explores LLMs and their applications in this field, and offers a definition of usability alongside an overview of usability evaluation methods. Section 3 analyzes the most extensive literature review on LLMs in software engineering to date, explaining how the findings motivated the decision to conduct a new systematic literature review instead of expanding the depth of the existing review, which was the first approach. Section 4 details the methodology and documentation of the review, and Section 5 presents the answers to the research questions based on the reviewed papers. Section 6 discusses these results and their limitations, leading to proposed directions for future research in section 7. Finally, Section 8 provides a summary of the entire paper.

In this thesis, the use of "we" is employed, although the research was conducted independently. Since the work has a collaborative nature, as key decisions were made in consultation with my supervisors. Additionally, this convention aligns with the standard practice I observed in the literature reviews I consulted. Therefore, I found the use of "we" appropriate to maintain a formal tone.

1.1 Task Definition

Our task is to answer the following research questions (RQs):

- **RQ1: Which usability evaluation techniques have been used to study the usability of LLMs in solving software engineering tasks?**
- **RQ2: What is the target group that can use LLMs for assistance with Software Engineering tasks?**
- **RQ3: What challenges arise when using LLMs for assistance with Software Engineering tasks?**

The initial approach was to expand the systematic literature review by Hou et al. [13] by analyzing the papers included in it and deriving answers to our research questions from those that conducted usability evaluation studies.

However, after examining the papers gathered in Hou et al.'s review [13], we found them insufficient to answer our research questions. Therefore, we decided to conduct our own systematic literature review.

1.2 Motivation

We are aware of two existing systematic literature reviews on the topic of LLMs in software engineering [13][28]. [13] collected a set of 229 papers, which constitutes the most extensive literature review conducted on this subject to date. Upon analyzing the papers gathered in the review [13], we find that none of them present a usability evaluation study on LLMs employed in software engineering. This lack of papers conducting usability evaluation studies provides us with the motivation to undertake a literature review focusing specifically on this aspect.

We aim to provide an overview of which user group is able to use LLMs to assist their work in software engineering, thereby drawing conclusions about the required academic level and level of experience. This overview can be valuable for companies, universities, or any institution aiming to incorporate large language models into their software engineering workflow. Furthermore, by identifying the challenges users currently face when using LLM-based tools for software engineering, we aim to pinpoint areas with room for improvement of the tools and language models themselves.

Additionally, we aim to identify gaps in current research on the usability of LLMs in software engineering. This is a primary reason for conducting such literature reviews according to Kitchenham [16]. By identifying these gaps, we can suggest areas for further investigation. For example, discovering which usability evaluation techniques have been applied to LLMs used in software engineering could reveal a lack of studies researching specific usability aspects, such as learnability or efficiency, thereby highlighting the need of conducting studies focusing on these aspects.

1.3 Existing Literature Reviews

The first literature review that investigates the intersection of software engineering and large language models was 'Towards an Understanding of Large Language Models in Software Engineering Tasks', published in August 2023 [28]. This Review obtained 123 research papers and provided answers to the following research questions:

- 'RQ1: What are the current works focusing on combining LLMs and software engineering?' [28]
- 'RQ2: Can LLMs truly help better perform current software engineering tasks?' [28]

In April 2024 a more extensive literature review was published with the titel 'Large Language Models for Software Engineering: A Systematic Literature

Review' [13]. This review gathered 229 papers, answering the following research questions:

- 'RQ1: What LLMs have been employed to date to solve software engineering tasks?' [13]
- 'RQ2: How are software engineering-related datasets collected, preprocessed, and used in LLMs?' [13]
- 'RQ3: What techniques are used to optimize and evaluate LLM for software engineering?' [13]
- 'RQ4: What software engineering tasks have been effectively addressed to date using LLM for software engineering?' [13]

These literature reviews primarily focused on categorizing and analyzing the types of large language models (LLMs) used in software engineering, the methods used to collect and preprocess datasets for these models, and the optimization and evaluation techniques employed on LLMs for software engineering. Despite these comprehensive investigations, the reviews did not extensively consider the practical usability or real-world effectiveness of LLMs in everyday software engineering tasks, concentrating instead on the technical and methodological aspects of integrating LLMs into the field.

2 State of the Art

This section offers an overview of key developments in automated assistance for software engineering, exploring tools not related to large language models, the evolution and application of large language models (LLMs), and an explanation of usability along with the usability evaluation methods.

2.1 Large Language Models

Early approaches like n-grams and basic neural networks have laid the road for the development of large language models (LLMs). These were the first models to capture simple language patterns and were used, for example, in text categorization. However, they were limited by their short context length. [6] Recurrent Neural Networks (RNNs) followed, introducing the ability to process sequences of arbitrary length by updating hidden states, though they still struggled with long-range dependencies [21]. To overcome these challenges, encoder-decoder architectures were introduced, which significantly improving sequence-to-sequence tasks like translation, but they were still constrained by the inefficiencies of sequential processing [23].

The breakthrough came with the introduction of the transformer architecture by Vaswani et al. in 2017 [26]. Transformers utilize a self-attention mechanism, allowing the model to weigh the importance of different words in a sentence based on all other words in the sequence. This parallel processing capability not

only improved computational efficiency but also enhanced the model's ability to capture context over long distances within the text. Self-attention enables transformers to dynamically focus on relevant parts of the input sequence,attending to tokens based on their context. This mechanism underpins the remarkable performance of models like BERT (Bidirectional Encoder Representations from Transformers) and GPT (Generative Pre-trained Transformer), which have set new benchmarks in various NLP tasks [26].

Large Language Models (LLMs) employing a transformer architecture represent a significant leap in natural language processing, distinguishing themselves from previous technologies through their scale and versatility. Unlike earlier models, which were often constrained by limited data and required extensive retraining for each specific task, LLMs are trained on vast datasets with billions of parameters. This extensive training allows them to process and generate human-like text across a wide array of topics. [3]

One of the most groundbreaking features of LLMs is their ability to perform few-shot learning. This means that, instead of needing large amounts of task-specific data, LLMs can adapt to new tasks with just a few examples. For instance, they can translate text, summarize documents, or engage in complex reasoning with minimal additional input. This capability allows them to generalize across different tasks without the need for the extensive fine-tuning that earlier models required, making them far more flexible and efficient. [3]

LLMs' ability to perform complex tasks with minimal additional training makes them especially useful in software engineering. Unlike previous models that required extensive task-specific data and fine-tuning, LLMs can adapt to various software engineering tasks, such as code generation, debugging and documentation by understanding and generating code or technical text from just a few examples. [3]

Furthermore, the scalability and accessibility of LLMs mean that they can be quickly deployed across various industries, providing powerful AI solutions without the need for specialized machine learning expertise. This has made advanced AI more accessible and usable, enabling businesses and developers to leverage AI for a broad range of applications with greater ease and efficiency. [3]

2.2 Software Engineering Assistance

Automated assistance in software engineering has a rich history that predates the advent of large language models (LLMs). This overview addresses both traditional tools and LLM-based tools, as well as the fields where they have been applied.

Pre-LLM Assistances One notable example is static code analysis tools like SonarQube, Veracode, Snyk, and Checkmarx. These tools analyze code without executing it, identify potential bugs, security vulnerabilities, and code smells. They scan codebases to detect patterns that may lead to issues, offering devel-

opers possibilities to enhance code quality and security early in the development process [5][22].

Additionally, code completion tools like IntelliSense, found in many IDEs, parse the code as the user types, building an internal representation of the code's structure and context. It uses this representation to provide real-time suggestions, completions, and documentation based on the current context. [25]

Automated test generation has also been around well before LLMs. Tools such as EvoSuite use genetic algorithms to automatically generate test cases for Java programs. EvoSuite examines the code structure and behavior to create effective test suites, ensuring high coverage and uncovering edge cases that might not be evident through manual testing.[24]

Moreover, tools like Unittestscribe generate documentation for unit test cases using natural language processing. Unittestscribe combines static analysis, NLP, backward slicing, and code summarization techniques to generate natural language documentation for unit test cases, enhancing understanding and collaboration among developers.[17]

In summary, even before the emergence of LLMs, robust solutions were provided for code analysis, code completion, test generation, and documenting code. These technologies have significantly streamlined software development processes, improved code quality, and boosted overall productivity.

LLM Assistances The literature review by Zheng et al. [28] shows that LLMs have been utilized in a variety of software engineering tasks. Specifically, LLMs have shown to be effective in code generation, enabling the automatic generation of code snippets based on high-level descriptions in natural language or partial code inputs. In code summarization, LLMs help generate concise and accurate descriptions of what a particular piece of code does, which aids in documentation and code understanding. Code translation involves transforming code from one programming language into another, where LLMs facilitate cross-language interoperability. LLMs are also employed in vulnerability detection to identify potential security weaknesses in code, thereby helping improve software security. In code evaluation, LLMs assist in assessing code quality and functionality. Additionally, LLMs support code management tasks, such as version control and repository organization, and enhance Q&A interaction by providing intelligent responses to developer queries.

The more extensive literature review by [13] offers a broader and deeper perspective on the application of LLMs in software engineering. It reveals that LLMs are predominantly used in software development and software maintenance. Within software development, code generation and program repair are the most prevalent tasks. LLMs facilitate the generation of new code and automatic debugging, significantly reducing development time and improving code reliability. In software maintenance, LLMs assist in activities such as refactoring and updating legacy code, which are crucial for maintaining software quality over time.

Furthermore, the authors of the literature review [13] highlight that LLMs

are also employed in software quality assurance, where they contribute to automated testing, bug detection, and code review processes. This application enhances the accuracy and efficiency of identifying and fixing defects, leading to higher-quality software products. In requirements engineering, LLMs help in extracting, tracing, and validating requirements from various sources, ensuring that the software satisfies user needs and meets regulatory standards. In software design, LLMs assist in architectural decision-making and design pattern recommendations, which streamline the design process and improve the overall architecture of software systems. Lastly, in software management, LLMs are used for project management tasks such as effort estimation, resource allocation, and timeline prediction, supporting better planning and execution of software projects.

According to these reviews, LLMs have a transformative impact across the entire software engineering life cycle, from initial requirements gathering to final deployment and subsequent maintenance. This promises that by leveraging LLMs capabilities, software engineers can achieve higher productivity, enhanced code quality, and more efficient project management, leading to more robust and reliable software solutions. However, these reviews did not provide detailed statements based on usability evaluation studies of LLMs in these areas.

2.3 Usability

Usability is defined as the extent to which a system, product, or service can be used by specific users to achieve certain goals. Usually, usability is measured in regard to three metrics: *effectiveness*, *efficiency*, and user *satisfaction* [1]. Researchers have developed a range of usability evaluation methods to assess different aspects of software systems' usability. These methods are generally divided into three categories: inspection methods, testing methods, and inquiry methods [11]. In the following, we provide an overview of usability metrics and evaluation methods.

Usability Metrics *Effectiveness* measures how accurately and completely users achieve their goals with minimal errors. High effectiveness means tasks are completed correctly and fully with minimal errors or frustration. Current usability standards emphasize metrics like task success rates and error rates to measure the effectiveness of systems [1].

Efficiency assesses the resources users expend, including time and effort, to achieve their goals. Efficient systems enable quick and easy task completion. Usability evaluations often use metrics like task completion time and user effort. [1].

Satisfaction includes positive attitudes, emotions, and comfort resulting from using a system, emphasizing user experience (UX). Modern evaluations use qualitative feedback, user interviews, and sentiment analysis to evaluate overall satisfaction [1].

In summary, the state of the art of usability is characterized by a user-centered approach that prioritizes effectiveness, efficiency, and satisfaction. The

integration of advanced measurement tools and methods, alongside updated standards, ensures that usability assessments are aligned with the evolving expectations of users. This approach is essential for designing systems, products, and services that not only meet functional requirements but also provide a positive and engaging user experience.

Usability Evaluation Methods Inspection methods focus on evaluating the user interface itself, testing methods concentrate on task performance and inquiry methods focus on gathering user feedback, collecting subjective impressions and experiences about the software. [11]

Inspection methods involve evaluators examining a user interface to determine its compliance with established guidelines or design principles. Examples include cognitive walkthrough and heuristic evaluation. In a cognitive walkthrough, evaluators follow specific tasks to assess how easily users can learn and understand the interface. Heuristic evaluation, on the other hand, involves experts independently reviewing the interface using a predefined set of heuristics to identify potential usability issues. [11]

Testing methods are designed to observe how users interact with software, focusing on task performance. These methods include performance measurement, which gathers quantitative data on how effectively users complete tasks. Another approach is the thinking aloud protocol, where users verbalize their thoughts while interacting with the software, offering insights into their cognitive processes. [11]

Finally, inquiry methods gather feedback directly from users to understand their subjective experiences with the interface. These methods include interviews, where evaluators ask users questions to discuss their experiences, and questionnaires, which collect quantitative data on user satisfaction. Surveys are another common approach, allowing evaluators to gather data from a large number of users quickly, often through predetermined questions sent via various communication channels. [11]

These methods, each with its specific focus and approach, provide comprehensive insights into the usability of software, helping to create more user-friendly, effective and efficient products.

3 Analysis of existing Systematic Literature Review

We examine the 229 papers gathered by Hou et al. in their systematic literature review [13] to identify studies addressing usability. Our goal is to collect papers that conduct usability evaluation studies on LLMs in software engineering and analyze them to answer our research questions.

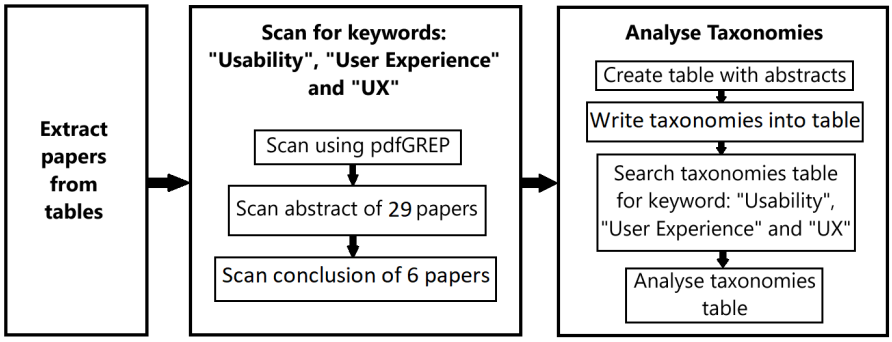

Figure 1: Steps of the analysis of the systematic literature review on LLMs for assisting software engineering by Hou et al. [13].

3.1 Method of Analysis

We extract the papers from Tables 6, 7, 8, 9, 10, 12, and 13 in the results section of the literature review[13]. We download these papers as pdf files and scan them for the keywords "Usability", "User Experience", and "UX" using pdfGREP. We manually scan the abstracts of papers where our keywords appear. If the abstract does not provide enough information, we further examine the conclusions to determine whether the papers conduct a usability evaluation study.

Next, we perform an analysis on the topics included in the papers using an internally developed tool at DLR. This tool analyzes text strings and provides an *OpenAlex* taxonomy categorization as output. *OpenAlex* is a comprehensive, open-access catalog of scholarly papers, authors, institutions, and research topics, serving as a free alternative to proprietary databases like *Scopus* and *Web of Science*. A key feature of *OpenAlex* is its taxonomy system, which hierarchically categorizes scholarly works into various fields of study. The taxonomies in *OpenAlex* are organized into levels, starting with broad fields at Level 0. At higher levels, the categories become increasingly specific, drilling down into subfields that represent more precise areas researched in the f scholarly paper.

We automated this analysis by creating a table with the titles and abstracts of the papers. We used the abstracts as input for our tool, which extracted the taxonomies from these abstracts.

Source Code 1 in the appendix shows the code we used for extracting the taxonomies. We offer the resulting table in the appendix, additionally, we create a table summarizing the most frequently occurring taxonomies and offer it in the next subsection. Additionally, figure 1 shows the steps we followed analysing the systematic literature review [13].

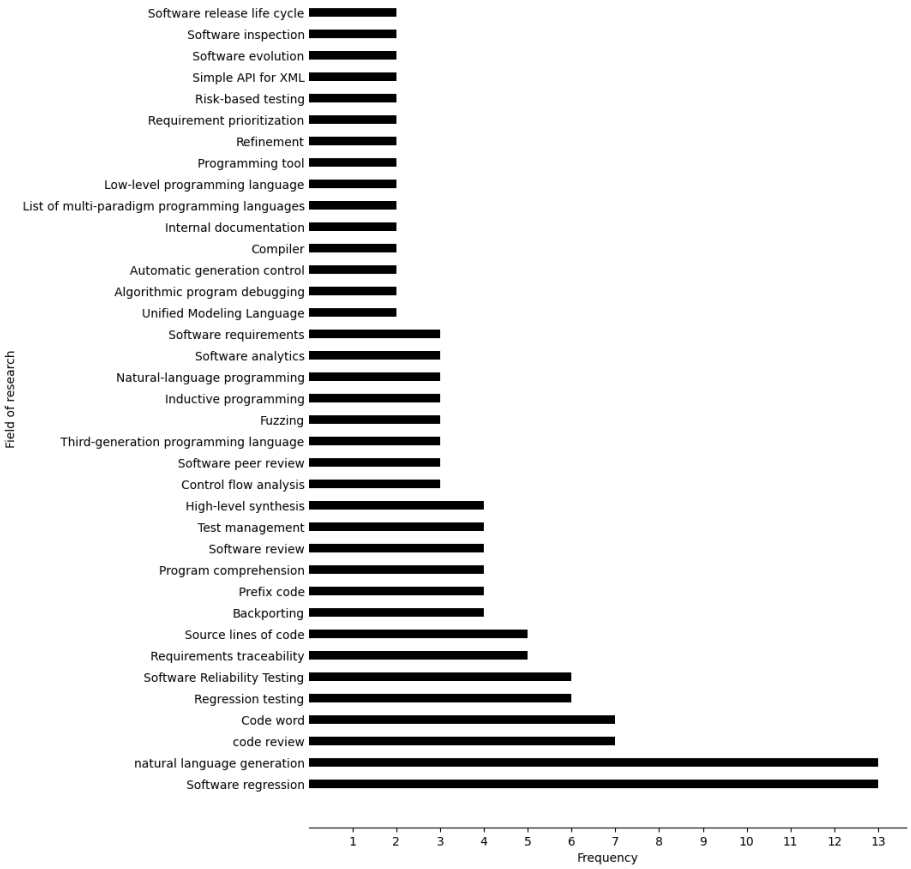

Figure 2: Taxonomies observed on the deepest level and their frequency, leaving out taxonomies that appeared one time.

3.2 Results of Analysis

89% of the studies gathered in the most extensive literature review about LLMs assistance in software engineering -[13]- did not conduct a usability evaluation study.

We found 203 of the 229 papers examined in the review [13], , accounting for 89% of the total, through extracting the papers from the tables in the results section. The residual 26 papers were not mentioned in these tables. In 29 of the 203 papers, a subset of the words "Usability", "User Experience", and "UX" appeared more than 8 times. We manually scanned the abstracts of these 29 papers and found that 23 papers did not address usability in a manner relevant to our research. These papers briefly mentioned that the LLM-based tools were found useful or that usability was considered in the design of the user interfaces. However, they did not include any usability evaluation studies or provide a detailed explanation of the usability of the design. We further examined the conclusions of the remaining six papers, where the abstract was not enough to draw conclusions about the papers' relevance for our RQs. We found that these papers did not include usability evaluation studies either and therefor do not provide answers to our RQs.

Regarding the taxonomies categorization 38 abstracts did not result in any output. Accordingly, we extracted the taxonomies categorization of 165 papers researching LLMs in software engineering. The resulted taxonomies were written in a table, which we offer in the appendix under Table 5. The keywords "Usability", "User Experience", and "UX" were not found in the resulted table, showing that the papers did not research the usability of the tools under investigation. Of the 165 papers, 81 papers provided categorizations in 6 taxonomy levels, 20 papers in 5 taxonomy levels, 50 papers in 4 taxonomy levels, and 14 in only 3 taxonomy levels. This indicates the significant variation in the range of researched topics, thoroughness, and complexity of the studies.

According to the categorization, the most dominant research fields related to LLMs in software engineering appear to be *natural language generation* and *software regression* (each researched 13 times), followed by *code review* and *code word* (each researched 7 times), and *regression testing* and *software reliability testing* (both researched 6 times). Figure 2 offers an overview of the deepest taxonomy levels found in the papers, showing only topics researched more than once. We consider the last taxonomy level found in each paper to indicate the actual researched topic, given the hierarchical nature of the taxonomy categorization.

Table 5 shows the full resulting categorization. We also offer a summary of extracted taxonomies in table 1, listing the most frequently observed taxonomies.

In the summary table 1, we note that 8 occurrences of the taxonomy *natural language generation* as the deepest taxonomy follow this categorization: *computer science* at level 0, followed by *artificial intelligence* and *natural language processing* at level 1, and then *natural language generation* and *natural language* at level 2, ending with *natural language generation*. This indicates a lack of vari-

ety and thoroughness in papers researching *natural language generation*, which is expected since the term appears very general and obvious when researching LLMs.

On the other hand, 3 occurrences of *software reliability testing* follow this categorization: *computer science* at level 0, followed by *operating system* and *programming language* at level 1, *software* at level 2, *software system* and *software development* at level 3, followed by *software construction* and *software quality* at level 4, and ending with *software reliability testing*. Other 2 occurrences follow this categorization: *computer science* and *engineering* at level 0, followed by *operating system* and *software engineering* at level 1, *software* and *dynamic testing* at level 2, *software system* and *software development* at level 3, *software construction* and *software quality* at level 4, leading to *software reliability testing* at level 5. This shows greater variety in research related to textitsoftware reliability testing in the context of LLMs in software engineering.

Furthermore, we found only 2 occurrences of *software regression* as the deepest taxonomy in our summary, following this categorization: *computer science* at level 0, followed by *operating system* and *programming language* at level 1, *software* at level 2, *software system* and *software development* at level 3, *software construction* and *software quality* at level 4, and ending with *software regression* at level 5. This indicates that *software regression* results from many other categorizations, showing the variety in research around that topic.

Lastly and most importantly, we did not notice any appearance of usability in the taxonomies. This confirms the lack of usability evaluation studies in the papers gathered in the literature review [13]. Therefore, we find it necessary to conduct our own literature review on this topic.

3.3 Discussion

During our analysis, we encountered a significant challenge in identifying all the papers included in the literature review [13]. We successfully located 89% of the papers, but 11% remained missing. The missing papers were not listed in the tables in the results section of the literature review [13], which made the process of finding them time-consuming and required thorough research. This limitation raises the concern that we may have overlooked papers relevant to our research questions. The absence of these papers potentially weakens our conclusion that no studies conducted a usability evaluation. If any of the missing papers did address usability, their exclusion could have affected our findings and conclusions. Despite this limitation, and considering the scope of a bachelor thesis, the analysis of the 203 papers we did locate still reveals a notable lack of usability evaluation studies. This gap is significant enough to warrant conducting our own literature review.

Additionally, we encountered a problem with extracting the taxonomies from the papers. Specifically, 38 abstracts did not produce any output when ran through the code used for taxonomies extraction. This technical issue means that the taxonomies we were able to identify represent only 165 papers out of the 229 originally gathered in the literature review [13]. As a result, our

Taxonomy 0	Taxonomy 1	Taxonomy 2	Taxonomy 3	Taxonomy 4	Taxonomy 5
computer science (92)	artificial intelligence, natural language processing (17)	natural language generation, natural language (10)	natural language generation (10)	software construction (1)	Software walk-through (1)
				second-generation programming language (1)	
	operating system, programming language (39)	software (28)	software system, software development (14)	software construction, software quality (7)	code review (1)
					Software regression (2)
					Software Reliability Testing (3)
					Test management (1)
computer science, mathematics (30)	statistics (21)	Decoding methods, coding (11)	variable-length code, Code word (7)	linear code, Concatenated error correction code (1)	prefix code (1)
				component-based software engineering, software construction, software development process, software design (1)	search-based software engineering (1)
computer science, engineering (16)	operating system, software engineering (7)	software, Dynamic testings (4)	software system, software development (4)	test-driven development, software construction, software quality, software development process (1)	Development Testing, Test management (1)
				software construction, software quality (2)	Software Reliability Testing (2)
				component-based software engineering, software construction (1)	Feature-oriented domain analysis (1)

Table 1: Summary of most frequently occuring taxonomies, the numbers refer to the frequency of appearance in the specific level.

categorization covers only 72% of the total papers. This limitation must be considered when interpreting our findings, as the missing 28% could contain important variations or additional insights that are not captured in our current analysis.

In summary, these two issues -the inability to identify all the relevant papers and the failure to extract all taxonomies from the abstracts- pose challenges to the completeness and reliability of our results. Despite these challenges, the findings provide a substantial overview of the field of research.

4 Self-implemented Systematic Literature Review

We conduct a literature review following the method of Kitchenham [16]. Therefore we perform a search based on keywords related to usability, LLMs and NLP, and software engineering in the following digital libraries: IEEE Xplore, ACM, Web of Science, arXiv, and ScienceDirect. The specific search queries are stated in section 4.1. The search results in 353 papers, of which we exclude 339 according to exclusion criteria listed in section 5. The remaining 14 papers were read and evaluated based on our research questions (1.1). Our findings are presented in the results section 4.3.

4.1 Literature Search

The search was conducted in the following databases: ACM Digital Library, IEEE Xplore Digital Library, ScienceDirect, Web of Science, Semantic Scholar, and arXiv. According to Kitchenham, IEEE Xplore, ACM Digital Library, and ScienceDirect are recommended electronic sources for software engineering [16]. Additionally, Google Scholar was considered relevant by Kitchenham in their guidelines [16]. However, we found it challenging to refine searches within Google Scholar due to the high volume of results. Therefore, while we did not use Google Scholar for our main search, we utilized it to identify other suitable digital libraries. The results from Google Scholar motivated us to conduct further searches in Web of Science, Semantic Scholar, and arXiv.

For our search, we chose keywords from each of the domains Usability, LLMs, and Software Engineering with subcategories and spelling variations.

We use the following keywords:

- usability, user experience, usability testing, heuristic evaluation, cognitive walkthrough, survey, questionnaire.
- LLMs, LLM, Large Language Model, Language Model, Natural Language Processing, NLP, GPT.
- Software Engineering, SE, Software Development, Software Testing, Software Design, code, program.

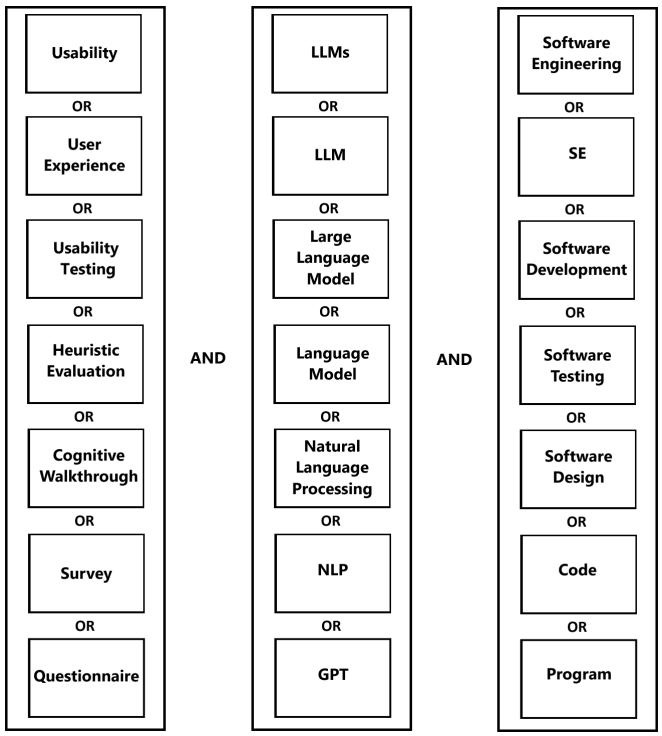

Figure 3: Query used to search for relevant papers in the digital libraries: ACM Digital Library, IEEE Xplore Digital Library, Web of Science, Semantic Scholar, and arXiv.

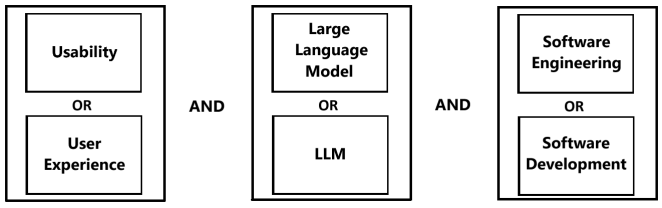

Figure 4: Query used to search for relevant papers in Science Direct.

Digital Librarys	Documentation
IEEE Xplore	Search strategy: Advanced search of Abstracts Date of search: 23.02.2024
ACM	Search strategy: Advanced search of Abstracts Date of search: 26.02.2024
Web of Science	Search strategy: Advanced search of Abstracts Date of search: 18.03.2024
Semantic Scholar	Search strategy: Search bar of the database in all fields Date of search: 12.03.2024
arXiv	Search strategy: Advanced search of Abstracts Date of search: 06.03.2024
ScienceDirect	Search strategy: Advanced search in field "Find articles with these terms" Date of search: 07.03.2024

Table 2: Documentation of literature search, showing the strategy and date of search in each digital library.

We limit our search to studies published from 2017 onwards, as the introduction of the transformer architecture in 2017 accelerated the advancement in research related to LLM-based tools and their application across various fields. [26][3]

We use the advanced search function of each library to enter the search query, where we only search through abstracts. We connect keywords from each Domain with the OR operator, resulting in three queries. We connect these with the AND operator, resulting in the query shown in figure 3. The database Science Direct requires an adapted query, since the amount of logical operators allowed in the search query is limited. Therefore the search query for Science Direct was limited to the query shown in figure 4. Table 2 shows the search strategy used in each Database and the Date, when the search is conducted.

In total we find 353 Papers. 27 papers in ACM Digital Library, 31 in IEEE Xplore, 42 in Science Direct, 69 in Web of Science, 11 in Semantic scholar and 173 in ArXiv.

4.2 Literature Selection

We exclude papers based on two types of criteria: Technical and content-related.

Technical criteria

-Links leading to empty papers.
-Papers not written in English.
-Duplicated papers.

Content-related criteria

-Non primary studies.

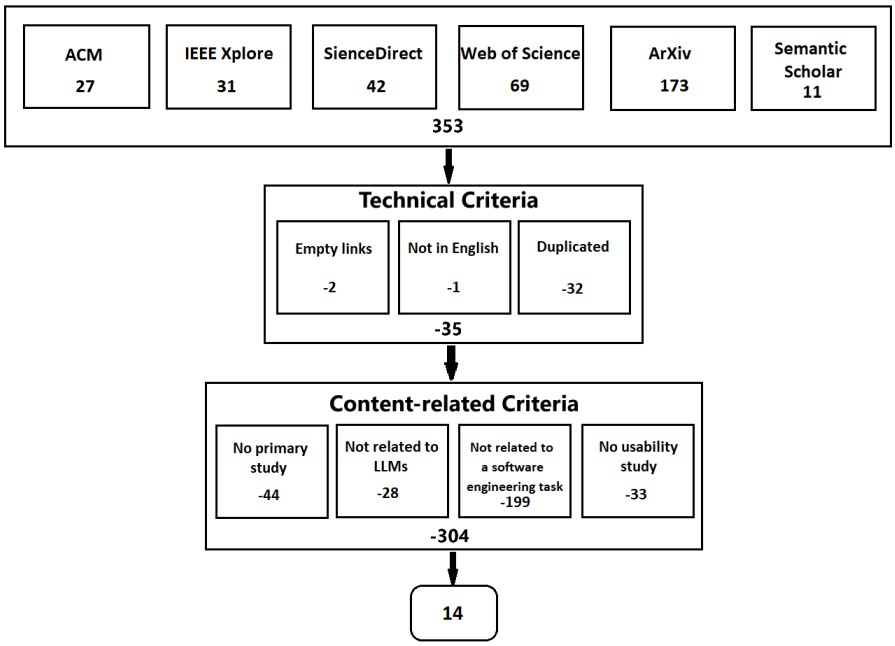

Figure 5: Number of papers found in each digital library, and papers excluded according to each technical and content-related criteria.

-Papers not researching LLMs.
-Papers not researching utilizing LLMs in Software Engineering/ Development tasks.
-Papers not researching the usability of LLMs in solving these tasks.

Two results linked to empty websites. One paper not written in English is found. These were excluded, resulting in 350 papers. 44 papers were non-primary studies and 32 were duplicated results. After excluding these, 274 papers were left. In the next step, papers that were not matching out research topics were excluded. 28 papers did not research LLMs, 199 papers, did not research applying LLMs in software engineering-related tasks, and 33 papers did research applying LLMs in software engineering-related tasks, but did not conduct any usability evaluation study. After excluding these papers, we had a set of 14 papers left. Figure 5 shows the number of papers found in each library, and the number of excluded papers according to each criteria.

4.3 Data Analysis

ChatGPT occurred in 3 papers. While only one paper conducts a usability evaluation study on chatgpt [12], two others ([27] and [24]) compare applying ChatGPT to newer LLM-based tools, namely CHAT-TESTER and AthenaTest,

for generating unit test cases.These newer tools are modified specifically for generating unit test case. CHAT-TESTER is based on ChatGPT but contains an interactive test refiner that fixes the compilation errors in tests generated by ChatGPT, AthenTest on the other hand is based on BART architecture.

Two papers conduct usability evaluation studies on copilot: [25] and [7]. [25] compares using Copilot for code generation with using Intellicense . Furthermore, Copilot appears in another usability evaluation study conducted on a conversational Software Engineering assistance tool [20]. Some questions in the paper [20] compared *The programmer's assistant* with Copilot.

Some papers we include have an educational tribute, e.g. [15] and [8]. Since they cover part of the software engineering process, we include them in our review. The tool USQA covered in [15] can also be utilized by engineers to help writing high quality user stories and hereby is not purely educational. Furthermore, we include the study [8] on ART (Automated Review Tool) since code reviewing is also an important task in software development. One tool that resulted from our search is SOCIO [18], [19]. SOCIO is a chatbot that doesn't use a large language model, but applies traditional NLP techniques. We include SOCIO since it is used for an important software development task, namely UML modelling. No other LLM-based tool we found is used for this task. Furthermore, the paper [18] uniquely presents a family of experiments, an evaluation method not performed by any other paper. Therefor, including the papers [18] and [19] adds to the variety of usability evaluation techniques.

Table 3 provides an overview on tools mentioned in the papers included in this review as well as the tasks they primarily solve and the LLMs they are based on. All tools utilize a transformer based LLM. Most tools are used for the generation or completion of code (5 out of 14). The next most prevalent task is unit test case generation (2 out of 14).

5 Results

We provide answers to our research questions based on the reviewed papers. The following sections present detailed data related to each research question, with a summary answer provided at the end of each section.

5.1 Evaluation Techniques

In this section, we answer *RQ1: Which usability evaluation techniques have been used to study the usability of LLMs in solving software engineering tasks?* We do this by identifying the types of usability evaluation studies conducted and reviewing the techniques employed to evaluate each usability criterion.

5.1.1 Types of usability evaluation studies

In 13 of the 14 papers a survey is conducted. The study about GenLine [14] undertakes interviews. Paper [4] on AtomXR includes both, surveys and interviews. In most studies [14][8][25][19][18][2][9][4][7][20][27][15] participants are

Tool	Task	LLM	Studies
ChatGPT	Mostly used for generating source code or detecting defects	Generative Pre-trained Transformer (GPT)	[12]
Copilot	Code completion	Generative Pre-trained Transformer (Codex)	[25][7]
GenLine	Code generation	Generative Pre-trained Transformer (GPT)	[14]
ART	Authoring code reviews	Generative Pre-trained Transformer (GPT-3, GPT-4)	[8]
Instigator	Searching for GUI layouts	Generative Pre-trained Transformer (GPT-3)	[2]
CodeCompose	Multi-line code authoring	Decoder-only transformer (InCoder)	[9]
AtomXR	Streamlined XR prototyping	Generative Pre-trained Transformer (GPT-3)	[4]
The Programmer's Assistant	Code generation	Generative Pre-trained Transformer (Codex)	[20]
Chat-Tester	Unit test case generation	Generative Pre-trained Transformer (ChatGPT)	[27]
USQA	User story quality analyzing	Bidirectional Encoder Representations from Transformers (BERT)	[15]
AthenaTest	Unit test case generation	BART (Bidirectional and Auto-Regressive Transformers)	[24]
SOCIO	UML modelling	NLP only	[19][18]

Table 3: Tools found in literature, the tasks they are used for and the LLM they are based on.

required to complete tasks using the tool under investigation before participating in the surveys or interviews. This is not the case for the surveys on ChatGPT [12] and AthenaTest [24]. In [12] the survey included questions about ChatGPT usage habits, satisfaction levels, and challenges faced while using ChatGPT. In [24] participants compare unit test cases generated by AthenaTest, ChatGPT, and EvoSuite.

The study [7] on copilot conducted surveys over the period of 4 weeks, where participants had to solve up to 12 challenges and submit surveys about each challenge right after solving it.

7 studies compared already established tools with new introduced LLM based ones. These are the following:

Paper [25] compared Copilot with Intellisense through a within-subject comparative study, where participants complete python programming task using Copilot and another one using Intellisense.

The papers [19] and [18] compare the Chatbot SOCIO with Creatly. Paper [19] conducts a Within-subject crossover user study dividing participants into two groups. One group is asked to solve python programming tasks using one tool first and then the other, while the second group solves the same tasks using the tools in the opposite order. Paper [18] extends that study by performing a family of experiments, using the study in [19] as a baseline experiment and conducting two identical replications.

Paper [4] compares three tools through a within-subject and across-subject study. Participants complete a series of tasks using the Unity development system first and using AtomXR running on desktop next. They then complete a series of tasks using AtomXR running on desktop first and then using AtomXR running on a headset next.

Paper [27] compares tests generated by ChatGPT and CHAT-TESTER, where participants evaluate the tests without knowing which tool generated which test.

Paper [24] compares test cases generated by AthenaTest, ChatGPT, and EvoSuite.

Paper [9] involves experiments on tens of thousands of engineers, exploring their perceptions of CodeCompose and comparing their experiences with it to single-line suggestions.

In 5 studies, data was recorded during task solving:

In [25] the audio and the screen-cast was recorded, while participants used Copilot to solve the tasks. These session recordings were later analyzed to identify the root cause of task failures.

In [19] and [18] Telegram conversations were recorded, this helped gathering data about time and number of discussion messages needed to complete a task using SOCIO.

In [14] participants were asked to video record their screen while using GenLine. This provided data to characterize participants request strategy, repair strategy, the content of the request, and the request complexity.

In [20] event logs were monitored. These provided timestamped records of interaction events with the programmer's assistant, including conversational exchanges, hiding/showing the assistant, use of the "try again" and "start over" features, and use of copy/paste. Furthermore, conversational transcripts between each participant and the Programmer's Assistant were extracted from the event logs.

5.1.2 Techniques to evaluate each Usability Criteria

The studies we review employed a range of techniques to examine usability criteria. In this section, we outline the methods used for evaluating each specific criterion.

Satisfaction In the papers [19],[18], and [15] on SOCIO and USQA participants filled in SUS (system usability scale) satisfaction questionnaires. These include questions about Ease of use and learnability. We list the 10 standard questions of SUS questionnaires in the appendix. In [8] and [12] on ART and ChatGPT participants were asked if they would keep using ART or ChatGPT in the future or if they would recommend others to use them. [27] and [24] asked questions about readability and understandability of ChatTester or AthenaTest. In the surveys [20] on the Programmer's Assistant, [9] on CodeCompose, [4] on AtomXR, and in the interviews on GenLine [14] participants were directly asked if they were satisfied with the tools output and with the interaction with it. Additionally in [4] participants were asked about learnability and ease of use of AtomXR. Finally, the study on Instigator [2] measured the attractiveness, perspicuity (expressing to what extent participants considered it easy to get familiar with Instigator and to learn how to use it), stimulation (expressing to what extent participants felt motivated to use Instigator in the context of work), and trustworthiness of content (expressing the extent to which participants believed that the layouts generated by Instigator are of good quality and reliable).

Effectiveness In study [7] about Copilot, participants were asked questions about code quality, including unit test success ratio, bugs, code smells, and vulnerabilities. The other study about copilot [25] also included questions about problems faced using copilot. Study [24] gathered data about test coverage using Athenatest. Meanwhile the study [27] on ChatTester asked participants if the generated tests were syntactically correct as well as about success compilation and passing execution. In [19] and [18] participants were asked about the perceived success in carrying out the task using SOCIO. Similarly in [14] participants were asked questions about perceived success and adaptability of GenLine. In [2] the adaptability and the quality of Instigator produced GUI

layouts were evaluated by the participants. The survey on The Programmer's Assistant [20] also included questions about the quality of Assistant's responses. The survey on ChatGPT [12] included questions about effectiveness in different use cases.

Efficiency In [7] participants were asked about time spent per task, with and without using Copilot. In [25] time spent per task was monitored through screen-cast. This enabled measuring time spent per task with and without Copilot's assistance. Similarly, through recording the Telegram conversations in [19] and [18] it was possible to measure both: Time taken by a team to complete the tasks using SOCIO, measured in minute, and number of discussion messages generated by a team during the completion of the tasks via a Telegram group. In [4] the time needed to complete each task with Unity, AtomXR on destop, and AtomXR on headset was recorded. Similarly, [27] compared time cost between AthenTest, ChatGPT, and ChatTester.

Quantitative Summary of Researched Criteria Some studies focused only on researching satisfaction, like the studies on ART [8], CodeCompose [9], and USQA [15]. The study on GenLine [14] focused on effectiveness only. On the other hand the studies on SOCIO [19], and [18] as well as the study on ChatTester [27] researched all usability criteria, based on the new standards for usability [1]. Meanwhile the studies on ChatGPT [12], The Programmer's Assistant [20], AthenaTest [24], and Instigator [2] left out efficiency, [4] on AtomXR left out effectiveness, and the studies on Copilot [7] and [25] left out satisfaction. Figure 6 shows the distribution of the studies on the criteria they examined through a set diagram. Knowing that Copilot and SOCIO was researched in two studies each, we notice that satisfaction was researched in 11 out of the 14 studies (79% of the time), effectiveness in 10 out of the 14 (71% of the time), and efficiency 6 out of 14 (43% of the time).

RQ1 - Summary

Which usability evaluation techniques have been used to study the usability of LLMs in solving software engineering tasks?
Most studies conducted surveys, with some also including interviews. Most of those required participants to use a tool before conducting surveys or interviews. Studies often compared established tools with new LLM-based ones using within-subject and across-subject methods. Some studies focused solely on one usability criterion, while others assessed multiple criteria, with satisfaction being the most and efficiency the least researched.

5.2 Target Group

In this section, we answer *RQ2: What is the target group that can use LLMs for assistance with software engineering tasks?* We do this by reviewing the

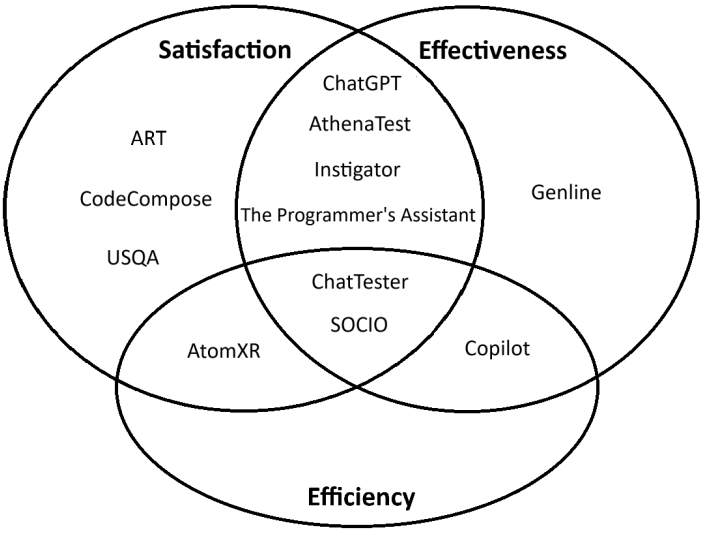

Figure 6: Criteria sets, including the studies on tools that researched them, also showing intersections, where studies researched two or more criteria.

academic and experience levels of participants in the reviewed usability evaluation studies, as well as the knowledge required for each tool. Additionally, We provide an overview of the use cases where these tools are found to be most useful.

5.2.1 Participants Demographics

In the studies [12] on ChatGPT, [8] on ART, [25] on Copilot, [18] on SOCIO, and [15] on USQA the participants were undergraduate students. Study [14] on GenLine involved interaction designers and UX designers, and survey [7] on Copilot included Software engineers, cloud engineers, and data engineers. In [24] on AthenaTest the participants were Microsoft developers. The participants in [9] on CodeCompose were software engineers.

In [4] on AtomXR, [19] on SOCIO, [20] on The Programmer's Assistant, and [27] on ChatTester participants professions are not specified, but their experience levels with programming languages according to each tool were mentioned. The survey [20] on The Programmer's Assistant shows a variety in participants' programming experience level: 40% of the participants have more than three years of experience, while 26% have less than one year of experience. In all the other studies, most participants had a medium level of experience. Table 4 shows the participants of the studies on each tool, mentioning their academic level and their level of experience accordingly.

Tool studied	Participants	Level of Experience	Studies
ChatGPT	Students	1-3 years of programming	[12]
Copilot	Students, software engineers, cloud engineers and data engineers	2-5 years of programming [25], not specified in [7]	[25][7]
GenLine	Interaction designers and ux designers	Variable experience levels with javascript, including not at all experienced to extremely experienced participants	[14]
ART	Undergraduate students	Some knowledge of several SE tools	[8]
Instigator	Software developers and ux designers	Very high experience in GUI design, and above avarage experience in software development overall	[2]
CodeCompose	Software engineers	not specified	[9]
AtomXR	Software developers and game developers	Medium experience in programming and game development	[4]
The programmer's assistant	Software developers	Variable experience levels in programming, including less than one year of experience to more than 3 years	[20]
Chat-Tester	Software developers	Java experience between 4-5 years	[27]
USQA	undergraduate students	Little experience, students were enrolled on a Computer Science program	[15]
AthenaTest	Microsoft developers	Java experience of 1-3 years on average	[24]
SOCIO	Undergraduate and graduates students	Students were completing a computer engineering degree in [18], and gradutes who were familiar with software analysis and design in [19]	[19][18]

Table 4: Participants demographics in usability studies of each tool.

5.2.2 Required Knowledge

Firstly, tools like SOCIO, USQA, and GenLine require a good level of English [18][15][14].

For SOCIO and USQA participants expressed the need to have prior knowledge about the output. For SOCIO one needs to be familiar with class diaramms [18], and for USQA users need to know how to write a user story, its components, and the quality aspects that should be considered when writing it [15].

On the other hand students using ChatGPT were not experienced in programming, they were also not familiar with LLMs or prompt engineering, although 58.5% recognized that effective usage of ChatGPT requires prompt engineering, only 38.7% of the students had taken the initiative to read about prompt engineering and/or explore some examples of it to enhance their utilization of ChatGPT. This did not affect their productivity using ChatGPT. [12]

Also using the programmer's assistant no differences were observed in output value ratings based on participants' familiarity with or recency of using Python [20].

In Copilot, the level of proficiency does not seem to significantly impact the tool's effectiveness and efficiency. Productivity increased by 52.27% for beginners using Copilot, while intermediates showed a 41.6% increase, and advanced users experienced a 40.48% increase. This shows Copilot is most helpful for beginners but is significantly useful for users with other proficiency levels as well [25].

On the contrary, novice participants using GenLine could not complete a task, where they had to create a flashcard app that changed a card from front to back with the click of a button, whereas all self-reported expert participants successfully completed it. Novice participants could only complete another simpler task involving creating a static search page with a text box and a logo [14]. This shows the need to have programming experience to successfully use Genline for complicated tasks.

5.2.3 Use Cases

ChatGPT is found to be more commonly utilized for tasks involving algorithm suggestions and defect detection. It is used less frequently for tasks such as log analysis and resolving security-related issues, with the primary task being generating source code. [12]

Copilot was found useful to provide a starting point for the tasks by participants in the survey [25], where the tasks included editing CSV file, web scrapping, and graph plotting.

ART provided helpful feedback for visual/style defects and documentation defects, and was not as helpful for functional and structural defects. [8]

AthenaTest proved useful for generating Test Cases for Defects4j projects, it showed more coverage than GPT3 and Evosuite on the following Defects4j projects: Apache Commons Lang, JFreeChart, Apache Common Cli, Apache

Common Csv, and Google Gson. [24]

USQA was percieved as good for beginning students who want to use the user stories as a way of writing requirements, most students found it useful for learning how to create user stories. [15]

The Programmer's Assistant was found most helpful for smaller or narrowly-scoped programming tasks, it helped when users forgot the syntax of a certain language and thereby reducing syntax errors. [20]

GenLine was similarily found especially helpful for people who are somewhat familiar with coding, but are unfamiliar with the depths of a specifc language, rusty on syntax, or rarely use a specifc library or API. Users stated that it can be also helpful in the context of working with others, as it could serve as a "universal translator" when two teams are collaborating and using different programming language. [14]

AtomXR was found to be useful for creating XR games, such as space shooters and chase games. It proved particularly efficient for switching between edit and test modes, taking only a second -much faster than the time required to test anything in the Unity environment. Additionally, the immersion with the headset provided by AtomXR significantly enhanced the fine-tuning of objects in space. [4]

RQ2 - Summary

What is the target group that can use LLMs for assistance with Software Engineering tasks?
LLMs can be useful for any target group performing SE tasks. The usability depends on the specific task and tool. ChatGPT and Copilot proved useful to users with various levels of experience, with Copilot helping provide a starting point and ChatGPT suggesting algorithms and detecting defects. GenLine and The Programmer's Assistant are more helpful for less complicated programming tasks. Notably, being more experienced enhances one's performance using GenLine, but does not play a significant role when using The Programmer's Assistant. AtomXR is particularly efficient for switching from editing to testing and enhances fine-tuning objects in space. To use AtomXR some experience in game design is recommended. USQA and ART show good educational use for undergraduate students, with USQA helping them learn how to create user stories and ART providing the most helpful feedback for visual and documentation defects. Tools for creating unit test cases, like Chat-Tester and AthenaTest, appear to be useful for developers regardless of experience levels.

5.3 Challenges

In the study on ChatGPT [12], 90.6% of the students experienced hallucinations in ChatGPT. On a scale of "Never", "Rarely", "Sometimes", "Often", and "Always", 27% of the participants experienced hallucinations often. Furthermore, in the study comparing ChatGPT with CHAT-TESTER [27], participants stated that ChatGPT-generated tests encountered diverse compilation

errors.

The syntax in GenLine and Instigator appeared to be challenging, as the studies [14] and [2] show. Participants in [14] found having unlimited syntax in GenLine not helpful, thus they felt the need to learn the syntax although it is natural language. Furthermore, the study on Instigator [2] showed that experts used vocabulary they knew, related to GUI design, while others did not know exactly what terms could be accepted in the language.

Participants in the studies [25] and [14] on Instigator and Copilot noticed a lack of explainability, specially compared with web search. Participants in [25] found it risky to adopt code blocks produced by Copilot, due to the lack of understanding, which could lead to complications and errors in the code that users could not debug. Furthermore, participants also lacked the possibility to compare multiple alternative solutions.

Participants in [20] also stated that the The Programmer's Assistant lacks the suggestion of multiple answers.

RQ3 - Summary

What challenges arise when using LLMs for assistance with Software Engineering tasks?
Challenges in using LLMs for software engineering tasks include frequent hallucinations in ChatGPT, syntax difficulties in conversational code generators, and lack of explainability of the generated code. Moreover, users found LLM-generated solutions less explainable, especially compared to web searches, and lacking alternative solution comparisons.

6 Discussion

Surveys were the most common method for evaluating the usability of LLMs in software engineering tasks. The use of surveys as the predominant evaluation method underscores the importance of collecting direct user feedback for assessing the usability of LLMs, compared to more passive methods like observational studies. Moreover, including task completions before answering the surveys ensures participants provide informed feedback based on hands-on experience, enhancing the quality and relevance of the data collected.

However, there is a notable lack of variety in the usability evaluation methods found in the papers we reviewed. Most of the studies relied solely on surveys and interviews, conducted after participants had used the tools. Both are inquiry evaluation methods. This indicates a deficiency in studies utilizing inspection and testing methods. Such evaluation techniques are crucial for gaining a comprehensive understanding of the ease of use, understandability, and efficiency of the tools.

Additionally, the comparative studies lack diversity as they all used within-subject designs. This can lead to carryover effects, where the influence of one condition affects performance in subsequent conditions. For instance, participants might approach tasks differently after using the first tool. Another issue

in within-subject studies is fatigue; participants might experience physical or mental tiredness as they progress through multiple conditions or tasks in a single session, leading to less accurate results that do not reflect their actual performance under normal conditions.

The variety of target groups of the studies demonstrates LLMs' broad applicability across different academic and expertise levels. While undergraduates highlight the potential for educational use, professional participants underscore the tools' relevance in industry. Additionally, the mixed experience levels among participants suggest LLMs can benefit both, novices and experts, though complex tasks may pose challenges for less experienced users.

On the other hand, we observe that only very few of the studies considered target groups with varying academic degrees or experience levels. Studies focusing on a single target group fail to capture the diversity of potential users, limiting the generalizability of the findings. Different user groups often have distinct needs, expectations, and challenges, so focusing on just one group might overlook specific usability issues relevant to others. This lack of variation makes it challenging to draw broader conclusions about potential users, as each study only considers a narrow segment.

Many of the reviewed papers also did not assess the efficiency of the tools under investigation, despite efficiency being a crucial criterion for determining the value and viability of tool adoption.

Furthermore, the identified challenges such as hallucinations and syntax issues underscore the need for improving LLMs accuracy and user-friendliness. The lack of explainability and frequent compilation errors highlight the necessity for better transparency, interpretability, and error-handling in LLM based tools to ensure their practical utility in software engineering.

It is important to note that the amount of research gathered in this review might not be extensive enough to draw accurate conclusions. The time and scope limitations of a bachelor thesis did not allow for a more thorough literature search. Furthermore, as this field of research has been getting a lot of attention recently, new studies may have emerged since our initial search, potentially rendering our findings outdated.

There is also a possibility of selection bias if the included literature is not representative of the entire body of available research. Journals tend to publish studies with positive results more frequently than those with negative or inconclusive findings. This overemphasis on positive results can create a false sense of consensus or effectiveness of the review results, leading to publication bias.

Additionally, our limited search query may have caused us to miss relevant research, as important keywords might have been overlooked, despite consulting expert researchers when selecting them. Furthermore, using a shorter query in Science Direct specifically could have resulted in missing relevant studies.

7 Future Research Directions

The limitations we identified in current research highlight the need for further studies on the usability of LLMs in software engineering. There's a clear need for more diverse usability evaluation methods. Studies employing inspection techniques, such as cognitive walkthroughs and heuristic evaluations, are necessary to systematically examine the tools' interfaces, focusing on how users logically navigate tasks. Additionally, studies utilizing testing methods like performance measurement are also needed, as these methods provide crucial feedback on how effectively users can perform tasks with these tools.

As an alternative to within-subject studies, between-subjects studies can be conducted, where different groups of participants are exposed to different conditions or treatments. Each participant experiences only one condition, which allows for comparisons across groups. This study design eliminates carryover effects, since participants are only exposed to one condition. Moreover, it reduces learning and fatigue effects that could influence results.

Furthermore, usability evaluation studies with a broader participants spectrum are needed. Testing with participants of varying experience levels helps identify how easy or difficult it is for newcomers to learn and use the software. This insight allows developers to streamline the user interface and improve usability for new users. On the other hand, experienced users often uncover advanced use cases or workflows that may not have been considered during initial design. This feedback helps refine functionality to accommodate experienced users without sacrificing usability for beginners. Including a broader variety of target groups could also reveal more challenges, as the preferences, behaviors, and experiences of one group may not represent those of other user groups.

Also usability evaluation studies that evaluate all usability criteria of a tool are needed, especially studies that assess efficiency. Since efficiency influences how effectively users can interact with a system, impacting satisfaction, productivity, and the overall success of the tool or system being evaluated.

Finally, further literature reviews on this topic, that includes more literature, incorporating methods such as snowball search or searching a bigger set of databases, could yield more accurate and thorough results.

8 Conclusion

Large Language Models (LLMs) are increasingly being leveraged to assist in various aspects of software engineering. Extensive research has been conducted on the use of LLM-powered tools to address a range of software engineering tasks. In this paper, we reviewed the usability evaluation study conducted on these tools. We begin by examining the types of usability evaluation study conducted (RQ1), followed by an analysis of the study participants to identify the target audience for each tool (RQ2). Finally, we identify the existing challenges associated with using LLMs in software engineering (RQ3). We reveal a lack of diversity in usability evaluation studies on LLM-based tools, including

limited variation in the academic backgrounds and experience levels of participants. Finally, we summarize the challenges users face when utilizing these tools. Based on these findings, we highlight gaps in the current research and propose directions for future studies.

References

[1] Nigel Bevan, Jim Carter, Jonathan Earthy, Thomas Geis, and Susan Harker. New iso standards for usability, usability reports and usability measures. In Masaaki Kurosu, editor, *Human-Computer Interaction. Theory, Design, Development and Practice*, pages 268–278, Cham, 2016. Springer International Publishing.

[2] Paul Brie, Nicolas Burny, Arthur Sluÿters, and Jean Vanderdonckt. Evaluating a large language model on searching for gui layouts. *Proceedings of the ACM on Human-Computer Interaction*, 7(EICS):1–37, 2023.

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[4] Alice Cai, Caine Ardayfio, AnhPhu Nguyen, Tica Lin, and Elena Glassman. Atomxr: Streamlined xr prototyping with natural language and immersive physical interaction. *arXiv e-prints*, pages arXiv–2311, 2023.

[5] G Ann Campbell and Patroklos P Papapetrou. *SonarQube in action*. Manning Publications Co., 2013.

[6] William B Cavnar, John M Trenkle, et al. N-gram-based text categorization. In *Proceedings of SDAIR-94, 3rd annual symposium on document analysis and information retrieval*, volume 161175, page 14. Ann Arbor, Michigan, 1994.

[7] Sayan Chatterjee, Ching Louis Liu, Gareth Rowland, and Tim Hogarth. The impact of ai tool on engineering at anz bank an emperical study on github copilot within coporate environment. *arXiv e-prints*, pages arXiv–2402, 2024.

[8] Aaron S Crandall, Gina Sprint, and Bryan Fischer. Generative pre-trained transformer (gpt) models as a code review feedback tool in computer science programs. *Journal of Computing Sciences in Colleges*, 39(1):38–47, 2023.

[9] Omer Dunay, Daniel Cheng, Adam Tait, Parth Thakkar, Peter C Rigby, Andy Chiu, Imad Ahmad, Arun Ganesan, Chandra Maddila, Vijayaraghavan Murali, et al. Multi-line ai-assisted code authoring. *arXiv e-prints*, pages arXiv–2402, 2024.

[10] Rebecca A Grier, Aaron Bangor, Philip Kortum, and S Camille Peres. The system usability scale: Beyond standard usability testing. In *Proceedings of the human factors and ergonomics society annual meeting*, volume 57, pages 187–191. SAGE Publications Sage CA: Los Angeles, CA, 2013.

[11] Sugandha Gupta. A comparative study of usability evaluation methods. *Int. J. Comput. Trends Technol*, 22(3):103–106, 2015.

[12] Khadija Hanifi, Orcun Cetin, and Cemal Yilmaz. On chatgpt: Perspectives from software engineering students. In *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*, pages 196–205. IEEE, 2023.

[13] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review, 2023.

[14] Ellen Jiang, Edwin Toh, Alejandra Molina, Kristen Olson, Claire Kayacik, Aaron Donsbach, Carrie J Cai, and Michael Terry. Discovering the syntax and strategies of natural language programming with generative language models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, pages 1–19, 2022.

[15] Samantha Jiménez, Arnulfo Alanis, Claudio Beltrán, Reyes Juárez-Ramírez, Alan Ramírez-Noriega, and Claudia Tona. Usqa: A user story quality analyzer prototype for supporting software engineering students. *Computer Applications in Engineering Education*, 31(4):1014–1024, 2023.

[16] Barbara Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. 2, 01 2007.

[17] Boyang Li, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Nicholas A Kraft. Automatically documenting unit test cases. In *2016 IEEE international conference on software testing, verification and validation (ICST)*, pages 341–352. IEEE, 2016.

[18] Ranci Ren, John W Castro, Adrián Santos, Oscar Dieste, and Silvia T Acuña. Using the socio chatbot for uml modelling: A family of experiments. *IEEE Transactions on Software Engineering*, 49(1):364–383, 2022.

[19] Ranci Ren, John W Castro, Adrián Santos, Sara Pérez-Soler, Silvia T Acuña, and Juan De Lara. Collaborative modelling: Chatbots or on-line tools? an experimental study. In *Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering*, pages 260–269, 2020.

[20] Steven I Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D Weisz. The programmer's assistant: Conversational interaction with a large language model for software development. In *Proceedings of*

the 28th International Conference on Intelligent User Interfaces, pages 491–514, 2023.

[21] Robin M Schmidt. Recurrent neural networks (rnns): A gentle introduction and overview. *arXiv e-prints*, pages arXiv–1912, 2019.

[22] Amarjeet Singh and Alok Aggarwal. A comparative analysis of veracode snyk and checkmarx for identifying and mitigating security vulnerabilities in microservice aws and azure platforms. *Asian Journal of Multidisciplinary Research & Review*, 3(2):232–244, 2022.

[23] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.

[24] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. Unit test case generation with transformers and focal context. *arXiv e-prints*, pages arXiv–2009, 2020.

[25] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 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*, pages 1–7, 2022.

[26] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[27] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv e-prints*, pages arXiv–2305, 2023.

[28] Zibin Zheng, Kaiwen Ning, Jiachi Chen, Yanlin Wang, Wenqing Chen, Lianghong Guo, and Weicheng Wang. Towards an understanding of large language models in software engineering tasks, 2023.

Appendix

The Code used to extract Taxonomies

```
import requests
import json
import csv
import os

CORPUS_API_URL = "https://sc-030206l.intra.dlr.de:20246/document/"
# Fill in apikey (BUT DON'T STORE IN REPOSITORY OR ANY OTHER PLACE WHERE IT CAN BE COMPROMISED)
CORPUS_PUB_API_KEY = "#######"

def get_taxotags(text_string, text_type="Science", text_lng="en"):
    taxo_parms_dict = {
        "enScience": {
            "taxonomy": "OpenAlex",
            "language": "en",
            "tuning_parameters": {
                "use_descriptions": True,
                "language_model": "all-MiniLM-L6-v2",
                "top_k": 10,
                "min_similarity": 0.35,
                "title_weight": 1.5,
                "saturation": 1.5,
                "min_agg": 0.35,
                "top_percent": 80,
                "second_percent": 80,
                "min_hscore": 0.35,
                "max_baseconcepts": 7,
                "max_paths": 2
            },
        },
        "deScience": {
            "taxonomy": "OpenAlex",
            "language": "de",
            "tuning_parameters": {
                "use_descriptions": True,
                "language_model": "paraphrase-multilingual-MiniLM-L12-v2",
                "top_k": 10,
                "min_similarity": 0.4,
                "title_weight": 1.5,
                "saturation": 1.5,
                "min_agg": 0.4,
                "top_percent": 80,
                "second_percent": 80,
                "min_hscore": 0.4,
                "max_baseconcepts": 7,
                "max_paths": 2
            },
        },
        "enMedia": {
            "taxonomy": "IptcMedia",
```

```
            "language": "en",
            "tuning_parameters": {
                "use_descriptions": True,
                "language_model": "all-MiniLM-L6-v2",
                "top_k": 10,
                "min_similarity": 0.25,
                "title_weight": 1.5,
                "saturation": 1.5,
                "min_agg": 0.25,
                "top_percent": 80,
                "second_percent": 80,
                "min_hscore": 0.25,
                "max_baseconcepts": 7,
                "max_paths": 2
            },
        },
        "deMedia": {
            "taxonomy": "IptcMedia",
            "language": "de",
            "tuning_parameters": {
                "use_descriptions": True,
                "language_model": "paraphrase-multilingual-MiniLM
    -L12-v2",
                "top_k": 10,
                "min_similarity": 0.25,
                "title_weight": 1.5,
                "saturation": 1.5,
                "min_agg": 0.25,
                "top_percent": 80,
                "second_percent": 80,
                "min_hscore": 0.25,
                "max_baseconcepts": 7,
                "max_paths": 2
            },
        }
    }
    if not taxo_parms_dict.get(text_lng + text_type):
        return None
    data = json.dumps({
        "payload": {
            "text": text_string
        },
        "taxo_parms": taxo_parms_dict[text_lng + text_type]
    })
    res = requests.post(
        CORPUS_API_URL + "taxonomytags",
        headers = {
            'accept': 'application/json',
            'ACCESS_TOKEN': CORPUS_PUB_API_KEY,
            'Content-Type': 'application/json'
        },
        data=data,
        verify=False
    )
    return res.json()
```

Source Code 1: Internal code to extract taxonomies and automate extraction out of table with abstracts. Pass key is needed.

SUS Questions

The ten questions are as follows:

1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.
6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.
8. I found the system very cumbersome to use.
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system. [10]

Full Taxonomy Table

Title	Authors	Taxonomy 0
UniversalNER: Targeted Distillation from Large Language Models for Open Named Entity Recognition	Wenxuan Zhou, Sheng Zhang, Yu Gu, Muhao Chen, and Hoifung Poon	chemistry
Towards JavaScript program repair with generative pre-trained transformer (GPT-2)	Márk Lajkó, Viktor Csuvik and László Vidács	computer science
Automated Summarization of Stack Overflow Posts	Bonan Kou, Muhao Chen, and Tianyi Zhang	computer science
Transformer-based language models for software vulnerability detection	Chandra Thapa, Seung Ick Jang, Muhammad Ejaz Ahmed, Seyit Camtepe, Josef Pieprzyk, and Surya Nepal	computer science
A Lightweight Framework for High-Quality Code Generation	Mohammed Latif Siddiq, Beatrice Casey, and Joanna Santos	computer science
Evaluating AIGC Detectors on Code Content	Jian Wang, Shangqing Liu, Xiaofei Xie, and Yi Li	computer science
LeTI: Learning to Generate from Textual Interactions	Xingyao Wang, Hao Peng, Reyhaneh Jabbarvand, and Heng Ji	computer science
CoditT5: Pretraining for source code and natural language editing	Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric	computer science
Measuring and Mitigating Constraint Violations of In-Context Learning for Utterance-to-API Semantic Parsing	Shufan Wang, Sebastien Jean, Sailik Sengupta, James Gung, Nikolaos Pappas, and Yi Zhang	computer science
ReCode: Robustness Evaluation of Code Generation Model	Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, et al.	computer science
API Entity and Relation Joint Extraction from Text via Dynamic Prompt-tuned Language Model	Qing Huang, Yanbang Sun, Zhenchang Xing, Min Yu, Xiwei Xu, and Qinghua Lu	computer science
Analysis of ChatGPT on Source Code	Ahmed Sadik, Antonello Ceravola, Frank Joublin, and Jibesh Patra	computer science
Enabling Programming Thinking in Large Language Models Toward Code Generation	Jia Li, Ge Li, Yongmin Li, and Zhi Jin	computer science
Think Outside the Code: Brainstorming Boosts Large Language Models in Code Generation	Xin-Ye Li, Jiang-Tian Xue, Zheng Xie, and Ming Li	computer science
Is ChatGPT the Ultimate Programming Assistant -- How far is it?	Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F. Bissyandé	computer science
A Syntax-Guided Multi-Task Learning Approach for Turducken-Style Code Generation	Guang Yang, Yu Zhou, Xiang Chen, Xiangyu Zhang, Yiran Xu, Tingting Han, and Taolue Chen	computer science
Improving Few-Shot Prompts with Relevant Static Analysis Products	Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl T Barr	computer science
Is Model Attention Aligned with Human Attention? An Empirical Study on Large Language Models for Code Generation	Bonan Kou, Shengmai Chen, Zhijie Wang, Lei Ma, and Tianyi Zhang	computer science
Assemble foundation models for automatic code summarization	Jian Gu, Pasquale Salza, and Harald C Gall	computer science
Piloting Copilot and Codex: Hot Temperature, Cold Prompts, or Black Magic?	Jean-Baptiste Dï¿½derlein, Mathieu Acher, Djamel Eddine Khelladi, and Benoit Combemale	computer science
Transformer-based Vulnerability Detection in Code at EditTime: Zero-shot, Few-shot, or Fine-tuning?	Aaron Chan, Anant Kharkar, Roshanak Zilouchian Moghaddam, Yevhen Mohylevskyy, Alec Helyar, Eslam Kamal, Mohamed Elkamhawy, and Neel Sundaresan	computer science
How Effective Are Neural Networks for Fixing Security Vulnerabilities	Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah	computer science
DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection	Yizheng Chen, Zhoujie Ding, Xinyun Chen, and David Wagner	computer science
The FormAI Dataset: Generative AI in Software Security Through the Lens of Formal Verification	Norbert Tihanyi, Tamas Bisztray, Ridhi Jain, Mohamed Amine Ferrag, Lucas C Cordeiro, and Vasileios Mavroeidis	computer science
CSGVD: A deep learning approach combining sequence and graph embedding for source code vulnerability detection	Wei Tang, Mingwei Tang, Minchao Ban, Ziguo Zhao, and Mingjun Feng	computer science
Examining zero-shot vulnerability repair with large language models	Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt	computer science
Evaluating representation learning of code changes for predicting patch correctness in program repair	Haoye Tian, Kui Liu, Abdoul Kader Kaboreé, Anil Koyuncu, Li Li, Jacques Klein, Tegawendé F. Bissyandé	computer science
Where is your app frustrating users?	Yawen Wang, Junjie Wang, Hongyu Zhang, Xuran Ming, Lin Shi, and Qing Wang	computer science
Stack Over-Flowing with Results: The Case for Domain-Specific Pre-Training Over One-Size-Fits-All Models	Manisha Mukherjee and Vincent J Hellendoorn	computer science
Automated Program Repair in the Era of Large Pre-Trained Language Models	Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang	computer science
Achieving reliable sentiment analysis in the software engineering domain using bert	Eeshita Biswas, Mehmet Efruz Karabulut, Lori Pollock, and K Vijay-Shanker	computer science
Sentiment analysis for software engineering: How far can pre-trained transformer models go?.	Ting Zhang, Bowen Xu, Ferdian Thung, Stefanus Agus Haryono, David Lo, and Lingxiao Jiang	computer science
Fast changeset-based bug localization with BERT	Agnieszka Ciborowska and Kostadin Damevski	computer science
Too Few Bug Reports? Exploring Data Augmentation for Improved Changeset-based Bug Localization	Agnieszka Ciborowska and Kostadin Damevski	computer science
Representation Learning for Stack Overflow Posts: How Far are We?	Junda He, Zhou Xin, Bowen Xu, Ting Zhang, Kisub Kim, Zhou Yang, Ferdian Thung, Ivana Irsan, and David Lo	computer science
AUGER: automatically generating review comments with pre-training models	Lingwei Li, Li Yang, Huaxi Jiang, Jun Yan, Tiejian Luo, Zihan Hua, Geng Liang, and Chun Zuo	computer science
A Multi-Step Learning Approach to Assist Code Review	Oussama Ben Sghaier and Houari Sahraoui	computer science
Boosting Automated Patch Correctness Prediction via Pre-trained Language Model	Quanjun Zhang, Chunrong Fang, Weisong Sun, Yan Liu, Tieke He, Xiaodong Hao, and Zhenyu Chen	computer science
BERT-and TF-IDF-based feature extraction for long-lived bug prediction in FLOSS: a comparative study	Luiz Gomes, Ricardo da Silva Torres, and Mario LÃºcio CÃ´rtes	computer science
Duplicate bug report detection by using sentence embedding and fine-tuning	Haruna Isotani, Hironori Washizaki, Yoshiaki Fukazawa, Tsutomu Nomoto, Saori Ouji, and Shinobu Saito	computer science
Inferfix: End-to-end program repair with llms	Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy	computer science
Practical program repair in the era of large pre-trained language models	Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang	computer science
Improving Requirements Completeness: Automated Assistance through Large Language Models	Dipeeka Luitel, Shabnam Hassani, and Mehrdad Sabetzadeh	computer science
CIRCLE: Continual repair across programming languages	Wei Yuan, Quanjun Zhang, Tieke He, Chunrong Fang, Nguyen Quoc Viet Hung, Xiaodong Hao, and Hongzhi Yin	computer science
Automatic detection of five api documentation smells: Practitioners' perspectives	Victor Dibia, Adam Fourney, Gagan Bansal, Forough Poursabzi-Sangdeh, Han Liu, and Saleema Amershi	computer science
A Chain of AI-based Solutions for Resolving FQNs and Fixing Syntax Errors in Partial Code	Qing Huang, Jiahui Zhu, Zhenchang Xing, Huan Jin, Changjing Wang, and Xiwei Xu	computer science
Large Language Models are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models	Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang	computer science
Large language models are edge-case fuzzers: Testing deep learning libraries via fuzzgpt	Yinlin Deng, Chunqiu Steven Xia, Chenyuan Yang, Shizhuo Dylan Zhang, Shujing Yang, and Lingming Zhang	computer science
Self-collaboration Code Generation via ChatGPT	Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li	computer science
Norbert: Transfer learning for requirements classification	Tobias Hey, Jan Keim, Anne Koziolek, and Walter F Tichy	computer science
The Scope of ChatGPT in Software Engineering: A Thorough Investigation	Wei Ma, Shangqing Liu, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, and Yang Liu	computer science
Gorilla: Large language model connected with massive apis	Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez	computer science
ChatUniTest: a ChatGPT-based automated unit test generation tool	Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin	computer science
PRCBERT: Prompt Learning for Requirement Classification using BERT-based Pretrained Language Models	Xianchang Luo, Yinxing Xue, Zhenchang Xing, and Jiamou Sun	computer science
Prompting Is All Your Need: Automated Android Bug Replay with Large Language Models	Sidong Feng and Chunyang Chen	computer science
Neural Program Repair with Program Dependence Analysis and Effective Filter Mechanism	Yuwei Zhang, Ge Li, Zhi Jin, and Ying Xing	computer science
A Light Bug Triage Framework for Applying Large Pre-trained Language Model	Jaehyung Lee, Kisun Han, and Hwanjo Yu	computer science
Keep the Conversation Going: Fixing 162 out of 337 bugs for 0.42 each using ChatGPT	Chunqiu Steven Xia and Lingming Zhang.	computer science
Addressing Compiler Errors: Stack Overflow or Large Language Models?	Patricia Widjojo and Christoph Treude	computer science
Predicting Code Coverage without Execution	Michele Tufano, Shubham Chandel, Anisha Agarwal, Neel Sundaresan, and Colin Clement	computer science
A deep context-wise method for coreference detection in natural language requirements	Yawen Wang, Lin Shi, Mingyang Li, Qing Wang, and Yun Yang	computer science
Explainable Automated Debugging via Large Language Model-driven Scientific Debugging	Sungmin Kang, Bei Chen, Shin Yoo, and Jian-Guang Lou	computer science
Using pre-trained models to boost code review automation. In Proceedings of the 44th International Conference on Software Engineering	Rosalia Tufano, Simone Masiero, Antonio Mastropaolo, Luca Pascarella, Denys Poshyvanyk, and Gabriele Bavota	computer science
Adaptive test generation using a large language model	Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip	computer science
Evaluating large language models trained on code	Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al	computer science
When language model meets private library	Daoguang Zan, Bei Chen, Zeqi Lin, Bei Guan, Yongji Wang, and Jian-Guang Lou	computer science
Automated repair of programs from large language models	Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan	computer science
Learning and evaluating contextual embedding of source code	Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi	computer science
Exploring the Robustness of Large Language Models for Solving Programming Problems	Atsushi Shirafuji, Yutaka Watanobe, Takumi Ito, Makoto Morishita, Yuki Nakamura, Yusuke Oda, and Jun Suzuki	computer science
Private-library-oriented code generation with large language models	Daoguang Zan, Bei Chen, Yongshun Gong, Junzhi Cao, Fengji Zhang, Bingchao Wu, Bei Guan, Yilong Yin, and Yongji Wang	computer science
Understanding the Effectiveness of Large Language Models in Code Translation	Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbar[illegible]	computer science
Low level source code vulnerability detection using advanced bert language model	Mansour Alqarni and Akramul Azim	computer science
SelfEvolve: A Code Evolution Framework via Large Language Models	Shuyang Jiang, Yuhao Wang, and Yu Wang	computer science
Discriminating Human-authored from ChatGPT-Generated Code Via Discernable Feature Analysis	Li Ke, Hong Sheng, Fu Cai, Zhang Yunhe, and Liu Ming	computer science
A systematic evaluation of large language models of code	Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn	computer science
Communicative Agents for Software Development	Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun	computer science
Extending Source Code Pre-Trained Language Models to Summarise Decompiled Binarie	Ali Al-Kaswan, Toufique Ahmed, Maliheh Izadi, Anand Ashok Sawant, Premkumar Devanbu, and Arie van Deursen	computer science
AI for Low-Code for AI	Nikitha Rao, Jason Tsay, Kiran Kate, Vincent J Hellendoorn, and Martin Hirzel	computer science
Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT	Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, Eray Tüzün	computer science
ToolCoder: Teach Code Generation Models to use APIs with search tools	Kechi Zhang, Ge Li, Jia Li, Zhuo Li, and Zhi Jin	computer science
Exploring the Effectiveness of LLMs in Automated Logging Generation: An Empirical Study	Yichen Li, Yintong Huo, Zhihan Jiang, Renyi Zhong, Pinjia He, Yuxin Su, and Michael R Lyu	computer science
An automated approach to estimating code coverage measures via execution logs	Fuxiang Chen, Fatemeh H Fard, David Lo, and Timofey Bryksin	computer science
Detecting Phishing Sites Using ChatGPT	Takashi Koide, Naoki Fukushi, Hiroki Nakano, and Daiki Chiba	computer science
Using pre-trained language models to resolve textual and semantic merge conflicts (experience paper)	Jialu Zhang, Todd Mytkowicz, Mike Kaufman, Ruzica Piskac, and Shuvendu K Lahiri	computer science
Measuring coding challenge competence with apps	Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al	computer science
An extensive study on pre-trained models for program understanding and generation	Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang	computer science
MultiPL-E: a scalable and polyglot approach to benchmarking neural code generation	Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et a[illegible]	computer science
Codegen2: Lessons for training llms on programming and natural languages	Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou	computer science
Understanding Programs by Exploiting (Fuzzing) Test Cases	Jianyu Zhao, Yuyang Rong, Yiwen Guo, Yifeng He, and Hao Chen	computer science
Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x	Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al.	computer science
Flakify: A black-box, language model-based predictor for flaky tests	Sakina Fatima, Taher A Ghaleb, and Lionel Briand	computer science
Is this Snippet Written by ChatGPT? An Empirical Study with a CodeBERT-Based Classifier	Phuong T Nguyen, Juri Di Rocco, Claudio Di Sipio, Riccardo Rubei, Davide Di Ruscio, and Massimiliano Di Penta	computer science
Refining ChatGPT-Generated Code: Characterizing and Mitigating Code Quality Issues	Yue Liu, Thanh Le-Cong, Ratnadira Widyasari, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D Le, and David Lo	computer science
Towards Understanding the Capability of Large Language Models on Code Clone Detection: A Survey	Shihan Dou, Junjie Shan, Haoxiang Jia, Wenhao Deng, Zhiheng Xi, Wei He, Yueming Wu, Tao Gui, Yang Liu, and Xuanjing Huang	computer science, biology
LLM4CBI: Taming LLMs to Generate Effective Test Programs for Compiler Bug Isolation	Haoxin Tu, Zhide Zhou, He Jiang, Imam Nur Bani Yusuf, Yuxian Li, and Lingxiao Jiang	computer science, biology
Using deep learning to generate complete log statements	Antonio Mastropaolo, Luca Pascarella, and Gabriele Bavota	computer science, biology, geography
GPT2SP: A transformer-based agile story point estimation approach	Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, and Michael R Lyu	computer science, economics
GPT2SP: A transformer-based agile story point estimation approach	Michael Fu and Chakkrit Tantithamthavorn	computer science, economics
Automatic detection and analysis of technical debts in peer-review documentation of r packages	Junaed Younus Khan and Gias Uddin	computer science, economics
An analysis of the automatic bug fixing performance of chatgpt	Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke	computer science, engineering
Exploring Distributional Shifts in Large Language Models for Code Analysis	Shushan Arakelyan, Rocktim Jyoti Das, Yi Mao, and Xiang Ren	computer science, engineering
No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation	Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng	computer science, engineering
Augmenting commit classification by using fine-grained source code changes and a pre-trained deep neural language model	Lobna Ghadhab, Ilyes Jenhani, Mohamed Wiem Mkaouer, and Montassar Ben Messaoud	computer science, engineering
Automating Method Naming with Context-Aware Prompt-Tuning	Jie Zhu, Lingwei Li, Li Yang, Xiaoxiao Ma, and Chun Zuo	computer science, engineering
Data-driven prototyping via natural-language-based GUI retrieval	Kristian Kolthoff, Christian Bartelt, and Simone Paolo Ponzetto	computer science, engineering
Can Large Language Models Write Good Property-Based Tests?	Vasudev Vikram, Caroline Lemieux, and Rohan Padhye	computer science, engineering
Traceability transformed: Generating more accurate links with pre-trained bert models	Jinfeng Lin, Yalin Liu, Qingkai Zeng, Meng Jiang, and Jane Cleland-Huang	computer science, engineering
VeriGen: A Large Language Model for Verilog Code Generation	Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg	computer science, engineering
On the validity of pre-trained transformers for natural language processing in the software engineering domain	Julian Von der Mosel, Alexander Trautsch, and Steffen Herbold	computer science, engineering
Conversational automated program repair	Chunqiu Steven Xia and Lingming Zhang	computer science, engineering
Large language models are few-shot testers: Exploring llm-based general bug reproduction	Sungmin Kang, Juyeon Yoon, and Shin Yoo	computer science, engineering
Large Language Models Based Automatic Synthesis of Software Specifications Muzahid	Shantanu Mandal, Adhrik Chethan, Vahid Janfaza, SM Mahmud, Todd A Anderson, Javier Turek, Jesmin Jahan Tithi, and Abdullah	computer science, engineering
Large Language Models Are Human-Level Prompt Engineers	Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba	computer science, engineering
Evaluating and improving transformers pre-trained on ASTs for Code Completion	Marcel Ochs, Krishna Narasimhan, and Mira Mezini	computer science, engineering
Outline, then details: Syntactically guided coarse-to-fine code generation	Wenqing Zheng, SP Sharan, Ajay Kumar Jaiswal, Kevin Wang, Yihan Xi, Dejia Xu, and Zhangyang Wang	computer science, engineering
Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation	Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang	computer science, engineering
Learning to predict user-defined types	Kevin Jesse, Premkumar T Devanbu, and Anand Sawant	computer science, history, biology, chemistry
CodeTF: One-stop Transformer Library for State-of-the-art Code LLM	Nghi DQ Bui, Hung Le, Yue Wang, Junnan Li, Akhilesh Deepak Gotmare, and Steven CH Hoi	computer science, mathematics
ClassEval: A Manually-Crafted Benchmark for Evaluating LLMs on Class-level Code Generation	Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou	computer science, mathematics
BEQAIN: An Effective and Efficient Identifier Normalization Approach With BERT and the Question Answering System	Jingxuan Zhang, Siyuan Liu, Lina Gong, Haoxiang Zhang, Zhiqiu Huang, and He Jiang	computer science, mathematics
AI Chain on Large Language Model for Unsupervised Control Flow Graph Generation for Statically-Typed Partial Code	Qing Huang, Zhou Zou, Zhenchang Xing, Zhenkang Zuo, Xiwei Xu, and Qinghua Lu	computer science, mathematics
Jigsaw: Large language models meet program synthesis	Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma	computer science, mathematics
Generation-Augmented Query Expansion For Code Retrieval	Dong Li, Yelong Shen, Ruoming Jin, Yi Mao, Kuan Wang, and Weizhu Chen	computer science, mathematics
xCodeEval: A Large Scale Multilingual Multitask Benchmark for Code Understanding, Generation, Translation and Retrieval	Ahmed Khanfir, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon	computer science, mathematics
Toward less hidden cost of code completion with acceptance and ranking models	Jingxuan Li, Rui Huang, Wei Li, Kai Yao, and Weiguo Tan	computer science, mathematics
Large Language Models are Few-Shot Summarizers: Multi-Intent Comment Generation via In-Context Learning	Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao	computer science, mathematics
CodeCompose: A Large-Scale Industrial Deployment of AI-assisted Code Authoring	Vijayaraghavan Murali, Chandra Maddila, Imad Ahmad, Michael Bolin, Daniel Cheng, Negar Ghorbani, Renuka Fernandez, and Nachiappan Nagappan	computer science, mathematics
An empirical study on code comment completion	Antonio Mastropaolo, Emad Aghajani, Luca Pascarella, and Gabriele Bavota	computer science, mathematics
Improving Code Example Recommendations on Informal Documentation Using BERT and Query-Aware LSH: A Comparative Study	Sajjad Rahmani, AmirHossein Naghshzan, and Latifa Guerrouj	computer science, mathematics
Codet5+: Open code large language models for code understanding and generation	Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi	computer science, mathematics
The Vault: A Comprehensive Multilingual Dataset for Advancing Code Understanding and Generation	Dung Nguyen Manh, Nam Le Hai, Anh TV Dau, Anh Minh Nguyen, Khanh Nghiem, Jin Guo, and Nghi DQ Bui	computer science, mathematics
Few-shot learning for sentence pair classification and its applications in software engineering	Robert Kraig Helmeczi, Mucahit Cevik, and Savas Yildirim	computer science, mathematics
PTM4Tag: sharpening tag recommendation of stack overflow posts with pre-trained models	Junda He, Bowen Xu, Zhou Yang, DongGyun Han, Chengran Yang, and David Lo	computer science, mathematics
Enhancing Traceability Link Recovery with Unlabeled Data	Jianfei Zhu, Guanping Xiao, Zheng Zheng, and Yulei Sui	computer science, mathematics
On Contrastive Learning of Semantic Similarity forCode to Code Search	Anthony Saieva, Saikat Chakraborty, and Gail Kaiser	computer science, mathematics
Enhancing Automated Program Repair through Fine-tuning and Prompt Engineering	Rishov Paul, Md. Mohib Hossain, Mohammed Latif Siddiq, Masum Hasan, Anindya Iqbal, and Joanna C. S. Santos	computer science, mathematics
An exploratory study on code attention in BERT	Rishab Sharma, Fuxiang Chen, Fatemeh Fard, and David Lo	computer science, mathematics
CERT: Continual Pre-training on Sketches for Library-oriented Code Generation	Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian Guang Lou	computer science, mathematics
Cross-Modal Contrastive Learning for Code Search	Zejian Shi, Yun Xiong, Xiaolong Zhang, Yao Zhang, Shanshan Li, and Yangyong Zhu	computer science, mathematics
What do they capture? a structural analysis of pre-trained language models for source code	Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin	computer science, mathematics
Automated Program Repair Based on Code Review: How do Pre-trained Transformer Models Perform?	Rishov Paul, Md Mohib Hossain, Masum Hasan, and Anindya Iqbal	computer science, mathematics
CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models	Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Tao Xie, and Qianxiang Wang	computer science, mathematics
Invalidator: Automated patch correctness assessment via semantic and syntactic reasoning	Thanh Le-Cong, Duc-Minh Luong, Xuan Bach D Le, David Lo, Nhat-Hoa Tran, Bui Quang-Huy, and Quyet-Thang Huynh	computer science, mathematics
Semantic Compression With Large Language Models	Henry Gilbert, Michael Sandborn, Douglas C Schmidt, Jesse Spencer-Smith, and Jules White	computer science, mathematics
Evaluating ChatGPT and GPT-4 for Visual Programming	Adish Singla	computer science, mathematics
An empirical study on the usage of transformer models for code completion	Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Antonio Mastropaolo, Emad Aghajani, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota	computer science, mathematics
Benchmarking Language Models for Code Syntax Understanding	Da Shen, Xinyun Chen, Chenguang Wang, Koushik Sen, and Dawn Song	computer science, mathematics
LmPa: Improving Decompilation by Synergy of Large Language Model and Program Analysis	Xiangzhe Xu, Zhuo Zhang, Shiwei Feng, Yapeng Ye, Zian Su, Nan Jiang, Siyuan Cheng, Lin Tan, and Xiangyu Zhang	computer science, medicine
TABASCO: A Transformer Based Contextualization Toolkit. Science of Computer Programming	Ambarish Moharil and Arpit Sharma	computer science, philosophy
Multilingual Code Co-Evolution Using Large Language Models	Jiyang Zhang, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric	computer science, philosophy
Identification of intra-domain ambiguity using transformer-based machine learning	Ambarish Moharil and Arpit Sharma	computer science, philosophy
LLM is Like a Box of Chocolates: the Nondeterminism of ChatGPT in Code Generation	Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang	computer science, philosophy, physics
Automated handling of anaphoric ambiguity in requirements: a multi-solution study	Saad Ezzini, Sallam Abualhaija, Chetan Arora, and Mehrdad Sabetzadeh	computer science, physics
Improving ChatGPT Prompt for Code Generation	Chao Liu, Xuanlin Bao, Hongyu Zhang, Neng Zhang, Haibo Hu, Xiaohong Zhang, and Meng Yan	computer science, physics
Copilot for Xcode: Exploring AI-Assisted Programming by Prompting Cloud-based Large Language Models	Chee Wei Tan, Shangxin Guo, Man Fai Wong, and Ching Nam Hang	computer science, physics
Exploring the Effectiveness of Large Language Models in Generating Unit Tests	Mohammed Latif Siddiq, Joanna Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinicius Carvalho Lopes	computer science, physics
Natural Language Generation and Understanding of Big Code for AI-Assisted Programming: A Review	Man-Fai Wong, Shangxin Guo, Ching-Nam Hang, Siu-Wai Ho, and Chee-Wei Tan	computer science, psychology
Towards Generating Functionally Correct Code Edits from Natural Language Issue Descriptions	Sarah Fakhoury, Saikat Chakraborty, Madan Musuvathi, and Shuvendu K Lahiri	computer science, psychology
From Copilot to Pilot: Towards AI Supported Software Development	Rohith Pudari and Neil A Ernst	computer science, psychology
Self-planning code generation with large language model.	Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li	computer science, sociology
Using transfer learning for code-related tasks	Antonio Mastropaolo, Nathan Cooper, David Nader Palacio, Simone Scalabrino, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota	economics, engineering
WizardCoder: Empowering Code Large Language Models with Evol-Instruct	Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang	mathematics
Spt-code: Sequence-to-sequence pre-training for learning source code representations	Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo	mathematics
Codefill: Multi-token code completion by jointly learning from structure and naming sequences	Maliheh Izadi, Roberta Gismondi, and Georgios Gousios	mathematics, sociology
Cctest: Testing and repairing code completion systems	Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Dong Chen, Shuai Wang, and Cuiyun Gao	psychology, philosophy, mathematics
Evaluation of Context-Aware Language Models and Experts for Effort Estimation of Software Maintenance Issues	Mohammed Alhamed and Tim Storer	
Improving code generation by training with natural language feedback	Angelica Chen, Jeremy Scheurer, Tomasz Korbak, Jon Ander Campos, Jun Shern Chan, Samuel R Bowman, Kyunghyun Cho, and Ethan Perez	
Aligning Offline Metrics and Human Judgments of Value of AI-Pair Programmers	Victor Dibia, Adam Fourney, Gagan Bansal, Forough Poursabzi-Sangdeh, Han Liu, and Saleema Amershi	
Large Language Models of Code Fail at Completing Code with Potential Bugs	Tuan Dinh, Jinman Zhao, Samson Tan, Renato Negrinho, Leonard Lausen, Sheng Zha, and George Karypis	
GrACE: Generation using Associated Code Edits	Priyanshu Gupta, Avishree Khare, Yasharth Bajpai, Saikat Chakraborty, Sumit Gulwani, Aditya Kanade, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari	
Shipwright: A human-in-the-loop system for dockerfile repair	Jordan Henkel, Denini Silva, Leopoldo Teixeira, Marcelo d'Amorim, and Thomas Reps	
Augmenting Greybox Fuzzing with Generative AI	Jie Hu, Qian Zhang, and Heng Yin	
ANPL: Compiling Natural Programs with Interactive Decomposition	Di Huang, Ziyuan Nan, Xing Hu, Pengwei Jin, Shaohui Peng, Yuanbo Wen, Rui Zhang, Zidong Du, Qi Guo, Yewen Pu, et al	
Attention, Compilation, and Solver-based Symbolic Analysis are All You Need	Prithwish Jana, Piyush Jha, Haoyang Ju, Gautham Kishore, Aryan Mahajan, and Vijay Ganesh	
Capturing failures of large language models via human cognitive biases	Erik Jones and Jacob Steinhardt	
PAC Prediction Sets for Large Language Models of Code	Adam Khakhar, Stephen Mell, and Osbert Bastani	
Less is more: Summary of long instructions is better for program synthesis	Kirby Kuznia, Swaroop Mishra, Mihir Parmar, and Chitta Baral	
Interactive code generation via test-driven user-intent formalization	Shuvendu K Lahiri, Aaditya Naik, Georgios Sakkas, Piali Choudhury, Curtis von Veh, Madanlal Musuvathi, Jeevana Priya Inala, Chenglong Wang, and Jianfeng Gao.	
DS-1000: A natural and reliable benchmark for data science code generation. In International Conference on Machine Learning	Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu	
CodeIE: Large Code Generation Models are Better Few-Shot Information Extractors	Peng Li, Tianxiang Sun, Qiong Tang, Hang Yan, Yuanbin Wu, Xuanjing Huang, and Xipeng Qiu	
Multi-task learning based pre-trained language model for code completion	Fang Liu, Ge Li, Yunfei Zhao, and Zhi Jin	
RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems	Tianyang Liu, Canwen Xu, and Julian McAuley	
Studying the usage of text-to-text transfer transformer to support code-related tasks	Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota	
LLMatic: Neural Architecture Search via Large Language Models and Quality-Diversity Optimization	Muhammad U Nasir, Sam Earle, Julian Togelius, Steven James, and Christopher Cleghorn	
Demystifying GPT Self-Repair for Code Generation	Theo X Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama	
QIGen: Generating Efficient Kernels for Quantized Inference on Large Language Models	Tommaso Pegolotti, Elias Frantar, Dan Alistarh, Markus Püschel	
Multilingual Adapter-based Knowledge Aggregation on Code Summarization for Low-Resource Languages	Iman Saberi, Fatemeh Fard, and Fuxiang Chen	
Utilization of Pre-trained Language Model for Adapter-based Knowledge Transfer in Software Engineering	Iman Saberi, Fatemeh Fard, and Fuxiang Chen	
Extending the Frontier of ChatGPT: Code Generation and Debugging	Fardin Ahsan Sakib, Saadat Hasan Khan, and AHM Karim	
On the effectiveness of transfer learning for code search	Pasquale Salza, Christoph Schwizer, Jian Gu, and Harald C Gall	
Bloom: A 176b-parameter open-access multilingual language model	Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al.	
Large Language Model Programs	Imanol Schlag, Sainbayar Sukhbaatar, Asli Celikyilmaz, Wen-tau Yih, Jason Weston, Jürgen Schmidhuber, Xian Li	
When GPT Meets Program Analysis: Towards Intelligent Detection of Smart Contract Logic Vulnerabilities in GPTScan.	Liu Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang	
ChatGPT vs SBST: A Comparative Assessment of Unit Test Suite Generation	Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo.	
The potential of LLMs for coding with low-resource and domain-specific programming languages	Artur Tarassow	
The Best of Both Worlds: Combining Learned Embeddings with Engineered Features for Accurate Prediction of Correct Patches	Haoye Tian, Kui Liu, Yinghua Li, Abdoul Kader Kaboré, Anil Koyuncu, Andrew Habib, Li Li, Junhao Wen, Jacques Klein, Tegawendé F. Bissyandé	
Clear: contrastive learning for api recommendation	Moshi Wei, Nima Shiri Harzevili, Yuchao Huang, Junjie Wang, and Song Wang	
On the Usage of Continual Learning for Out-of-Distribution Generalization in Pre-trained Language Models of Code	Martin Weyssow, Xin Zhou, Kisub Kim, David Lo, and Houari Sahraoui	
Generating Data for Symbolic Language with Large Language Models	Jiacheng Ye, Chengzu Li, Lingpeng Kong, and Tao Yu.	
Self-Edit: Fault-Aware Code Editor for Code Generation	Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin	
ALGO: Synthesizing Algorithmic Programs with Generated Oracle Verifiers	Kexun Zhang, Danqing Wang, Jingtao Xia, William Yang Wang, and Lei Li	
Making the most of small Software Engineering datasets with modern machine learning	Julian Aron Prenner and Romain Robbes	
Aspect-based api review classification: How far can pre-trained transformer model go?	Chengran Yang, Bowen Xu, Junaed Younus Khan, Gias Uddin, Donggyun Han, Zhou Yang, and David Lo	

Table 5: Full table resulted when using our code on abstracts of papers obtained in the literature review [13], including papers that produced no output.