Master's Thesis

AUTOMATED CODE GENERATION FOR PAIR PROGRAMMING: A SYSTEMATIC MAPPING STUDY

LOAY ELZOBAIDY

July 13, 2023

Advisor:
Advisor Name   Chair of Software Engineering

Examiners:
Prof. Dr. Sven Apel   Chair of Software Engineering
Second Examiner   Affiliation 2

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,______________________ ____________________________

(Datum/Date) (Unterschrift/Signature)

ABSTRACT

The software industry has recognized the potential impact of code generation tools, making it a significant topic in today's world. These tools are capable of improving the efficiency of software development, saving time and costs. Pair programming is another widely adopted technique in software engineering, popular for its ability to create high-quality software within a short time frame.

To explore the potential of using code generation tools in pair programming, this study aims to conduct a systematic mapping review. We gathered 25 studies related to this topic and categorized them into nine dimensions. Our primary objective was to address two main questions: (1) what is the present state of automated code generation tools in research, including the variation in studies over the years, the methodologies, technologies, objectives, and tasks used in these studies, and (2) what gaps still need to be filled in this field?

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

1 INTRODUCTION

1.1 GOAL OF THIS THESIS

The goal of the thesis is to investigate the current state of code generation tools that can be used during pair programming. This research aims to explore five main characteristics that are important in determining the state of code generation tools research. The first characteristic is the methodologies used during the study. These methodologies are important in guiding the researchers on which methodologies have been heavily used in this field.

The second characteristic that this study aims to investigate is how the studies related to code generation tools have evolved over time. The research will examine how the field has developed over the last three years, including the number of papers that have been published in each year. This will help us to identify how much attention is this field getting at the current time.

The next two characteristics are the current tasks and objectives used in the field of code generation tools. This research aims to identify the current tasks that developers use code generation tools for and the objectives they hope to achieve. This will help to understand the most important aspects of code generation tools and how they are being used in the current time. The study will also highlight any limitations or areas where improvements are needed.

In conclusion, the main goal of this thesis is to identify the current state of code generation tools used during pair programming. By investigating the methodologies, the evolution of the field during the last three years, current tasks, objectives, and technologies used in the field, the study aims to identify any gaps in the existing research. This research is essential for researchers to understand the most effective ways to use code generation tools and to identify areas where further research is needed.

1.2 OVERVIEW

The thesis is divided into several parts. The first part is the motivation section, where the importance of automated code generation tools and pair programming tools will be discussed. The main objective of this section is to establish why the study is significant. The motivation section will outline the current trends, challenges, and opportunities related to automated code generation tools and pair programming.

The second part of the thesis is the research question and study chapter. This section will detail what the study aims to investigate and how the research questions will be addressed.

It will also provide insights into the process that will be used to collect and analyze data. This section will also explain the dimensions of the study and will explain why they were chosen. This section will help readers understand the scope of the research and how it will be carried out.

The third part of the thesis is the evaluation chapter. In this section, a summary of the papers analyzed will be presented, along with the results obtained. The summary will include the research questions and sub-questions, as well as the main findings. The section will also present the results based on the different dimensions identified in the study. The purpose of this section is to give a detailed analysis of the data collected and highlight the main findings.

The fourth part of the thesis is the related work chapter. This section will review the literature that was not included in the study but have interesting findings. It provides another angle of view at the field and the current research that is done in this area but doesn't fall in the scope of this study.

The final part of the thesis is the conclusion. In this section, the research questions will be answered based on the findings extracted. The section will also provide suggestions for future research related to this study and the field in general. The conclusion will summarize the main findings and their implications for the field of automated code generation tools and pair programming. The section will also discuss the limitations of the study and suggest ways to address them in future research.

2 BACKGROUND

Systematic mapping
is a methodology that provides a structure for the research papers, reports, and data that has been conducted in a certain research area by putting them into categories, mentioned here [4] that a systematic research process creates a visual summary map of its results. This visual summary shows important data that can be used to deduce conclusions about the field of research like how mature are the tools used in this research field and if they are ready for production or still in the lab phase. This methodology used frequently in medical research.

The main objective is frequently, to map the frequencies of publication over time to see trends. A secondary goal can be to identify the forums in which research in the area has been published. Based on Kai Petersen[16], only one specific instance of a systematic mapping was investigated within software engineering(Bailey et al. 2007). This could be as a result of the fact that systematic maps, a technique for compiling software engineering studies, was not yet been heavily identified at that time.

By categorizing the published research reports and findings, a systematic mapping study gives them a structure. It frequently presents a map, a visual representation of its findings. Pedreira and Garcia mentioned [27] that the objective of a systematic mapping study is to present a better-organized view of the current state of the art in the field and to identify any gaps or weaknesses present.

According to Peterson [16] the systematic mapping process followed several steps. First, they start by defining the research question (research scope). Then, they conduct a search on primary studies which is the part they identify the search string. The primary research is obtained by manually searching through appropriate conference or journal papers or utilizing search terms on scientific databases. They try to get papers that fit within the search scope.

The third step involves screening papers for inclusion and exclusion. During this stage, the research team assesses whether the papers fit the search criteria or not. The inclusion and exclusion criteria are determined based on the research questions. By skimming through the abstracts and summaries of primary studies, the team decides whether the papers fall within the research scope or not.

The final step is data extraction and mapping of studies. In this stage, the research team calculates the frequencies of publications in each category. The goal of analyzing the results is to display the publishing frequencies for each category. This allows for the identification of gaps and opportunities for further research by revealing which categories have received the most attention in previous studies.

Pair Programming

is a software engineering process that involves two programmers collaborating at the same workstation while using this agile software development technique. This process is a component of the extreme programming methodology, which is gaining popularity in businesses. The central concept of pair programming is that two individuals working together on the same problem can derive process improvements that result in better software.

Extreme programming is a software development methodology that emphasizes informal, real-time communication over the specialized work outputs required by many conventional design methodologies. Pair programming works well within XP for various reasons, including vocabulary growth and cross-training. It is so important to XP that the methodology mandates that all production code be authored by pairs.

In pair programming, one person assumes the role of the observer or navigator, who carefully observes the driver's(the second partner) work and offers advice while the other person, the driver, actively types on a computer or documents an architecture or design. The two programmers routinely switch between these two duties.

While reviewing, the observer also takes into account the direction of the work, generating suggestions for enhancements and potential issues to handle in the future. The driver can then focus only on completing the current task, using the observer as a safety net and guide.

Begel and Nagappan [6] found that the largest perceived benefits of pair programming were overall fewer bugs in the code, spreading code understanding among the team, and producing higher quality code. Additionally, although the following benefits were not statistically significant in the survey they conducted, some participants reported that they benefited from learning from their partners, better design, and constant code reviews.

Williams and Kiesler mentioned [40], pair programmers are typically expected to collaborate whenever possible. However, there may be situations, such as illness, scheduling conflicts, or efficiency concerns, that require the two individuals to work independently. Experienced pair programmers prioritize the stages of the development cycle, deciding which ones are most crucial to work on together and which ones can be completed separately. When they reunite, they must determine how to incorporate the individually created work.

As a bigger group starts using pair programming as the standard method of working, the long-term continuity of a specific pair becomes less important. An individual programmer can maintain sufficient general awareness to fill in for an absent partner at a moment's notice by partnering regularly with every member of the group.

Language models

is a type of deep learning model that provides probability distributions of a collection of terms in a language. While earlier language models were based on non-neural techniques, there has been a trend toward using neural networks in recent years. These models are commonly used in natural language processing to generate text. Recurrent neural networks (RNNs) have been the primary building block of language models in this shift. They take input sequences and produce a corresponding output probability distribution for the subsequent tokens in a sequence.

Melis and dyer [24] After extensive autonomous black-box hyper parameter tuning and reevaluation of various well-known designs and regularization techniques, it was discovered that traditional LSTM architectures outperform more modern models when properly regularized, which was a rather unexpected result.

RNN-based language models are useful for a wide range of tasks, such as part-of-speech labeling, question-answering, and machine translation. While RNNs generally offer better performance than other models, they also suffer from a short-term memory problem, which can impact their performance, particularly when dealing with lengthy input sequences.

Alon [2] introduced a new method for AnyC2C called structural language modeling, which leverages the inherent syntax of programming languages to represent a code snippet as a tree. By decomposing the program's abstract syntax tree into a set of conditional probabilities over its nodes, SLM estimates the probability of the AST. Their neural model computes these conditional probabilities by considering all possible AST paths that lead to a target node. Unlike previous structural methods that severely restricted the types of expressions that could be generated, their approach can generate arbitrary expressions in any programming language. When applied to producing Java and C# code, their model outperformed seq2seq and several other state-of-the-art structured techniques.

Hellendoorn [13] explored the possibility of using language models to evaluate how well new code integrates with an existing code base. They developed a code completion tool that demonstrated how it could significantly improve the default suggestions provided by the Eclipse IDE. The researchers assessed the practical potential of the high regularity that language models discovered in source code through their tool.

Wang and Chollak [13] proposed a method for bug detection using N-gram Language Models. Their approach, called Bugram, utilizes n-gram language models instead of rules to identify bugs. Bugram models program tokens sequentially using a n-gram language model. The model then calculates the probability of token sequences in the program, and sequences with low probabilities are identified as potential bugs. Low probability token sequences in a program are considered abnormal, which could indicate errors, unethical behavior, or unique or specialized applications of code that developers should be mindful of.

In this subsection, we discussed the advancements of statistical language models in software engineering and their current use in code generation. These language models have been found to be useful in various aspects of software engineering, including code generation. We will examine in this study the current state of using these code generation tools in the field of pair programming.

Code Generation

The field of software development has now advanced to the point where machine learning tools and techniques can be employed in the coding process, thanks to developments in deep learning and natural language processing. As a result, developers can now make extensive use of code completion and generation technologies offered by integrated development environments (IDEs) or as add-ons to text editors.

Perez[29] investigated the applicability of comparable approaches in a highly organized environment with tight syntax restrictions. They specifically suggested an end-to-end machine learning model built on top of pre-trained language models for Python code production. By earning a BLEU score of 0.22—46 percent improvement over a respectable sequence-to-sequence baseline— they showed that a fine-tuned model may perform well in code generation tasks.

As NLP/DL technology develops, these code completion tools get more complex. An advanced code completion tool is Copilot from GitHub. An "AI pair programmer trained on billions of lines of public code," according to Copilot's description. Copilot, a text editor add-on for the VSCode text editor, creates potential code completions for developers by taking into account the program's context.

Dehaerne and Dey did a systematic review code generation using machine learning [10], they found that code generation tools belongs to one of three groups. The first category is Description-to-code, these descriptions are often obtained from code comments written before a code snippet. Second is Code-to-description, This task's goal is to produce a description of the code, typically in the form of a comment. It is sometimes referred to as source code summarizing.

The third category is Code-to-code, This paradigm's most widely used application category is automatic software repair(APR). APR receives flawed or buggy code as input, and the model try to produce identical code without the bug. The majority of code generation studies fall in the first category with 46 %, the second category had 25 % selected studies.

Codex

is a GPT language model fine-tuned on publicly available code from GitHub, specifically developed to produce code. It is claimed to produce best results for Python code-writing capabilities. A distinct production version of Codex powers GitHub Copilot. It may generate code fragments that are typically both syntactically and semantically sound when given a brief user description.

Chen [8] investigated whether it was possible to train large language models to produce functionally correct code bodies from natural language docstrings. They discovered that the model performed well on a dataset of human-written problems with a level of difficulty comparable to simple interview problems by fine-tuning GPT using code from GitHub. To enhance model performance, they recommended producing multiple samples from a model and training on a distribution that is more similar to the evaluation set. They also found that training a model to generate docstrings from code bodies, the reverse task, was straightforward, and that these models had similar performance profiles.

Xu [41] evaluated large language models trained on code, including Codex, the current state-of-the-art model. Although Codex itself is not open-source, they discovered that current open-source models, despite being primarily intended for natural language modeling, do produce comparable results in several programming languages. They found that Codex, which is allegedly focused on Python, performs surprisingly well in other programming languages too, and even better than GPT-J and GPT-NeoX that were trained on the Pile. However, most current models still perform worse than Codex.

Prenner [30] investigated if Codex can be used to fix bugs in existing code. They found that Codex was able to synthesize a correct solution for 45% of the problems in QuixBugs. When given buggy code and asked to fix it, Codex was able to produce five more correct program implementations, representing a 28% improvement. In addition, providing extra input-output examples helped Codex successfully repair one of the seven bugs that no other configuration could solve (specifically, the "sub-sequences" bug) using test cases.

OpenAI developed Codex using a unique approach, based on their discovery the repeatedly sampling from the model can be an effective tactic for generating useful answers to challenging problems. By using this approach and sampling multiple times per problem, they were able to solve more problems. This method can increase the chances of finding a suitable solution. By utilizing Codex's ability to generate code, developers can benefit from this approach and increase their chances of finding efficient and effective solutions to their coding challenges.

Copilot

is an AI-powered tool created by Microsoft that functions as an "AI pair programmer." It is capable of generating code in multiple programming languages based on the context provided as a prompt, including nearby code, comments, and method names.

Copilot offers three primary functionalities: auto-fill for repetitive code, suggestions for tests matching the implementation code, and conversion of comments into code. The tool is using in its core Codex that have been specifically trained and fine-tuned for the purpose of generating code.

GitHub suggests that not all the code used was tested for bugs, insecure practices, or personal data, despite the glowing reviews of Copilot's productivity improvements on the website. The company writes they have put a few filters in place to prevent Copilot from generating offensive language, but it might not be perfect.

The company also cautions that although this is uncommon and that the data has been discovered to be synthetic or pseudo-randomly generated by the algorithm, the model could suggest email addresses, API keys, or phone numbers. However, the majority of the Copilot-generated code is original. Only a small fraction of the generated code could be replicated exactly in the training set.

The foundation of Copilot is OpenAI's Codex, This is a refined version of GPT-3. This package can be used within visual studio code with different settings and was promoted as being a tool that the programmer can pair with.

Alpha Code

DeepMind Alpha Code is an innovative AI system developed by Google-owned DeepMind. The system relies on a powerful combination of deep learning and reinforcement learning techniques to achieve its model. It's powered by a neural network with millions of parameters that have been fine-tuned through training on simulated games. By using deep learning, AlphaCode is able to analyze and understand complex patterns in data, allowing it to make accurate predictions and decisions. And by using reinforcement learning, it is able to learn from its mistakes and continually improve its performance over time. The potential applications of AlphaCode are wide-ranging and include different fields from robotics and self-driving cars to medical diagnosis and scientific research.

AlphaCode was tested on a suite of programming competitions on the Codeforces platform and achieved an average ranking of 54.3 percent, which is a significant achievement. The system uses specially trained transformer-based networks to generate millions of diverse programs, and then filters and clusters those programs to select just the top 10 submissions. This marks the first time an AI system has been able to perform competitively in programming competitions, which has the potential to affect the field of computer programming.

CodeWhisperer

Amazon CodeWhisperer is a new AI-powered tool developed by Amazon Web Services (AWS) that aims to assist developers in the software development process. The tool utilizes natural language processing (NLP) and machine learning techniques to provide suggestions and auto-complete code segments based on the developer's programming language. The system is designed to learn from developers' usage patterns and improve its recommendations over time. According to AWS, CodeWhisperer has shown promising results in reducing the time and effort required to develop high-quality code.

CodeWhisperer can extract patterns and structures from the code to make suggestions to developers, including auto-completion of code segments, highlighting code errors, and providing recommendations for code improvement. The system's machine learning algorithms also enables it to learn from developers' usage patterns, making it more effective over time. It was released June 2022

One of the features of CodeWhisperer is its ability to suggest code for a developer based on their intent. The system uses natural language processing (NLP) techniques to analyze the developer's code and provide suggestions that are aligned with their intended goals. This feature can improve the productivity of developers by reducing the time required to write code manually. CodeWhisperer can be integrated with other AWS services, such as Amazon SageMaker, to provide a complete AI-assisted development experience.

3 MOTIVATION

Today, there is an increasing interest in deep learning applications in software engineering. Those deep learning applications have been introduced in multiple area of software engineering like automated testing, which makes it easier for software testers to automate the software testing process. There are many different automated testing tools available, both open source and commercial tools. An automated tool for Java class robustness testing is called JCrasher. It is a free, open source tool that produces automated unit tests for Java classes with the aim of improving code quality and locating bugs. JCrasher looks at the type information of methods in Java classes and builds code fragments that will produce instances of different types.

Another field is the Bug localisation. It is a field in the wider concept of feature location, which is the process of identifying where in the source code a particular feature has been developed. CodeSonar is a bug detection tool that combines static and dynamic analysis to scan source code for errors and vulnerabilities. It also provides a graphical interface to help developers understand their code and pinpoint areas in need of improvement. The tool is designed to help developers find and fix bugs quickly and efficiently. Mashhad[22], used CodeBERT for program repair. The third field is the code documentation generation. Code documentation is text that explains what the software code does, why it's written in a certain way, and how to use it. The two basic types of documentation are the documentation found within the code and the documentation found to support the code. Doxygen creates technical software documentation from sources for annotated programs. Finally deep learning has been introduced to code generation.

Different technologies have been introduced in this field like CodeBERT. Its model is an expansion of the BERT model created in 2020. This model can help developers by suggesting codes for specific tasks, and many other translations tasks using programming language and natural language. Then the introduction of Codex by OpenAI. Codex is an AI system that translates natural language to code. It can be used in several tasks related to code generation. Recently, Open AI and Microsoft introduced Copilot, a promising Deep Learning (DL)-based solution, as an industrial product. Promoted by Microsoft as the new "AI pair programming", Copilot is a package that uses Codex in its core to auto generate code from developers comments.

In software engineering, automatic program generation has long been a desired solution due to its potential to save time, effort, and money for all stakeholders in the industry. By automating the code synthesis process, developers can focus on other important aspects of software development such as testing and debugging, leading to faster and more efficient development. Additionally, automatic program synthesis can reduce the risk of errors in code and improve the overall quality of the final product.

Pair programming is an effective software development technique that involves two developers working together in the same environment. It enables faster development, better code quality, and improved debugging, as each developer can focus on different aspects of the project. Working together also helps to promote communication and collaboration, which can lead to more creative solutions.

In pair programming, code generation can enhance the efficiency of the development process by allowing two developers to focus on different aspects of the project. For example, one developer can write the code while the other uses a code generator to generate unit tests or documentation. This can help reduce development time, improve code quality, and decrease the need for debugging. Also, code generation can help to make sure the code is consistent and follows the same standards and conventions.

So in this study we will try to build a systematic mapping review for AI pair programming using code generation tools, since it is a new field and it was not yet fully explored. Systematic mapping has shown great benefits in other fields of research such as medicine and other engineering areas. This method is highly useful specially at any field of research. This is for the reason that it is great in showing what is the current state of research in this field and what is the gaps that still exists in the field. This is beneficial to any research field regardless of the age of the field.

4 RESEARCH QUESTION & RESEARCH STUDY

4.1 RESEARCH QUESTION

There is two main questions we are trying to answer in the thesis First:

RQ1 What is the current state of the automated code generation in the current research?

Which can be divided into five sub-questions. By answering each one of them we will get an answer for the main question. The following are the five sub-questions.

RQ1.1 How did the number of publications from the automated code generation change over time?

RQ1.2 What are the most used research methodologies in automated code generation?

RQ1.3 What are the research objectives in automated code generation?

RQ1.4 What are the research tasks in automated code generation?

RQ1.5 What are the used technologies in automated code generation?

Then we are trying to answer the second main question,

RQ2 What are possible research gaps in the field of automated code generation?

Methodology in research is defined as the systematic approach to solving a research topic through the collection of data using various approaches. The way the collected data is interpreted, and the inferences from the study data is refereed to as the methodology in research. A research methodology can be viewed as being the blueprint of a research or study.

Research objectives are the specific goals or objectives of a research project. They are typically expressed in terms of what the researcher hopes to learn or accomplish by conducting the research. Research objectives provide direction to the research. Research gaps signify areas that have not been looked into or areas of research that have not been investigated thoroughly. These spots represent areas where more research is required to give more data or to fill in the voids. It is important to recognize such research gaps as they can offer guidance for upcoming research.

We can answer this question after executing the mapping study, which involves looking at all the relevant papers. Then, we can extract the data from these papers and come to a conclusion. By doing this, we will get an overview of which areas have already been explored and which areas have not been explored yet.

4.2 PROCESS

The process inspired by the mapping study process by Peterson [17] is divided into three main parts. Each part is divided into smaller steps. The first part is the definition part which is the initial process. The second part is executing the search and papers filtration. Finally the last part is mapping study post the search process.

The steps in the method of Systematic Mapping Study. As the first phase shown in Figure 4.1,

4.2.1 *Definition part*

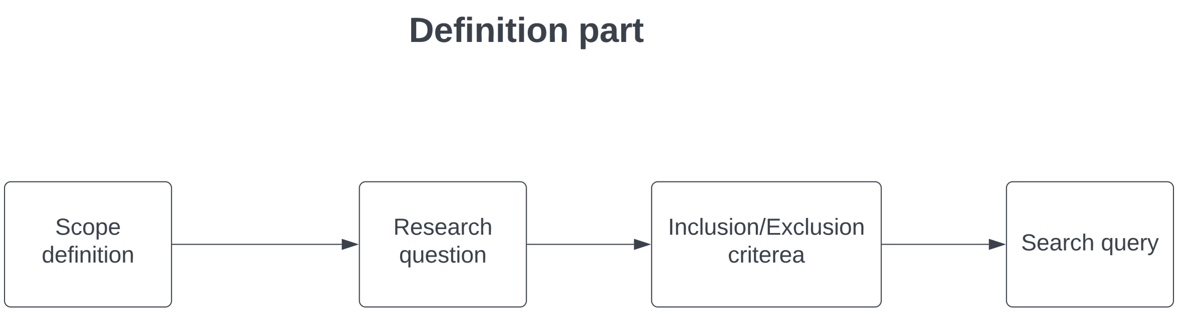

Figure 4.1: Phases of the definition process

The researcher defines the research scope to emphasize the particular area of interest, including the field and topic he wants to study. He then checks related work done in this area and examines it. Based on this examination, the researcher formulates Research Questions (RQs). The next step in this part will be to define the inclusion and exclusion criteria used to determine which research studies should be included and excluded from the review. These criteria can include the type of research conducted, the scope of the research, the time period of the research, and the methods used in the research. The criteria should be tailored to the particular mapping review and may be adjusted as new information is discovered.

The final step in this definition part is the search query definition. This criteria can vary from one search engine to another, as some engines have limitations for the size of the string, the syntax for the conjunction and disjunction, and the syntax for exact words. For example, in Google Scholar the maximum length for query size is 256 characters and searches are not case sensitive. To find an exact match, we use quotation marks around the search term. This means that the words may not always appear together when you search for a phrase without quotes. By placing quotation marks around any groups of words or precise phrases, an exact match of the phrase in the document's title and body text can be found.

Giving a better understanding of the syntax of the Boolean operator to manage the search; For example when two words or phrases on either side of AND is a conjunction, while to exclude results that include a word or phrase, put the word NOT in front of it. The two words or phrases on either side of OR is a disjunction.

Information Sources: The indexing systems and digital libraries used in this systematic mapping search include IEEE Xplore, Springer, ACM, Google scholar. Additionally, we only interested in the papers from 2021 to 2023 in this area. because this was the period were Github copilot, which is the state of the art of this field, was released.

4.2.2 *Search Execution*

The second part of the process phase is the search execution as shown in Figure 4.2

In this part, the researchers run the search engine with the corresponding query and retrieve the papers, all titles, abstracts and papers (reading in detail) to decide the appropriate paper based on the inclusion and exclusion area. This phase starts first with running the query on the search engine. Then, filtering the papers that fits with the search scope based on their titles first so if there is a paper that doesn't fit from the title it is excluded. Then comes the part where filtering papers based on their abstracts, followed by reading full text papers and all the filtration work is done by applying the inclusion and exclusion criteria mentioned in the previous part.

The next part is the scoring part. We give each paper a score from A to F. Each letter means a different category so we have a 6 categories we can group papers shown in table below.

We have 6 categories for the papers to be grouped. The first is Category A, which are the closest type of studies to our interests; these are studies that tested the tools in a pair programming environment. The second category is Category B, which are papers that tested tools designed for pair programming and compared it to human performance. The next category is Category C, which are papers tested tools designed for pair programming and the objective they were testing is one of the objective we are interested to investigate, such as

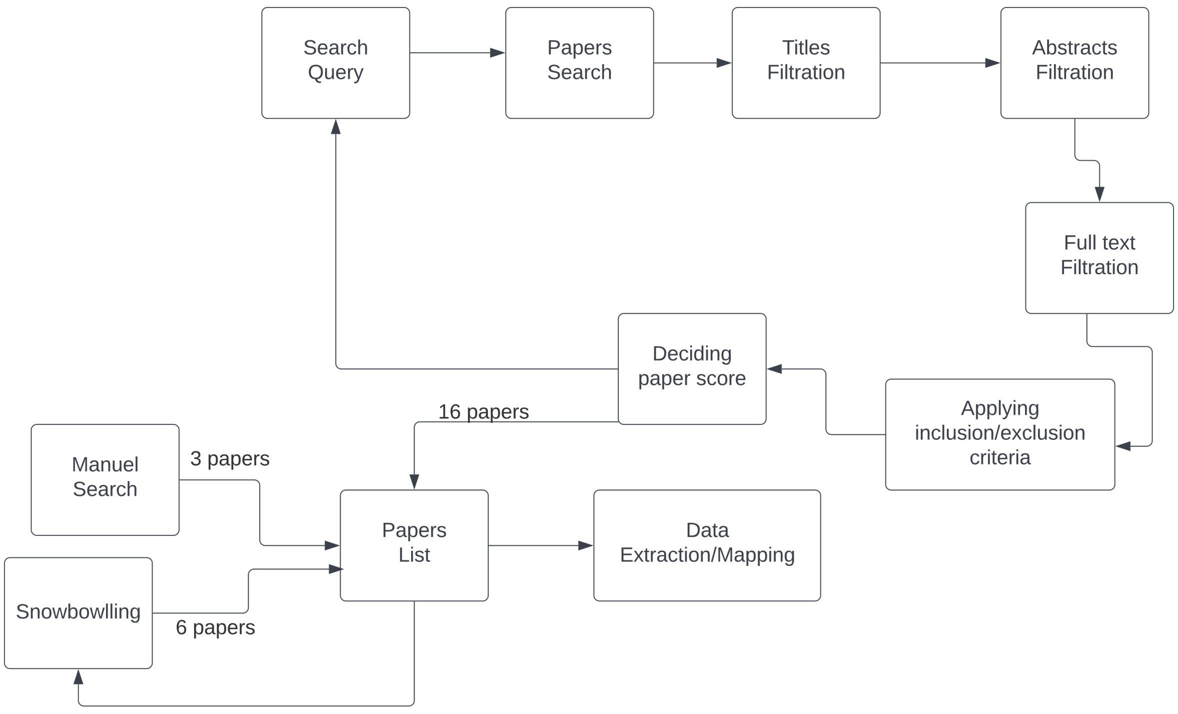

Figure 4.2: Phases of the search process

generating correct, usable, efficient or secure code, or that measured effect on the developers.

The next category is Category D, which are papers that tested tools designed for pair programming but they were testing the tool for an objective we are not investigating in this study, such as code contains bugs, better structure, and testable code. The next category is Category E, which are papers that are used in pair programming, but not related to pair programming. The last category are other tools that can't be used in pair programming nor in code generation. Based on these categories, we decided to only include papers from Categories A to C, as the goal is to check the current state of the art of code generation for pair programming.

Table 4.1: This table represent the categories the papers are grouped on

Category	Meaning
A	tools tested in pair programming setup
B	tools tested against human performance
C	tools tested for an included programming objective
D	tools tested for a not included programming objective
E	non code generation tool tested for pair programming
F	out of scope tools.

After deciding the papers score we create the list of papers to be included. this was the automated part we then execute a manual search to check on the most popular conferences in the last years from 2021 to 2023 to check relevant papers we mainly were interested in 3 conferences which were ICSE, FSE, ASE.

The last part of the search process was the snowballing part. Snowballing is a technique employed in mapping reviews to locate pertinent research papers. It is a type of organized search approach that consists of locating and obtaining papers connected to the research topic from the bibliographies of list of included papers we have. The snowballing method can be used to detect papers from any origin, including databases, journals, and it is especially beneficial for locating papers published in specialized or localized sources.

The selection process was as following the researcher selected the paper based on the criteria mentioned above then he double checked with his supervisor. Everyone of the researcher and the supervisor had a vote to score the papers. In case they both disagree, the supervisor asks to a fellow colleague to solve the category of the paper.

4.2.3 *Mapping part*

The third part of the study is the mapping part as shown in Figure 4.3

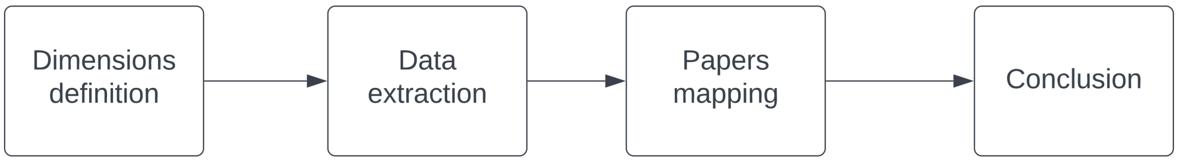

Figure 4.3: Phases of the mapping process

We start first by defining the dimensions which will be discussed in details in the next section. We decide in this part the mapping dimensions that all the papers will be mapped based on.

The next part is the data extraction so we extract from the papers the relevant data that we need to map the papers like which technologies used and what time the papers was published etc. and any interesting results in the papers that can help the readers to identify any research gaps in this field.

The last step is the mapping process, which is done by mapping each paper to its corresponding value in each dimension and extracting important statistics. These statistics data can be used to answer of the research questions and resulting in the systematic to enable the researchers to draw the conclusion of the data obtained.

4.3 INCLUSION/EXCLUSION CRITERIA

4.3.1 *Inclusion criteria*

- Terms that match the search query.
- Academic journal, conference papers.
- Contributions to industry/professional conferences, and online publications.
- English-language papers published up between 2021 and 2023
- Papers that test code generation tools in the pair programming context.
- Papers that compared code generation tools to human code.
- Papers that test tools that are designed for pair programming for an objective that we are investigating like code correctness, efficiency, security and usability.

4.3.2 *Exclusion criteria for titles and abstract*

- Papers that don't specifically address automated code generation for pair programming.
- Papers that test aspects of the tools that are not investigated in this study.
- Papers that are only available as PowerPoint presentations or abstracts.
- Personal websites or blogs.
- Product catalogs

4.3.3 *Exclusion criteria for full text*

- Pair programming in general.

- Code generation in general that can't be used in pair programming.
- Papers presenting a summary of a workshop

The reason we chose 2021 as the starting point is because this is the year that Github copilot was published. Github copilot is considered the first tool that can be used in code generation; prior to this, most tools weren't sufficiently developed for this purpose, so we decided to filter from this year onward.

The reason we are including these papers is because we were looking for papers that include code generation and have potential applications for pair programming, such as testing the tool in a pair programming setup or the tool being advanced enough to aid the developer in a pair programming context.

4.4 SEARCH QUERY

The search query we will be executing over the different databases IEEE, ACM, Google scholar is the following:

```
(Evidence based OR Empirical study)

AND

(Software engineering)

AND

("Github copilot" OR "AlphaCode" OR "CodeWhisperer" OR "Code generation tool")

AND

(Code generation OR Code completion)

AND

(Code security | Code usability | Code correctness | Code efficiency | developer
experience)

AND

("Pair programming" OR Assisting programming)
```

We decided to have six components in our search query, where each component represents a different scope. The search query is composed of sets of conjunctions and disjunctions, so we are using the operator AND for conjunctions and the operator OR for disjunctions. We are using quotation marks ("") for an exact word search, while if the word is left without the quotes, the search engine can search for other synonyms of the word. Therefore, in each of the terms in the query, we have added quotation marks for terms that we are looking for an exact match, and we left the term without quotation marks when the search engine can search for synonyms. The first part of the query is the type of research we are searching for; the two types we have identified are evidence based or empirical study. The second part is software engineering, which is the field in which this study is conducted.

The third part of the query is the technologies we are interested in. We are generally looking for code generation tools. The current state-of-the-art in this area is GitHub copilot. We have also added two other tools that fit with our study scope. The first one is AlphaCode, a new Artificial Intelligence system for developing computer code developed by DeepMind. This tool is mainly modeled to solve contest problems, as these contests setup can be seen as a pair programming setup. Most of these competitions are solved in teams of two to three members, who pair with each other to solve the problems. The second tool is CodeWhisperer, a machine learning tool developed by Amazon. This tool assists developers by generating code suggestions based on their comments in natural language.

The fourth component of the query is function in automated code generation, which includes two tasks: code generation, which refers to the tool's ability to write code correctly, and code completion, which refers to the tool's ability to suggest and complete code. The fifth component is the objective, which represents the purpose of the query. This part is essential as there are multiple variables to consider, but we are mainly interested in tasks that can make the tool suitable for pair programming.

To assess code quality, we are using the standard ISO 9126, which was published by the International Organization for Standardization (ISO) in 2001. The standard defines a framework for software quality characteristics and metrics, which can be used to evaluate and measure the quality of a software product. ISO 9126 identifies six main quality characteristics of software, including functionality, reliability, usability, efficiency, maintainability, and portability. Each of these characteristics is further divided into sub-characteristics.

Functionality is divided into suitability, accuracy, interoperability, security and compliance. While the sub-characteristics of Reliability are maturity, fault tolerance and recoverability. Usability is divided into four sub-characteristics understandability, learnability, operability and attractiveness. Efficiency on the other hand is divided into time behavior and resource utilization. Maintainability is divided into analyzability, changeability, stability and testability. Finally, Portability is divided into four characteristics adaptability, installability, co-existence and replaceability.

we are focusing on four fields: code correctness, which refers to the tool's ability to produce correct and running code that is usable; code usability, which means that the code should be structured in a way that other developers can use it and continue working on it; code efficiency, which can be measured by how the code behaves in terms of time and memory consumption; and code security, as it is a important aspect of code characteristics in modern development. Additionally, we are adding a term that we called "developer experience" because we found multiple papers addressing how satisfied developers were and how productive they were while using these tools, quantitatively (e.g., lines of code written) or qualitatively (e.g., good experience).

In our analysis of code quality, we had to make the decision to exclude three characteristics of it, code portability, code reliability, and code maintainability. We made this decision based on the insufficiency of research done on these aspects, despite our best efforts to find relevant studies. While we did come across a study that tested the code maintainability of Copilot and CodeWhisperer, it was only published on April 21st, 2023, and we did not have enough time to fully analyze its findings, but we added it to the relevant work chapter, discussed later in this study. Unfortunately, we were unable to find any research on the other two aspects of the code quality which are code portability, and code reliability. Therefore, we decided to focus only on the aspects of code correctness, efficiency, usability and code security for our analysis.

The last part of the query is the reason we are using the tool; we have two reasons. The first is pair programming, but we found papers that address the pair programming tool as an assisting programming tool, which is why we added this term to the search query.

4.5 DIMENSIONS

The Dimensions of the mapping study are the categories that each research paper is grouped based on. In this study, we came up with nine dimensions shown in Figure 4.4, that we gonna group the papers based on. The first one is the research methodology shown in Figure 4.5. Research methodology refer to the philosophical approach taken to the study of a particular subject. the three values we are using is quantitative research, qualitative research or mixed.

Qualitative research is a method used in several fields of study to gain an understanding of people's opinions, thoughts, experiences, and behaviors. This type of research utilizes unstructured data, such as interviews, focus groups, and observations. It is used to gain a more detailed insight into the motives behind people's attitudes and actions.

Quantitative research on the other hand is a method used to collect and analyze numerical data to show patterns and relationships. It can be used to determine the correlation between different variables. This type of research usually involves the gathering of data from a sizable group of people or objects to determine trends.

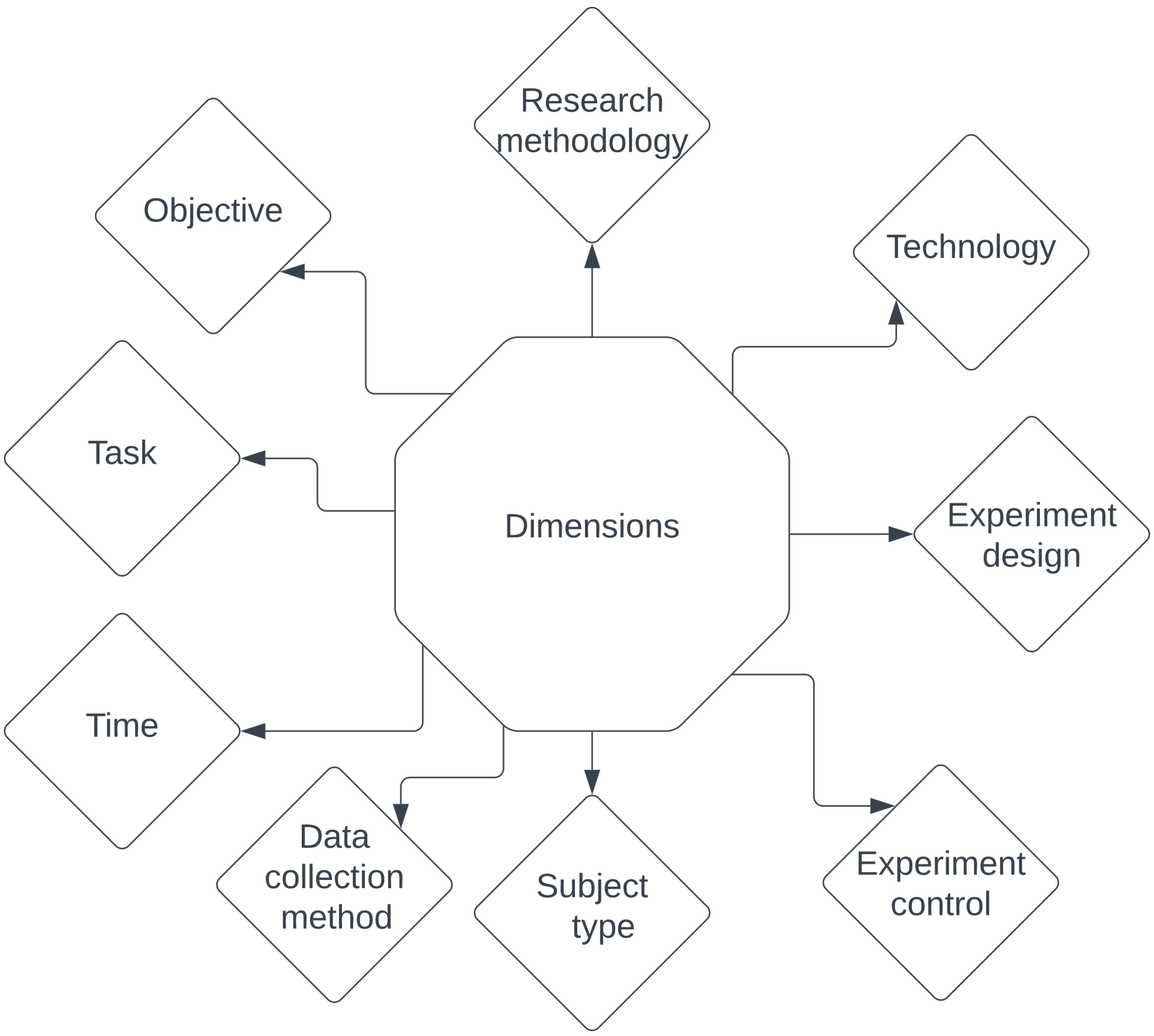

Figure 4.4: Overview of the dimensions

Mixed methods is a research technique that utilizes both qualitative and quantitative approaches in the same study. This approach enables researchers to mix both methods in order to answer questions or verify hypotheses.

The second dimension is technology shown in Figure 4.6, which represents the code generation technology used in this study. There are different technologies in the research field at the moment. We chose the state-of-the-art technologies that can be used in the pair programming process, first GitHub copilot, which is the current state-of-the-art technology for code generation, built on top of Codex and been promoted as the "AI pair programmer".

Then we added AlphaCode, an AI system that has been trained to design algorithms to solve problems, and then implement them into code. This system was designed by Google deep mind, and they followed a different approach, so it is not a tool that generates code from comments but it takes a problem and tries to solve it, and then writes it into code. The main reason we added AlphaCode is that the testing environment is mostly competitive

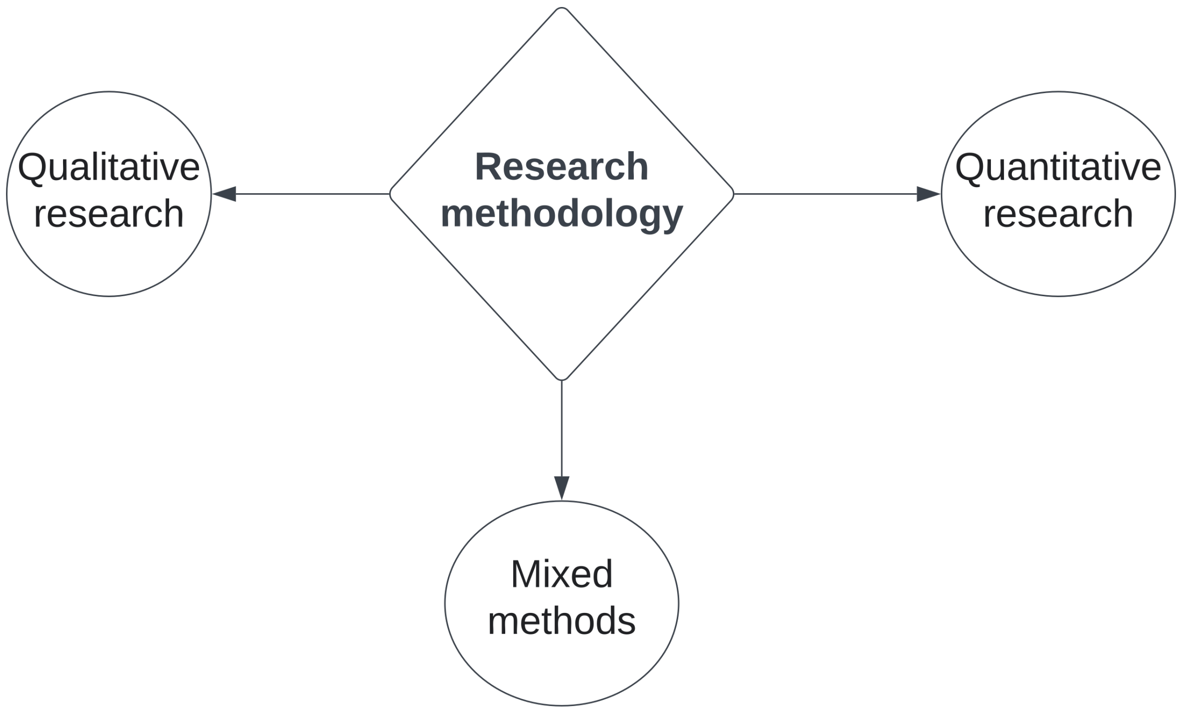

Figure 4.5: Methodology dimension

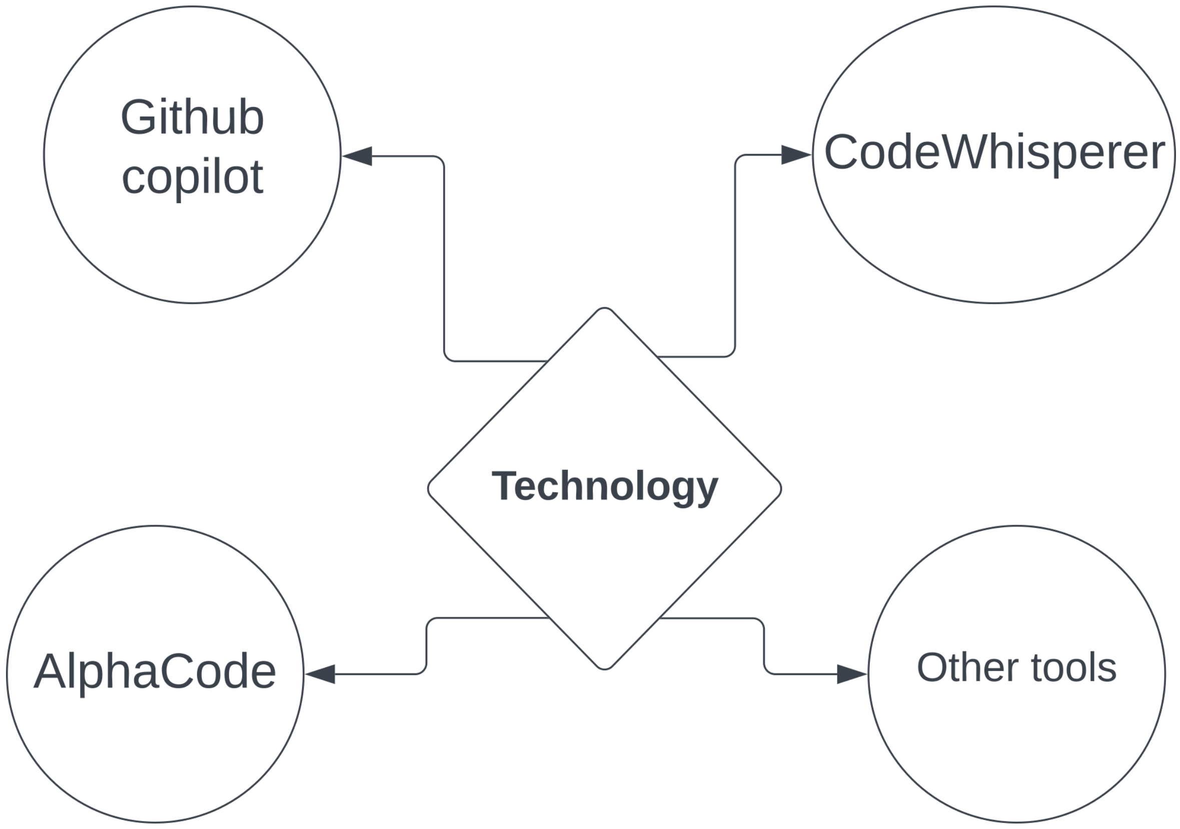

Figure 4.6: Technology dimension

programming contests that follow the same setup of pair programming. If AlphaCode performed well in problems contests environments then most likely it will perform well in the pair programming setup as well.

The third value for this dimension is CodeWhisperer, which is a code generation tool developed by Amazon to assist programmers while developing code. Although there is not much research done on this tool at the time of this study, we added it to the dimensions because it is designed to be a pair programming tool aiding developers. Then we added "Other tools", which are tools used in the paper but it is not any of the previous three tools.

The third dimension is the Experimental design. Shown in Figure 4.7, this dimension explains the type of experiment used in this study. First value is a lab study, it is only repeatable in a lab environment so we can't really tell how this technology is ready for industry. The second value is the simulated environment, which is a virtual environment that replicates the conditions of a real-world research setting, like testing the tool on students developers. Third value is Field study which is a study done outside of the lab in the industry's normal setup. This is a great way to access the possibility of using this technology in the industry. The last value is Secondary study which addresses studies that don't have experiments yet they are secondary studies done of primary studies like surveys and mapping studies.

Then we added another dimension which is the experiment control shown in Figure 4.8, which shows the degree of control in the experiment, we have two values of control, controlled experiment which the researcher control the study group in the experiment, and there is non-controlled experiment, which is an experiment, where the researcher doesn't control the study group in the experiment.

Next, we added another dimension, which is the data collection method shown in Figure 4.9. This dimension access which method they used to collect the data within the study, we have three values, collecting data by interview, survey or by observation. We thought although there is other ways of collecting data, these are the only values that fits with most the studies in this field. The next dimension is the objective shown in Figure 4.10. This dimension is used to know what was the goal of the primary study so what is the objective they were trying to achieve with the tool. The five objectives we are interested in are the ones we mentioned before in the categories part which are code correctness, code usability, code efficiency, code security and developers experience.

The seventh dimension is the task shown in Figure 4.11, which is the task given to the tool to do while accessing it. trying to solve problems with the tool, or pair programming with the tool while working on a programming task or just asking the tool to suggest code. The next dimension is the subject type shown in Figure 4.12, which represents the type of subject used in the study whether this study is done on humans, a dataset, or finally on an oracle. Last we added the final dimension the **time**, in which year this study was published. We are only limiting studies since 2021 so we have three years' values till 2023. This dimension will help us rate the number of publications per year and whether it increased or decreased

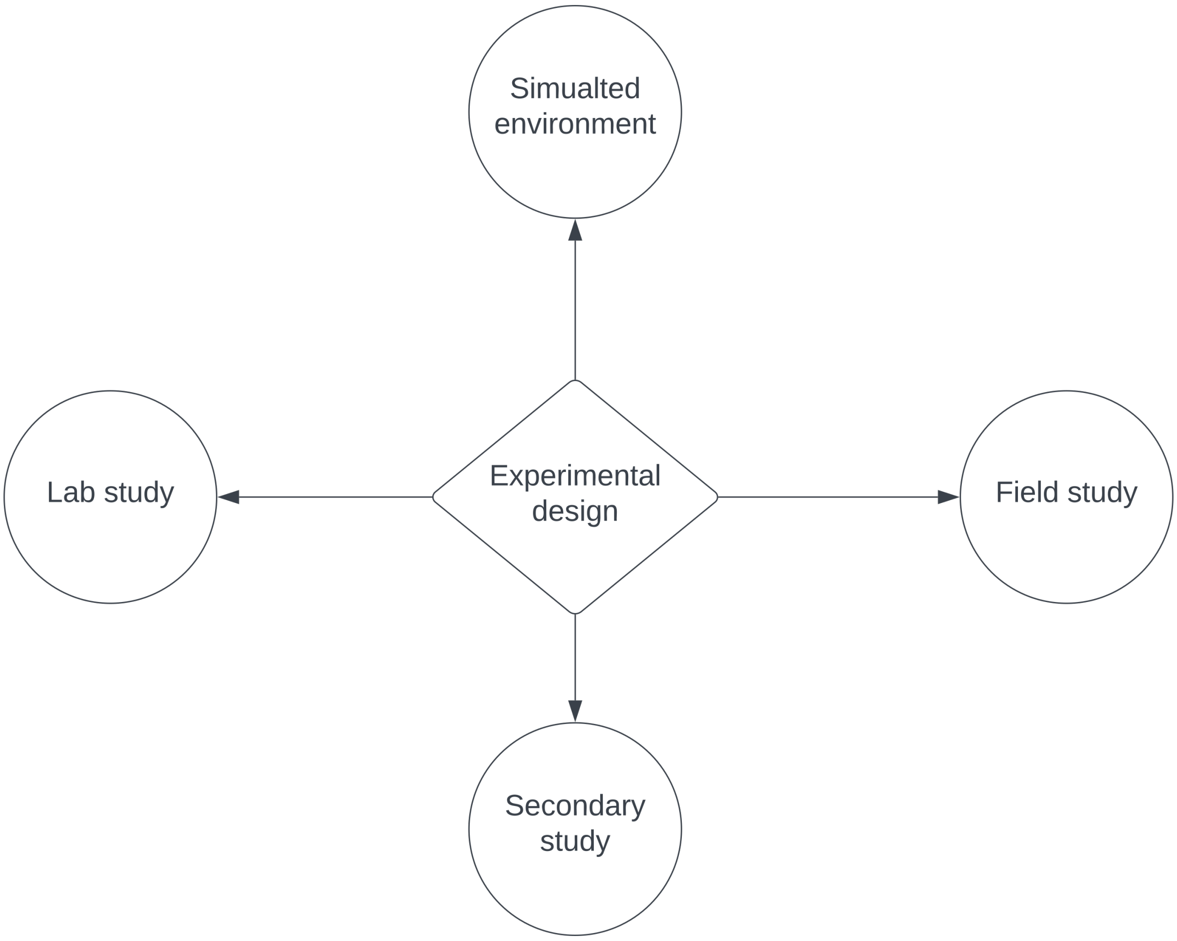

Figure 4.7: Experimental design dimension

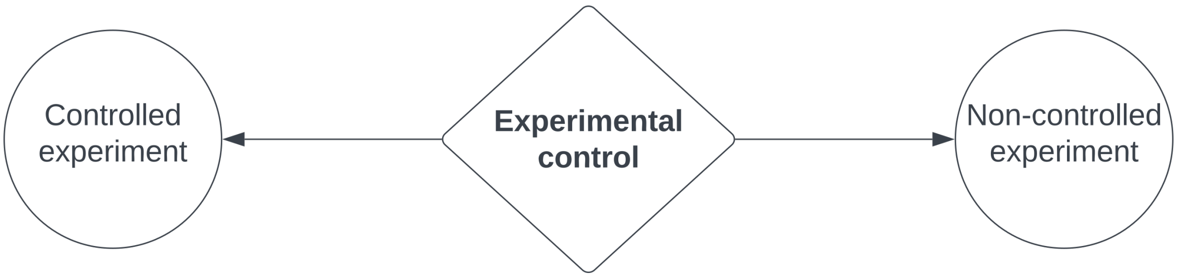

Figure 4.8: Experimental control dimension

over the years.

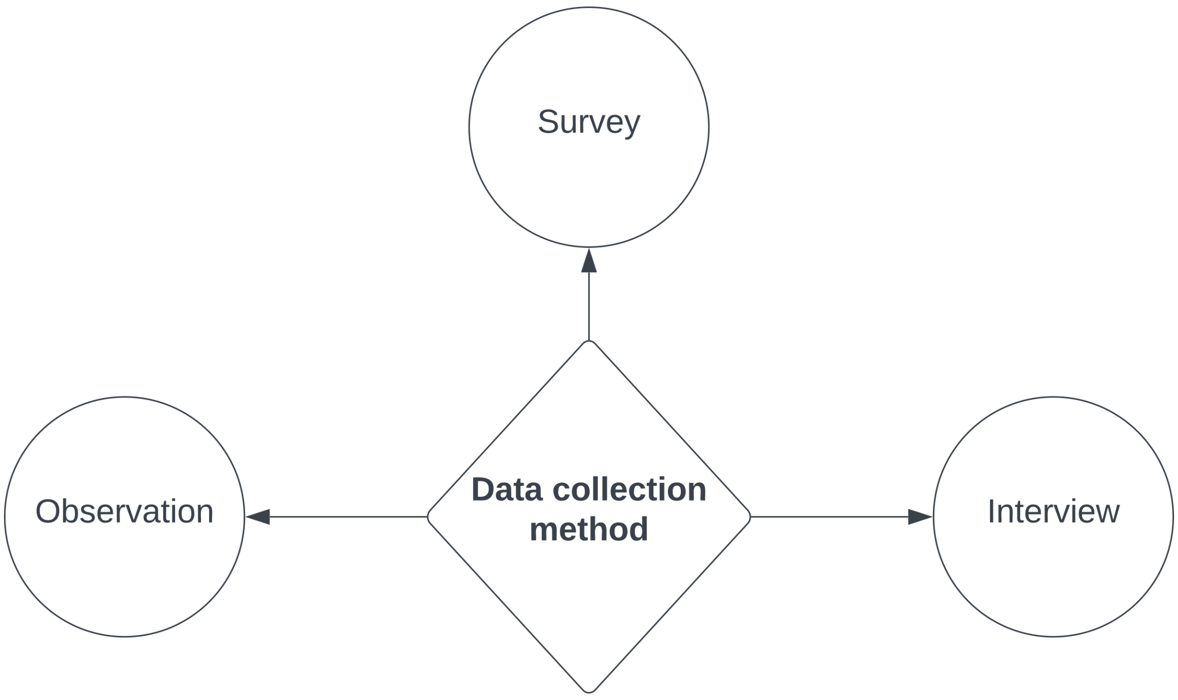

Figure 4.9: Data collection methods dimension

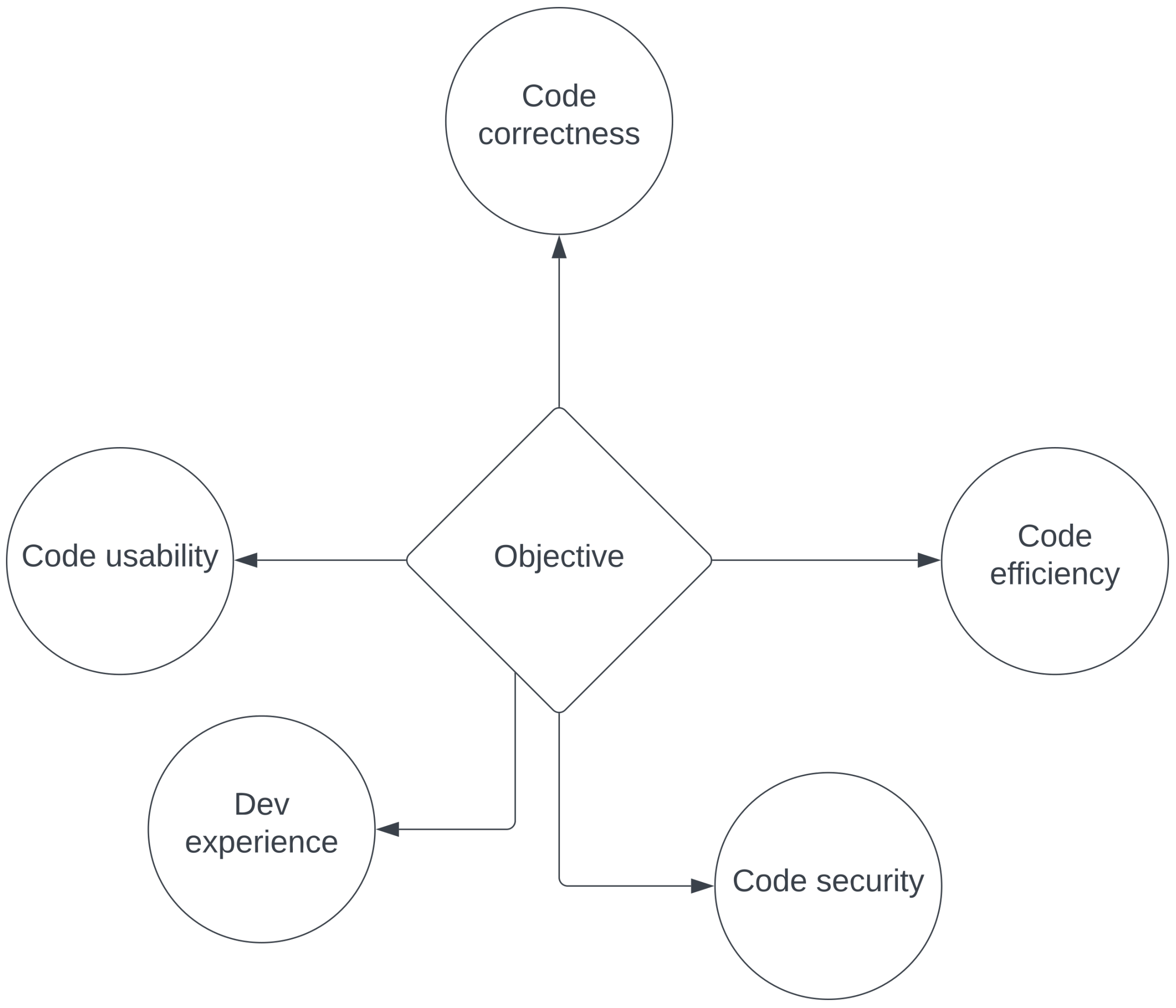

Figure 4.10: Objective dimension

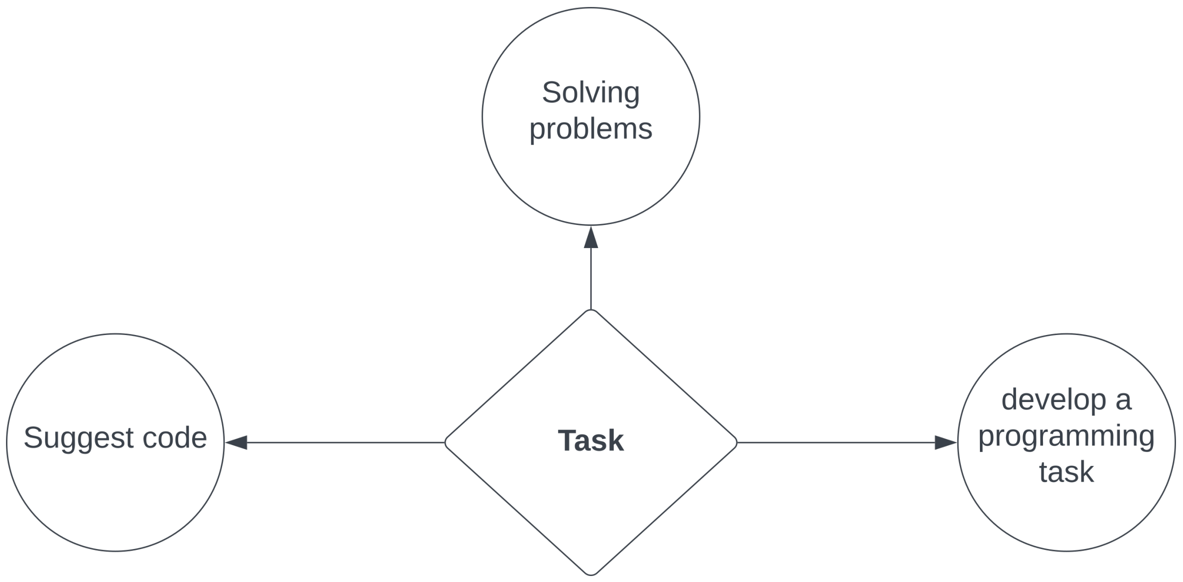

Figure 4.11: Task dimension

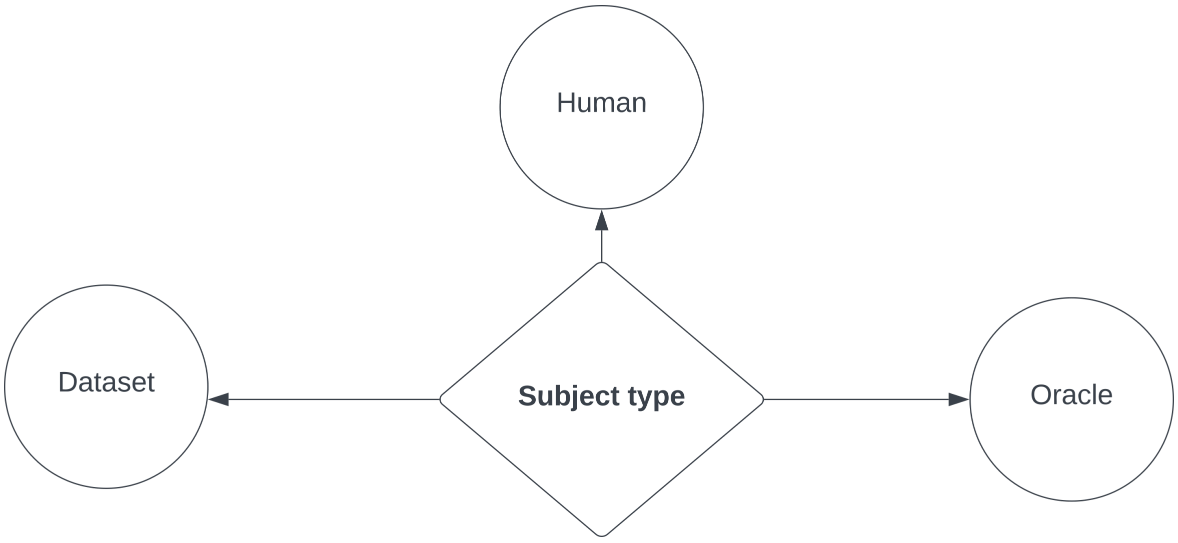

Figure 4.12: Subject type dimension

5 EVALUATION

This chapter is divided into several parts. First part is the process we followed and what were the results we got. Then, there is a part of summary of papers we found with its findings. We added after a part of the mapping study with a table of each column representing each dimension. Then we added a part with the statistics collected from the mapping process. The final part is the discussion of the results.

5.1 PROCESS

We run the query on the different search engines. Then we went through the results and identified papers that were the most relevant to our research scope in the different search databases, IEEE Xplore, ACM, Google scholar and Springer.

To ensure that our search was accurate, we filtered search engines to years between 2021 and 2023. This enabled us to limit our search results to papers that were the most up-to-date and relevant to our research topic. Additionally, we excluded papers that were written in languages other than English.

the final list of papers we got from the search engines with the queries were 16 papers. then we did the manual search that we checked the most frequent conferences websites in software engineering for the last 4 years to find relevant papers and we got more 3 papers.

The last phase in the search part was the snowballing phase. Snowballing in systematic mapping reviews is a method of identifying relevant literature for a systematic review. It involves using the references from existing literature to identify additional studies. This method is used when there is a lack of a comprehensive bibliographic database to search for studies, since it is a new area of research we decided to use this technique. It relies on the references in existing literature to identify additional relevant studies. By following the references in the existing literature, it is possible to build a larger corpus of relevant studies.

This method added 6 more papers relevant to the search scope. so now the final list of papers were 25 papers.

Then, we moved forward with the mapping process where we give each paper out of the final list a value in each category. We added every study in a table where the columns represent the categories and in each cell we added the corresponding value for the column. The table in the result section you can see the whole mapping study.

5.2 PAPERS SUMMARY

GitHub Copilot AI pair programmer: Asset or Liability?[9]

Table 5.1: Paper 1

Reference	P1
Category	B
Reason	comparing the code generated by the tool to those generated by humans.

Summary

They investigated Copilot's abilities in two distinct programming tasks, creating correct and effective solutions for basic algorithmic problems and comparing Copilot's suggested answers with those of human programmers. For the latter, they evaluated Copilot's performance in tackling a few core computer science issues, such as sorting and building simple data structures. They used the copilot to attempt several simple algorithm tasks from the "Introduction to Algorithms" book. Sorting algorithms, data structures, graph algorithms, and advanced design and analysis techniques were the main categories of algorithms they chose.

Conclusion

Their findings demonstrate Copilot's capability to produce accurate and ideal answers to some key algorithm design issues. Nonetheless, the developer's prompt's clarity and depth have a significant impact on the quality of the created programs. Their findings demonstrate that the correct ratio of human solutions is higher than the correct ratio of Copilot, while the Copilot-generated flawed solutions may be fixed with less effort.

Is GitHub's Copilot as Bad As Humans at Introducing Vulnerabilities in Code?[3]

Table 5.2: Paper 2

Reference	P2
Category	B
Reason	comparing security of the code generated to those generated by humans.

Summary

They conducted a comparative empirical investigation of various tools and language models from the standpoint of security. The purpose of this study is to assess the performance of Copilot, a CGT, in comparison to that of human developers. They specifically looked into whether Copilot is equally likely to introduce software flaws as human developers. They made advantage of the Big-Vul dataset, which described as a collection of vulnerabilities added by human developers. They reproduced the situation prior to the bug's introduction for each entry in the dataset and let Copilot provide a completion. Three independent coders personally review the completions to determine whether they contain the same vulnerability.

Conclusion

Their experiments have concluded that Copilot is not as bad as human developers at introducing vulnerabilities in code. They have also observed that Copilot is less likely to generate vulnerable code corresponding to newer vulnerabilities, and is more likely to generate certain types of vulnerabilities. Their findings suggest that Copilot performs better against vulnerabilities with simpler fixes.

What is it like to program with artificial intelligence?[35]

Table 5.3: Paper 3

Reference	P3
Category	A
Reason	checking the tool used in a pair programming context by a survey

Summary

This study is a survey that examines the similarities and differences between programming using large language models (LLM-assisted programming) and earlier conceptualizations of programmer help. They reference previously conducted usability and design studies as well as publicly available experience reports of LLM-assisted programming. They explore potential problems and unresolved research questions associated with bringing big language models to end-user programming, especially for users with little or no programming experience.

Conclusion

LLM aid is similar to a very capable and adaptable compiler, a companion in pair programming, or a seamless search-and-reuse function. LLM-assisted programming, however, has a distinct way in other areas, which creates new difficulties and chances for study into human-centered programming. Supporting non-expert end users in using these tools effectively presents even more difficulties.

Grounded Copilot: How Programmers Interact with Code-Generating Models? [5]

Summary

Table 5.4: Paper 4

Reference	P4
Category	A
Reason	testing the tool in pair programming setting.

They observed 20 participants as they used Copilot to complete several programming tasks designed. Some of the tasks required contributing to an existing code base, which they believe mimics more a realistic software development setting; the tasks also spanned multiple programming, in order to avoid language bias. They then iterated between coding the participants' interactions with Copilot, consolidating their observations into a theory.

Conclusion

their theory proposes that user interactions with Copilot can be classified into two modes- acceleration and exploration. Design recommendations for future programming assistants were provided based on the interviews, such as providing short, high-confidence code suggestions for acceleration mode and better recommendations for exploration mode.

Is GitHub Copilot a Substitute for Human Pair-programming?[14]

Table 5.5: Paper 5

Reference	P5
Category	A
Reason	testing the tool in pair programming setting.

Summary

The focus of their investigation, which involved 21 participants, is on the productivity and quality of the code. A participant was given a project to code for experimental design under three conditions that were delivered in a randomized order. Pair programming with Copilot, human pair programming as a driver, and human pair programming as a navigator. The codes produced by the three trials were examined to ascertain the average number of lines of code added in each condition and the average number of lines of code eliminated in the next step. Whereas the latter gauges the caliber of the generated code, the former gauges the productivity of each condition.

Conclusion

According to the findings, Copilot increases productivity as indicated by the number of lines of code added, but the quality of the code created is lower as indicated by the number of lines of code removed during the subsequent trial.

Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models[38]

Table 5.6: Paper 6

Reference	P6
Category	A
Reason	testing the tool in pair programming setting.

Summary

In this study, 24 participants took part in a within-subjects user study to understand the use and opinion of Copilot, a LLM-based code generation tool. This research explored how Copilot affects the programming experience, how users recognize errors in code generated by Copilot, what coping mechanisms users employ when they find errors in code generated by Copilot, and what risks are associated with using Copilot.

Conclusion

It was found that, even though Copilot did not necessarily speed up the completion time or success rate, many participants still chose to use it in their programming tasks as it gave a useful starting point and saved them the trouble of searching online. However, they had trouble understanding, modifying, and debugging the code snippets generated by Copilot, which impeded their task-solving performance. Suggestions to enhance the design of Copilot were put forward based on the observations and the feedback of the participants.the majority of the participants (19 out of 24) preferred using Copilot.

Security Implications of Large Language Model Code Assistants: A User Study[34]

Table 5.7: Paper 7

Reference	P7
Category	A
Reason	testing the tool in pair programming setting.

Summary

58 computer science undergraduate and graduate students with programming backgrounds, were split randomly into 'control' (no Codex LLM access) and 'assisted' (with Codex LLM access) groups. Given the high frequency of memory-based errors in low-level

languages such as C and C++ and their relative severity, they designed a study requiring participants to complete a set of 12 functions that perform basic operations on a linked list representing a "shopping list" in C.

Conclusion

Their findings suggest that the LLM has a positive impact on functional correctness, and does not raise the rate of serious security bugs. This result is somewhat unexpected given the existing published studies on how LLMs can recommend vulnerable code. When considering the origin of bugs detected, the data implies that users do not use the added productivity benefits to fix them - while they may change aspects of the suggestions.

On the Robustness of Code Generation Techniques: An Empirical Study on GitHub Copilot.[23]

Table 5.8: Paper 8

Reference	P8
Category	B
Reason	compared the code generated to code developed by human developers.

Summary

This paper presents an empirical study that examined if semantically equivalent natural language descriptions result in the same recommended function. They used Copilot to generate 892 Java methods starting from their original Javadoc description, and generated different semantically equivalent descriptions both manually and automatically.

Conclusion

The results show that modifying the description results in different code recommendations in 46 % of cases, and differences in the semantically equivalent descriptions might impact the correctness of the generated code (±28 %).These findings suggest that testing the robustness of DL-based code recommenders may play an important role in ensuring their usability and in defining possible guidelines for the developers using them.

Productivity Assessment of Neural Code Completion.[44]

Summary

This paper investigates whether usage measurements of developer interactions with GitHub Copilot can be used to predict perceived productivity as reported by developers. Analysis of 2,631 survey responses and usage measurements collected from the IDE reveals that an acceptance rate of shown suggestions is a better predictor of perceived productivity

Table 5.9: Paper 9

Reference	P9
Category	A
Reason	tested the tool in a pair programming environment through surveys

than alternative measures.

Conclusion

It is found that acceptance rate varies significantly across the developer population and over time. The results suggest that acceptance rate can be used to monitor performance of a neural code synthesis system, although other approaches are still needed for further investigation due to the many human factors involved.

An Empirical Evaluation of GitHub Copilot's Code Suggestions [25]

Table 5.10: Paper 10

Reference	P10
Category	C
Reason	tested the correctness of the code generated by the tool.

Summary

They build Copilot inquiries in four different programming languages using 33 LeetCode questions. They run the tests provided by LeetCode to determine whether the 132 correct Copilot solutions pass them, and they assess usability using SonarQube's cyclomatic complexity and cognitive complexity metrics. Java ideas from Copilot have the highest correctness score (57 %) while JavaScript proposals have the lowest (27 %). Generally, Copilot's recommendations are simple, and there aren't any significant distinctions across the programming languages. They also uncover other possible Copilot flaws, such as the creation of overly complex code and the reliance on undefined auxiliary functions.

Conclusion

RQ1 Summary: Copilot's correctness (passing all tests) varies by language, with Java as highest (57 %) and JavaScript lowest (27 %).
RQ2 Summary: The median cognitive complexity and cyclomatic the complexity of Copilot solutions is 6 and 5, respectively, with no statistically significant differences between languages.

Assessing the Quality of GitHub Copilot's Code Generation [42]:

Table 5.11: Paper 11

Reference	P11
Category	C
Reason	tested the correctness of the code generated by the tool.

Summary

An experimental setup was created to evaluate the generated code in terms of validity, correctness, and efficiency. In this experiment, the HumanEval dataset was used which contained 164 problems with task ID, prompt, canonical solution, and unit tests. The code generation step was manually implemented and the Python 3.8 interpreter was used to check for code validity. The number of passed unit tests was measured and divided by all unit tests for a particular problem to assess code correctness. The OpenAI API was utilized to obtain the time and space complexities of the canonical solution and the generated code.

Conclusion

The results revealed that GitHub Copilot generated valid code with a 91.5 % success rate, and was able to correctly generate 47 (28.7 %) out of 164 problems, partially correctly generate 84 (51.2 %), and incorrectly generate 33 (20.1 %). The authors concluded that GitHub Copilot is a promising tool, however further and more comprehensive assessment is needed in the future.

Systematically Finding Security Vulnerabilities in Black-Box Code Generation Models [12]

Table 5.12: Paper 12

Reference	P12
Category	C
Reason	tested the security of the code generated by the tool.

Summary

They proposed an approach for automatically detecting security vulnerabilities in code generation models. This is done through a novel black-box model inversion approach. They identified thousands of vulnerabilities in state-of-the-art code generation models, such as the widely used GitHub Copilot. At the time of publication, they published a set of security prompts to investigate the security weaknesses of the models and compare them in various security scenarios. They also released their approach as an open-source tool which can be used to evaluate the security of the black-box code generation models.

Conclusion

The study showed that their method can identify numerous security vulnerabilities in code generation models. To encourage further research, they publish a benchmark of promising non-secure prompts generated by CodGen and Codex models. The benchmark includes 381 non-secure prompts from CodeGen and 537 non-secure prompts from Codex that can be used to evaluate and compare vulnerabilities of the models related to different CWEs.

An Empirical Evaluation of Competitive Programming AI: A Case Study of AlphaCode[19]

Table 5.13: Paper 13

Reference	P13
Category	B
Reason	assessing the correctness of the code generated compared to humans.

Summary

This paper presents an empirical study to compare the similarities and performance differences between codes generated by AlphaCode and human codes. They were to trying to assess how the generated codes similar to human codes, and if they can perform better. To answer this, they examined the source code similarity, execution time, and memory usage of generated and human codes. The dataset includes 44 generated codes that solve 22 problems written in C++ and Python, and 21,508 C++ and 10,228 Python human codes from Codeforces. They implemented a mechanism to measure the execution time and memory usage of generated and human codes.

Conclusion

The results indicate that the generated codes from AlphaCode are similar to human codes, with an average maximum similarity score of 0.56. The performance of the generated code was found to be on par with or worse than the human code in terms of execution time and memory usage. They observed that AlphaCode tends to generate more similar codes to humans for low-difficulty problems, while employing excessive nested loops and unnecessary variable declarations for high-difficulty problems, resulting in low performance.

Github Copilot in the Classroom: Learning to Code with AI Assistance [32]

Summary

Table 5.14: Paper 14

Reference	P14
Category	B
Reason	assessing the correctness of the code generated compared to humans.

In this experiment, Copilot was used to help solve coding assignments from a Python course designed to introduce algorithmic problem solving to students with no programming experience. The first course was CS0, with eight programming assignments covering common introductory topics. Additionally, they added a second course, DS1, that was an introductory course in data science with seven data-oriented programming assignments. The DS1 assignments used non-standard Python libraries that are commonly used in data science, as well as non-programming components such as ethics write-ups.

Conclusion

The study found that Copilot is user-friendly and accurate enough for novice programmers to use in solving fundamental programming tasks. In an introductory data science course, Copilot-generated solutions received scores between 68% and 95% when human-graded, and were difficult to distinguish from student-authored solutions. The authors encourage computer science educators to integrate AIDEs into their coursework and development workflow.

Using GitHub Copilot to Solve Simple Programming Problems. [39]

Table 5.15: Paper 15

Reference	P15
Category	B
Reason	assessing the correctness of the code generated compared to humans.

Summary

This paper is a report of the author experience in using Copilot as if he were a student tasked with writing code and tests for a given CS1 problem. They tried to assess how Copilot perform, compared to Davinci, in terms of the correctness and variety of the generated code, tests and explanations. They also tried to answer, if a suggestion is incorrect, can Copilot be interactively led to a correct one.

Conclusion

It is found that Copilot fares worse than Davinci in both accounts. It also looks at if Copilot can be interactively led to a correct answer, and concludes that using Copilot is often a 'hit and miss' affair. The paper notes that Codex and Copilot pose challenges to academic integrity which educators must adapt to, and that detecting and punishing the use of Copilot is an important topic to consider.

The Impact of AI on Developer Productivity: Evidence from GitHub Copilot.[28]

Table 5.16: Paper 16

Reference	P16
Category	A
Reason	tested the tool in a pair programming environment.

Summary

This paper presents results from a controlled experiment with GitHub Copilot. Recruited software developers were asked to implement an HTTP server in JavaScript as quickly as possible. They examined the productivity effects of AI tools on software development and carries out a controlled trial of GitHub Copilot. The trial involves programmers being assigned to implement an HTTP server in JavaScript as quickly as possible. The treated group had access to GitHub Copilot while the control group did not, but were free to search the internet and use Stack Overflow.

Conclusion

On average, participants in both treated and control groups estimated a 35% increase in productivity, which is an underestimation compared with the 55.8% increase in their revealed productivity. This result provides indirect evidence that treated group benefited from Copilot during their task as their productivity significantly higher than the control group.

SecurityEval Dataset: Mining Vulnerability Examples to Evaluate Machine Learning-Based Code Generation Techniques. [37]

Table 5.17: Paper 17

Reference	P17
Category	C
Reason	assessing the security of the code generated by the tool.

Summary

This paper introduces SecurityEval, a dataset created for assessing the performance of machine-learning-based code generation models from a security standpoint. Compiled from Python samples, the dataset encompasses 75 distinct vulnerability types from the Common Weakness Enumeration (CWE) in 130 samples. The samples are formatted as prompts suitable to be used in a generalized source-code generation model. They also demonstrated using their dataset to evaluate one open-source (i.e., InCoder) and one closed-source code generation model(GitHub Copilot). The authors demonstrate how to use the SecurityEval prompts to evaluate an open-source code generation model (InCoder) and a closed-source code generation tool (GitHub Copilot).

Conclusion

Although a code generation model can help software engineers to develop software quickly, the generated code can contain security flaws. Their dataset has 130 Python code samples spanning 75 types of vulnerabilities (CWEs). They demonstrated that the dataset, when combined with static analyzers, could be used to automate or semi-automate the evaluation of the security of generated code.

How Readable is Model-generated Code? Examining Readability and Visual Inspection of GitHub Copilot[21].

Table 5.18: Paper 18

Reference	P18
Category	B
Reason	assessing the usability of the code generated compared to human.

Summary

In this paper, they have conducted an empirical study of GitHub Copilot using eye tracking in a natural software development environment (VS Code IDE). The focus was on the readability of and visual inspection of Copilot generated code. An experiment involving 21 participants was performed to compare code written with Copilot to code written by human pair-programmers (as driver and navigator). Static code analysis, human readability annotators, and eye tracking were used to check the readability of Copilot generated code compared to code written completely by human programmers and how the programmers inspect Copilot generated code.

Conclusion

The results of this study indicate that model generated code is of similar complexity and readability compared to code written by human pair programmers. However, the eye tracking data suggest that programmers tend to pay less attention to model generated code, which is statistically significant. Thus, the conclusion is that it is more important for programmers to read the code and be aware of complacency and automation bias when using model generated code.

Conversing with Copilot: Exploring Prompt Engineering for Solving CS1 Problems Using Natural Language.[11].

Table 5.19: Paper 19

Reference	P19
Category	C
Reason	assessing the correctness of the code generated by the tool.

Summary

They tested Copilot on 166 programming problems by copying the problem description into a Visual Studio Code editor and observing the suggested code. If any test cases failed, the authors deleted the code and modified the problem description using natural language. They observed the failing test cases and engineer the description by adding comments to it that clarify the problem or that provide a strategy for solving the problem. They did not modify any code. These steps were repeated until all test cases passed or no more clarifications could be made.

Conclusion

They found that it was able to solve about half of these problems on its first attempt, and 60 percent of the remaining problems with only slight changes to the problem description using natural language. They think that this type of prompt engineering, which is likely to become a regular occurrence when Copilot initially fails, can be a helpful way of teaching computational thinking skills and could alter the way code writing skills are developed. More than half of the remaining problems were resolved by adjusting the prompts to incorporate algorithmic hints. This technique was successful across nearly all categories of problems and is viewed as having educational importance because it requires students to examine code errors, transform abstract ideas from problem descriptions into concrete computational steps, and communicate them clearly in natural language.

Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions [26].

Summary

They tested Copilot on code related to high-risk cyber security weaknesses, such as those from the MITRE Top 25 CWE list. 89 different scenarios were produced, resulting in

Table 5.20: Paper 20

Reference	P20
Category	C
Reason	assessing the security of the code generated by the tool.

1,689 programs. Of these. The main contributions of this research were: analysing Copilot's software and hardware code completion behaviour when given prompts that are related to security. Understanding how context can influence the AI's code generation and confidence and providing advice to software and hardware designers, especially those who are not familiar with security, about using AI pair programming tools.

Conclusion

40 percent were found to be vulnerable. The study's findings suggest that even though Copilot is capable of producing a vast quantity of code quickly, developers must exercise caution when using it as a co-pilot. It is recommended that Copilot be paired with security-conscious tools during both the training and generation phases to prevent security vulnerabilities.

An Empirical Study of Code Smells in Transformer-based Code Generation Techniques[36]

Table 5.21: Paper 21

Reference	P21
Category	C
Reason	assessing the security of the code generated by the tool.

Summary

The study examines the presence of harmful code smells in training sets and the output of transformer-based code generation models for Python, using three open-source datasets and two code generation tools. The paper provides a comprehensive empirical analysis of the occurrence of code smells, a comparison of open-source and closed-source techniques, and a discussion of the implications of the findings for researchers and practitioners. The researchers collected various datasets used to train code generation models and tested them for code smells, security issues, and manual validations. They then took samples of the code generated by the tools and analyzed them to detect if the same issues existed. They tested copilot to identify code smells, which can be violations of code conventions or security concerns mapped to a Common Weakness Enumeration, they employed Pylint and Bandit, respectively.

Conclusion

In three frequently used datasets, Pylint detected a total of 264 distinct types of non-security code smells, with the most common ones being "Undefined variables," "line too long," "too few public methods," and "bad indentation." The fine-tuned GPT-Neo model's output contains code smells. The most frequent non-security smells in both the training set and the output are Undefined variables, lines too long, Duplicate code, and Unused argument. The output also has security smells, such as the use of assert statements. The suggestions generated by GitHub Copilot are executable, but they often contain code smells that indicate sub optimal coding practices and security vulnerabilities.

CCTEST: Testing and Repairing Code Completion Systems [20]

Table 5.22: Paper 22

Reference	P22
Category	C
Reason	assessing the effectiveness of the code generated by the tool.

Summary

The article introduces CCTEST, an automated testing and enhancement framework for code completion systems that identifies erroneous cases and enhances code output by selecting the one closest to the "average" appearance. The framework treats code completion systems as a "black box" and tests eight widely-used LLM-based code completion systems. CCTEST is implemented in Python, with about 5k LOC, and focuses on mutating Python code. The program is parsed using tree-sitter, and feasible PSC transformations are applied to generate transformed prompts. The evaluation dataset is formed from LeetCode solution programs and CodeSearchNet, with programs selected based on token length and complexity. Overall, the dataset contains 613 code snippets from LeetCode and the test split from CodeSearchNet.

Conclusion

CCTEST generates valid and consistent prompt mutants that can help improve code completion systems. It identifies defects in different systems and recommends using a threshold of T = 9 for best results. All PSC transformations are shown to be effective, finding 33,540 programs exposing code completion errors. With enhancement, the average performance of code completion systems is notably increased by 40.25% and 67.43 %.

Jigsaw: Large Language Models meet Program Synthesis [15]

Table 5.23: Paper 23

Reference	P23
Category	C
Reason	assessing the effectiveness of the code generated by the tool.

Summary

They discuss their program and assessment of a their tool Jigsaw, designed to generate Python Pandas API code using inputs from multiple sources. They found that as big language models advance in generating code from intent, Jigsaw can contribute to enhancing the precision of the code generation systems. They measure the accuracy of their program synthesis approach by evaluating how many of the specifications in the dataset have been correctly synthesized into programs. A program is considered correct if it satisfies the input/output examples and meets the intent of the natural language description, as confirmed through manual inspection. To address randomness in the output of the models, the evaluation is conducted three times, and the mean accuracy and standard deviation are calculated.

Conclusion

Jigsaw's performance was tested on the TensorFlow dataset, where Codex alone solved 8 out of 25 tasks, but variable transformation improved the performance to 15 tasks. The authors found that Semantic Repair has the potential to enhance the performance to 19 tasks. the proposed pre-processing and post-processing modules were found to be effective and generalized to other libraries and programming languages.

From Copilot to Pilot: Towards AI Supported Software Development [31]

Table 5.24: Paper 24

Reference	P24
Category	B
Reason	testing the code generated to code developed by human developers

Summary

This study seeks to analyse the performance of Copilot. It investigates how Copilot performs compared to a human and the boundaries of the AI-supported tool. It introduces a taxonomy of software abstraction hierarchies and evaluates Copilot's code suggestions on 25 Pythonic idioms, sourced from renowned Python developers, and 25 best practices in JavaScript, sourced from the AirBNB JavaScript coding style guide. They looked at Copilot's top code suggestion and whether the idioms and best practices are listed in the tool's ten viewable suggestions. They also explored Copilot's code suggestions for code smells. The study aims to help understand in which areas Copilot performs better than a human, and in which areas it performs worse.

Conclusion

They found that most generated code snippets contain insecure code (about 68% and 74% of code generated by InCoder and Copilot, respectively). They argued that while AI-supported software can be useful in managing coding syntax and warnings, more abstract matters such as code smells, language idioms and design rules are not yet resolved. They argued that AI-supported software development, where AI aids designers and developers in complex software engineering tasks, can be achieved. They concluded that AI can support designers and developers in more complex software development tasks, but there is still a lot of improvement that need to be achieved.

Taking Flight with Copilot: Early Insights and Opportunities of AI-Powered Pair-Programming Tools [7]

Table 5.25: Paper 25

Reference	P25
Category	C
Reason	testing the dev experience of using the tool

Summary

They conducted three studies to investigate how developers use Copilot: an analysis of forum discussions from early users, a case study of Python developers using Copilot for the first time, and a large-scale survey of Copilot users to measure its impact on productivity. The Copilot case study was conducted over two days, with researchers spending an hour with each participant. Participants in the Copilot case study were asked to launch Copilot via a pre-configured Code space , with descriptions and guidance given to point out features of Copilot. The function accepted an integer and returned whether it was a prime number.Participants understood the basics of Copilot and its functionality, they were asked to build a tic-tac-toe game with specific criteria. Then, they were asked to build a feature to email the researcher when a game is complete.

Conclusion

This study of Copilot revealed that developers spend more time reviewing code (as suggested by Copilot or other tools) than writing code. As AI-powered tools become more integrated into software development tasks, developer roles will shift so that more time is spent assessing suggestions related to the task than completing the task itself. In the context of Copilot there is a shift from writing code to understanding code, and initial user studies suggest that this is an efficient way of working. However, this assumption may not always hold in different contexts and tasks, so finding ways to help developers understand and assess code and its context will be important.

5.3 RESULTS

5.3.1 *Mapping study*

In this section we will do the mapping study and present the table shown below of the mapping study.

Table 5.26: Mapping every paper to the corresponding methodology, technology and experimental design

Paper	Research Methodology	Technology	Experimental design
P1	Quantitative	Copilot	Simulated environment
P2	Quantitative	Copilot	Lab Study
P3	Qualitative	Copilot	Secondary study
P4	Mixed	Copilot	Simulated environment
P5	Quantitative	Copilot	Field study
P6	Mixed	Copilot	Field study
P7	Quantitative	Copilot	Simulated environment
P8	Quantitative	Copilot	Lab Study
P9	Mixed	Copilot	Field Study
P10	Quantitative	Copilot	Field Study
P11	Quantitative	Copilot	Lab Study
P12	Mixed	Copilot	Lab Study
P13	Quantitative	AlphaCode	Lab Study
P14	Quantitative	Copilot	Simulated environment
P15	Mixed	Copilot	Simulated environment
P16	Quantitative	Copilot	Simulated environment
P17	Quantitative	Copilot	Lab study
P18	Mixed	Copilot	Simulated environment
P19	Quantities	Copilot	Lab study
P20	Quantitative	Copilot	Lab study
P21	Mixed	Copilot	Lab study
P22	Quantitative	Copilot	Lab study
P23	Quantitative	Other	Lab study
P24	Quantitative	Copilot	Simulated environment
P25	Qualitative	Copilot	Simulated environment

Table 5.27: Mapping every paper to the corresponding experimental control, subject type and data collection method

Paper	Experimental control	Subject Type	Data collection method
P1	Controlled	Human	Observation
P2	Controlled	Dataset	Observation
P3	Controlled	Dataset	Survey
P4	Controlled	Human	Interview
P5	Controlled	Human	Observation
P6	Controlled	Human	Observation/Interview
P7	Controlled	Human	Observation
P8	Controlled	Dataset	Observation
P9	Non-Controlled	Human	Survey
P10	Controlled	Oracle	Observation
P11	Controlled	Oracle	Observation
P12	Controlled	Oracle	Observation
P13	Non-Controlled	Human	Observation
P14	Controlled	Human	Observation
P15	Controlled	Human	Observation
P16	Controlled	Human	Observation
P17	Controlled	Dataset	Observation
P18	Controlled	Human	Observation
P19	Controlled	Oracle	Observation
P20	Controlled	Dataset	Observation
P21	Controlled	Dataset	Observation
P22	Controlled	Oracle	Observation
P23	Controlled	Oracle	Observation
P24	Controlled	Human	Observation
P25	Controlled	Human	Interview/Survey

Table 5.28: Mapping every paper to the corresponding task, time and objective

Paper	Task	Time	Objective
P1	Solving problems	2023	Code correctness/efficiency
P2	Suggest Code	2023	Code security
P3	Suggest Code	2022	Dev experience
P4	Develop a programming task	2022	Dev experience
P5	Develop a programming task	2022	Dev experience
P6	Develop a programming task	2022	Code efficiency/Dev experience
P7	Develop a programming task	2023	Code security
P8	Suggest Code	2023	Code correctness
P9	Develop a programming task	2022	Dev experience
P10	Solving problems	2022	Code correctness/usability
P11	Solving problems	2022	Code correctness
P12	Suggest Code	2023	Code security
P13	Solving problems	2022	Code efficiency
P14	Solving problems	2022	Code correctness
P15	Solving problems	2023	Code correctness
P16	Develop a programming task	2023	Dev experience
P17	Suggest code	2022	Code security
P18	Develop a programming task	2022	Code usability
P19	Solving problems	2022	Code correctness
P20	Suggest code	2022	Code security
P21	Suggest code	2022	Code security
P22	Suggest code	2022	Code efficiency
P23	Suggest code	2021	Code efficiency
P24	Suggest code	2023	Code usability
P25	Develop a programming task	2023	Dev experience

5.3.2 *Statistical results*

Figure 5.1 shows that out of the 25 studies analyzed, 7 were categorized under category A, while 8 were classified under category B. The remaining 10 studies were mapped to category C.

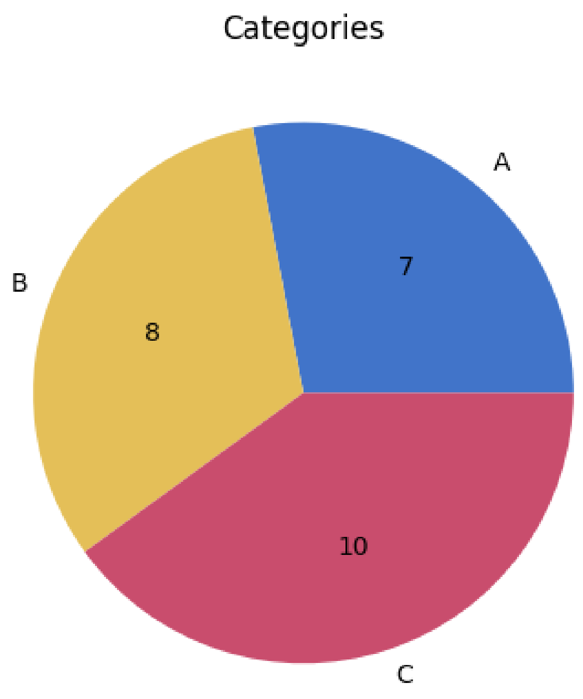

Figure 5.1: Categories mapping statistics

Examining the first dimension, time, the results changed over the years. In 2021, only one study were selected for the final list of papers. The majority of the results emerged in 2022. In 2023, nine papers have been included in the final list. These statistics can be seen in Figure 5.2

The second dimension we analysed is the technologies used in the studies. The majority of research was done using GitHub Copilot with 23 studies. Only one research paper has examined AlphaCode, and there were no papers on CodeWhisperer. We only choose one paper in the analysis were based on other language models using Codex and Gpt. These statistics can be seen in Figure 5.3

Then, we analysed the objective used in the studies. We were mainly concerned with 5 objectives. Code correctness which had 7 occurrences in the chosen paper. Code efficiency was examined 5 times in the papers. code usability only occurred 3 times in the final list

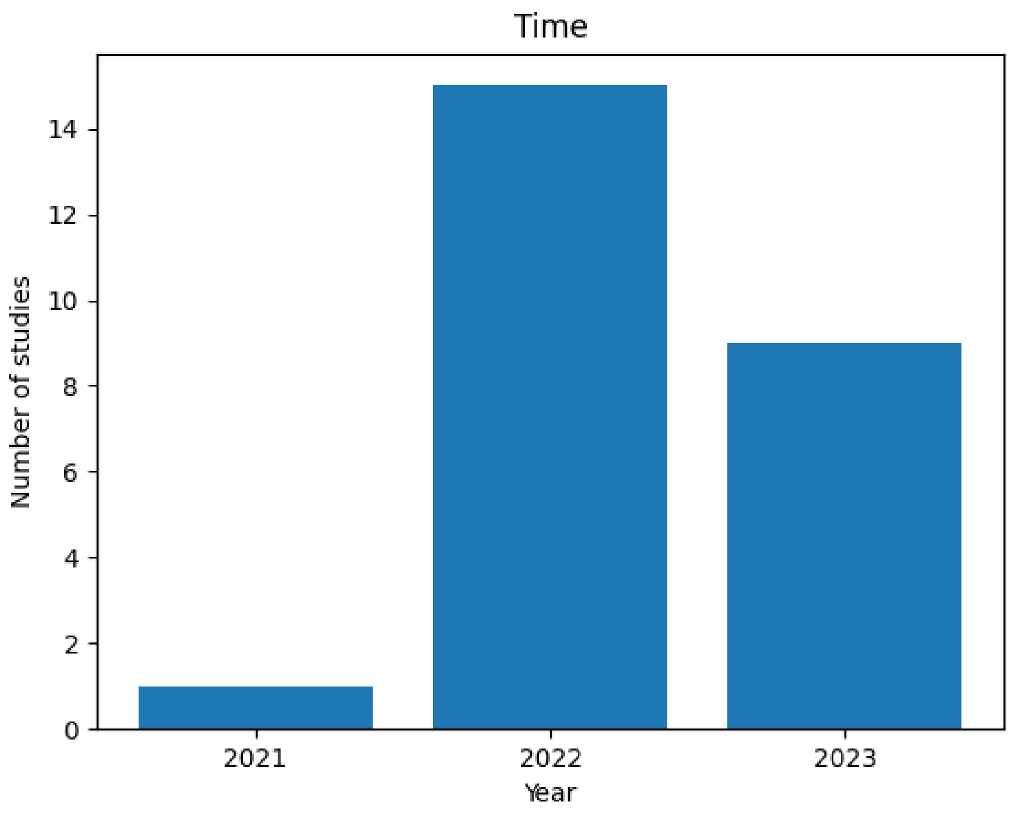

Figure 5.2: Change over time statistics

and code security 6 times Finally, developers experience was examined 7 times in the final list. These statistics can be seen in Figure 5.4

Then, we analysed the design used in the studies. We were mainly concerned with 4 designs. Lab study was is the most used design in the papers with 11 occurrences. Simulated environment was occurred in only 9 studies. Field study occurred in 4 studies. The final value secondary study occurred only in one study. These statistics can be seen in Figure 5.5

Then, we analysed the experiment control found in the studies. We found 23 experiments out of the 25 experiments were controlled while the other 2 were non controlled. These statistics can be seen in Figure 5.6

Then, we analysed the subject type. We were concerned with 3 subject types. Human subject type were found 13 times in the studies. We found 6 dataset out of the 25 subject types. Only 6 studies out of the total number of papers were found to be using an oracle for testing. These statistics can be seen in Figure 5.7

Then, we analysed the Data collection type. We were concerned with 3 collection types. Observation type were found 21 times in the studies. We found 3 survey out of the 25

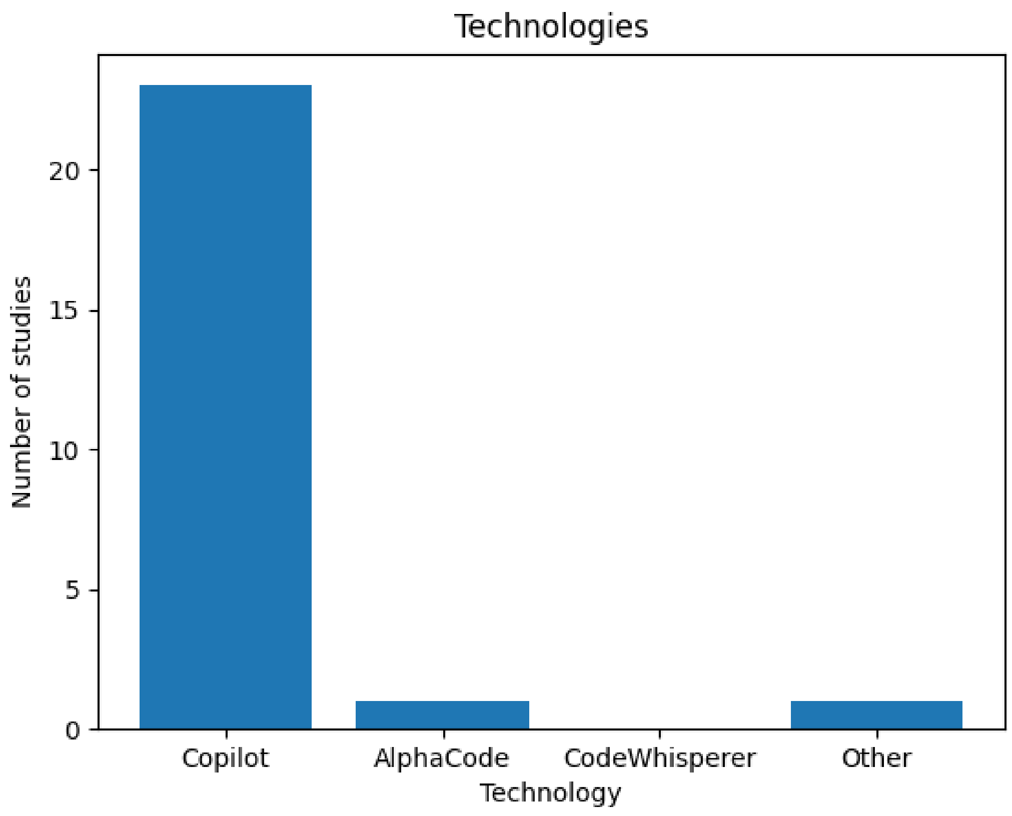

Figure 5.3: Used technologies statistics

subject types. Only 3 studies out of the total number of papers were found to be using an Interviews for testing. These statistics can be seen in Figure 5.8

Then, we analysed the Task. We were concerned with 3 tasks. Solving problems task was found 7 times in the studies. We found 8 developing programming tasks out of the 25 papers. 10 studies out of the 25 papers were using suggest code. These statistics can be seen in Figure 5.9

Then, we analysed the methodology. We were concerned with 3 methodologies. quantitative were found in 16 studies. We found 7 mixed methodologies out of the 25 papers. Only 2 study out of the 25 papers were using Qualitative. These statistics can be seen in Figure 5.10

5.4 DISCUSSION

Out of the 25 studies analyzed, 7 tested the tools in a pair programming environment, while 8 studies compared the code generated by the tools to code written by human developers. The high number of studies that fit within this scope may be because these tools are promoted as "AI pair programmers" and are intended to be used in this specific function.

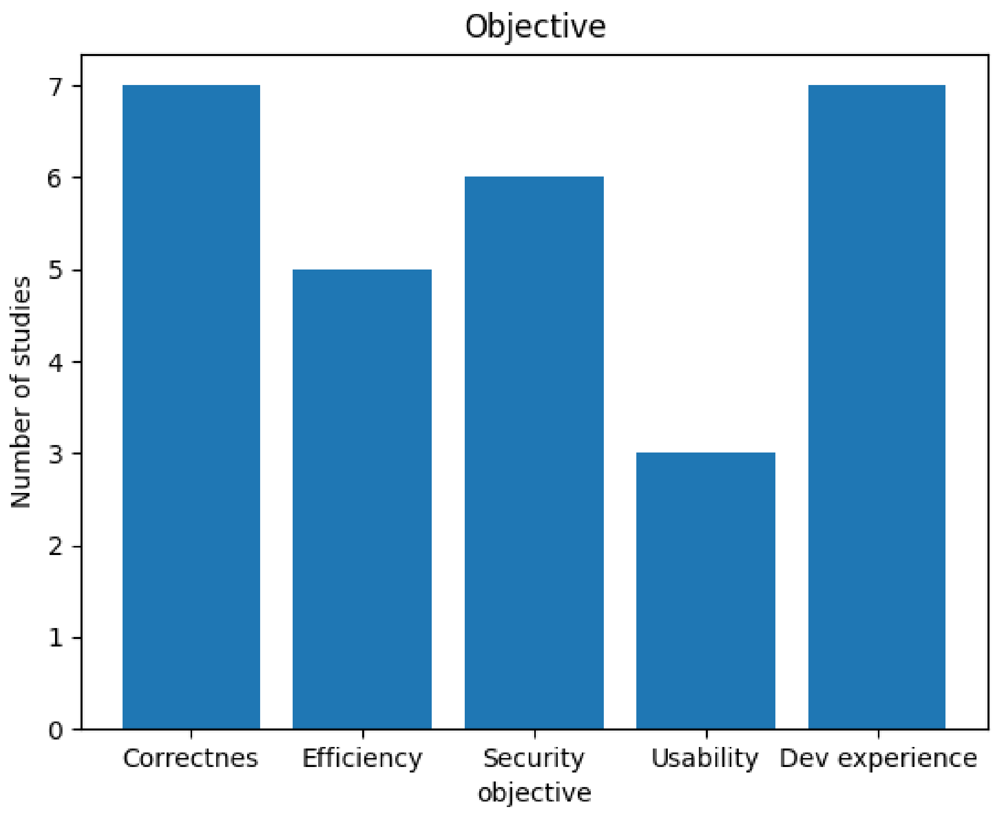

Figure 5.4: Different objectives statistics

Finally, 10 studies tested both the performance of the tools and the developers' experience. This category had the highest percentage, which may be because these tools are relatively new and still in the early stages of testing.

The number of studies on this topic changed over time, from 1 in 2021 to 15 in 2022. This can be attributed to the release of Github Copilot, which is currently the state of the art tool in this field, at the end of 2021. As a result, research on this specific tool increased significantly in 2022.

However, the number of studies dropped to 9 in 2023. This can be explained by the fact that the study only covered the period up until April 2023, which is one-third of the year. It is expected that this number will significantly increase by the end of 2023, as this field is currently receiving a lot of attention, specially after the release of other tools like codeWhisperer.

The second analysis focused on the technology used in each study. Out of the total number of studies, 23 used Copilot in their analysis. This can be attributed to the fact that Copilot is currently the state of the art tool in this field and is being promoted by Microsoft as the "AI pair programmer". AlphaCode, on the other hand, only had one study included

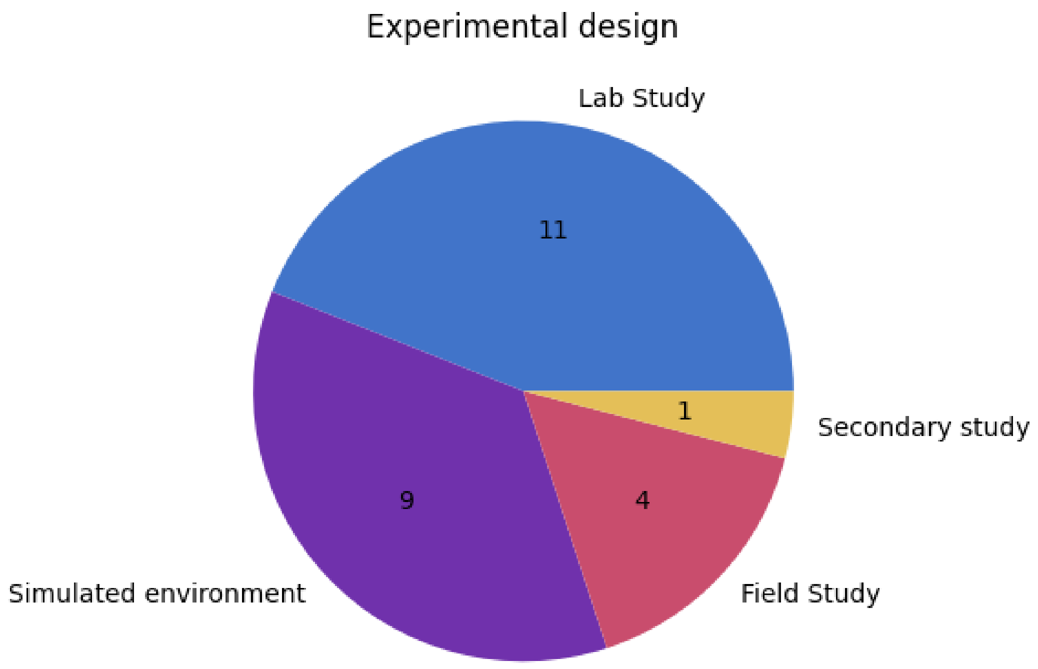

Figure 5.5: Experiment design statistics

in the final list.

This could be due to the fact that AlphaCode is not as popular as Copilot at the moment and its main use is for solving programming contest problems. CodeWhisperer did not have any studies included in the analysis, which can be explained by the fact that it was only released in the second half of 2022 and didn't get as much as Copilot. Only one study was included that used a different tool, which utilized Codex and GPT.

The third analysis focused on the objective of the studies. The most common objective in the papers was code correctness and dev experience, which can be attributed to the fact that people typically look for tools that are able to produce correct code and their effect on developers. The second most common objective was code efficiency. The last two objectives were code efficiency and code usability, with 5 and 3 studies, respectively.

In the fourth analysis, only one study was selected, which was a secondary study. Most of the selected studies were lab studies, which is the primary phase of testing. The second most common type of analysis was simulated analysis, which can be explained by the fact that these types of tools are already mature enough to be used in such environments. The least common type of analysis was field study, with only 4 studies. This can be due to the fact

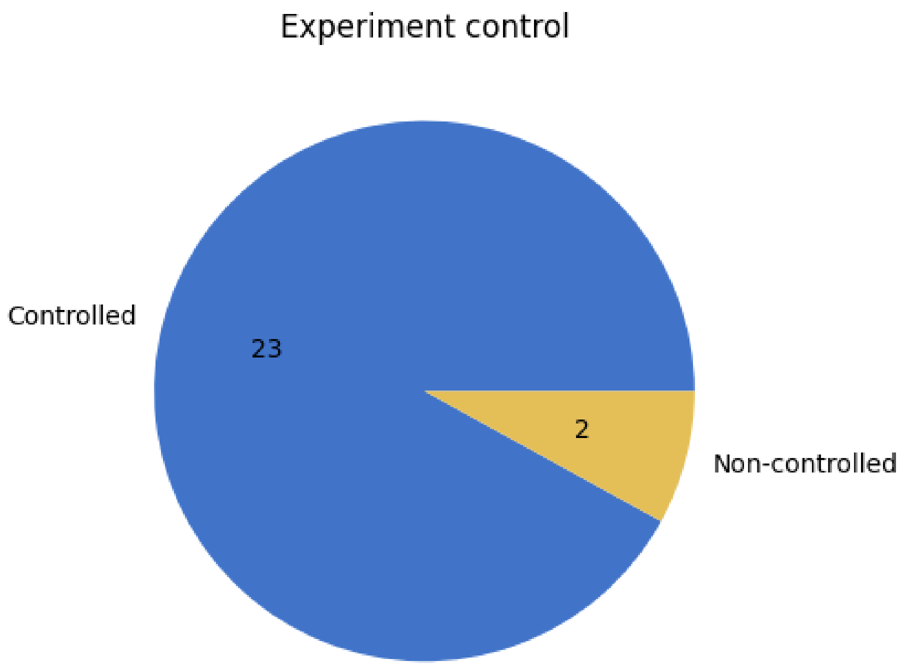

Figure 5.6: Experiment control statistics

that testing software engineering in the field is costly for companies and research institutions.

Then we analysed the experiment control, most of the studies were lab studies and simulated environment studies this can explain the reason why most of the studies were controlled experiment with 23 compared to to more studies done in the field environments where there is less control over the groups. Then the subject types more than half were Human subject types and that's a good indicator because it gives an estimate how the tool behaviour when pairing with humans.

then 6 studies were using oracles that can be explainable to the fact that using an online judge or a code test analyser is one of the simplest and effective ways to test the correctness and efficiency of generated snippets of code. Then the data collection method most of the studies were using observation and this to the fact that this is method is the most applicable one with datasets and oracle which can only be observed for the results. while other method like interview 3 studies and surveys can only be done on humans subject type.

The eighth analysis examined the task required for each study. Developing a task with the tool was examined 8 times, which is a good number because it is the most significant way to test if the tool can be used in pair programming since this is the way the tool will be

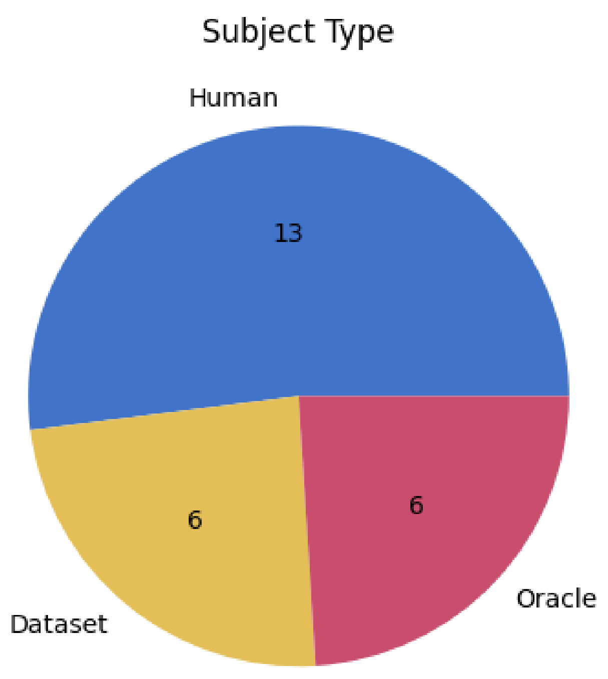

Figure 5.7: Different subject types statistics

used on a daily basis. Solving problems was examined 7 times as well. Suggesting code in different scenarios was examined 10 times, which is understandable as it fits with different scenarios for testing the tool, such as examining datasets.

The final analysis focused on the research methodology, and the most commonly used methodology was quantitative data analysis in 16 studies. This can be attributed to the fact that most of the studies focused on checking correctness, efficiency, and security, which can be tested using quantitative data. The second most commonly used methodology was a mix of qualitative and quantitative data. Only two study relied solely on qualitative data, and one of them was a survey done on different research papers.

5.5 THREATS TO VALIDITY

There are two main threats to validity in this study. The first is related to choosing the dimension values. The problem with choosing these values is that they are dependent on each individual study, so there is no standardized method for choosing them. Additionally, different researchers may have different interpretations of certain terms, such as code effi-

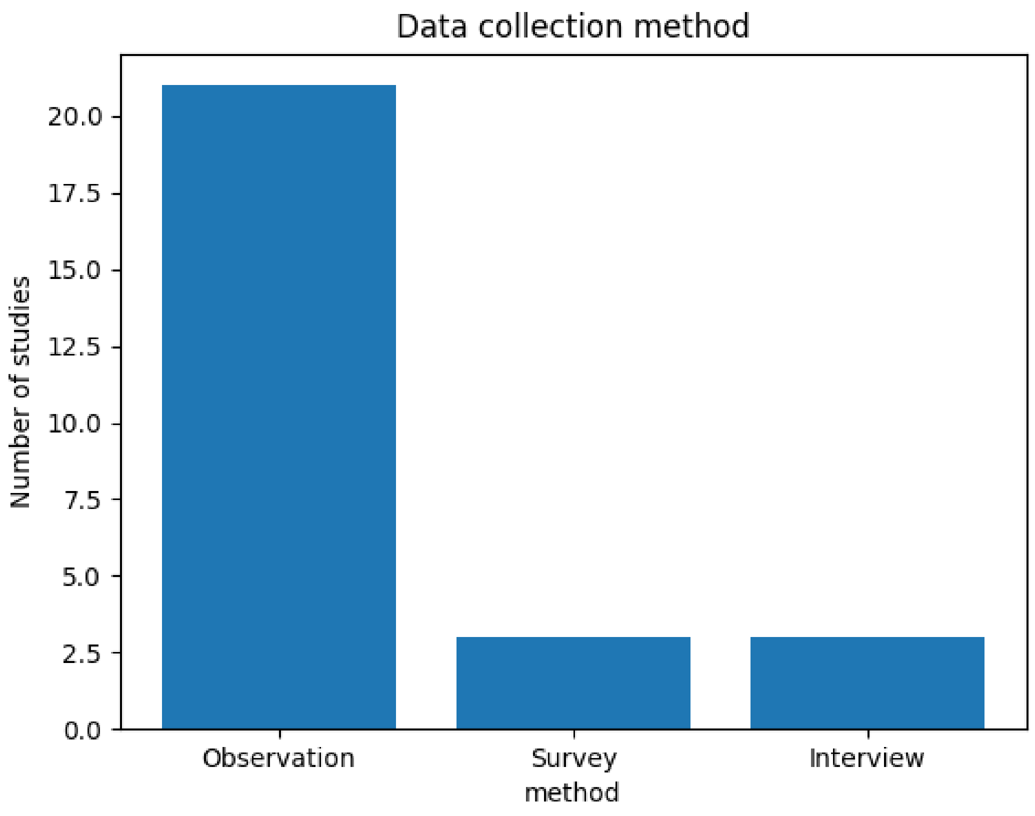

Figure 5.8: Different data collection statistics

ciency versus code correctness. This can lead to inconsistencies in the analysis.

The second threat to validity is related to the use of code quality values in the objective dimension. While we followed the ISO standard, we found that many papers did not check for maintainability, reliability, or productivity. Therefore, we were only able to analyze three values for code quality.

Another potential threat to validity is that we primarily used search engines such as Google Scholar, IEEE Xplore, and ACM Library. There may be relevant papers on other databases that we were unable to find, which could impact the analysis.

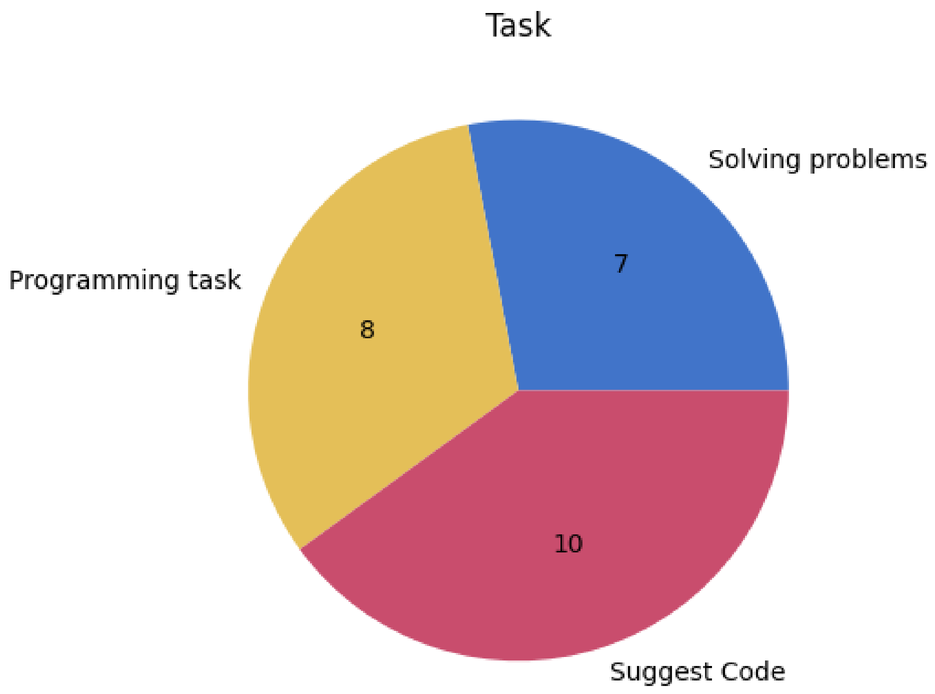

Figure 5.9: Different tasks statistics

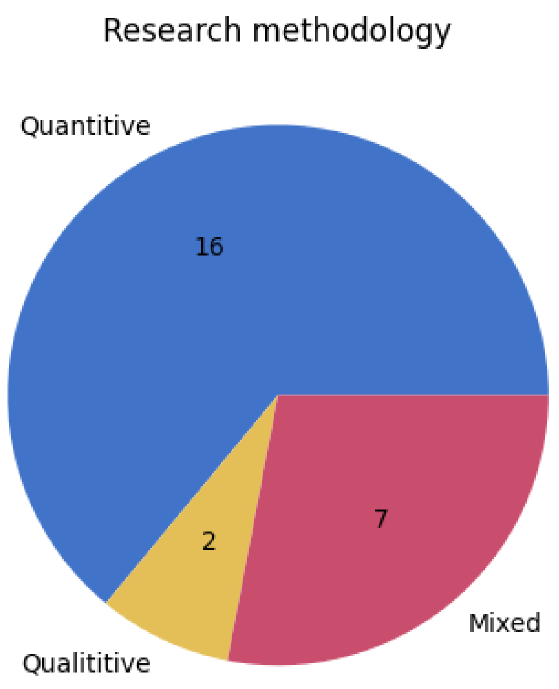

Figure 5.10: Used methodologies statistics

6 RELATED WORK

Yetistiren evaluated [43] and showed the effectiveness GitHub Copilot, Amazon CodeWhisperer, and ChatGPT, with respect to different metrics related to code quality. These metrics include Code Validity, Code Correctness, Code Security, Code Reliability, and Code Maintainability, and the aim is to determine the advantages and limitations of each tool. This study falls within the scope of this study, so it should have been included in the experiment but due to time constraint(it was published 23 of April 2023), we couldn't include it so we gave a summary about it in this section.

They tested the code generation abilities of three tools, GitHub Copilot, Amazon CodeWhisperer, and ChatGPT, using the HumanEval Dataset as a benchmark. The code generated by these tools are assessed using the suggested metrics for measuring the quality of code. They are trying to test the tools for following attributes: valid, correct, secure, reliable and maintainable. They also tried to test the impact of utilizing docstrings on the quality of the generated code and utilizing meaningful function names on the quality of the generated code. The last thing they tried to assess in this study was how the code generation tools evolved over time.

Their metric for code validity is binary with two possible values, valid or not valid. They assessed code correctness by measuring the extent to which the generated code performs as intended, using problem-specific unit tests that come with the HumanEval dataset. They calculated average code correctness by dividing the sum of all code correctness scores by the problem count. They considered the code correctness score of invalid code generations as 0. They calculated average code correctness.

Finally, they used SonarQube to assess code security, code reliability, and code maintainability metrics. For code security, the term vulnerability is defined as a potential risk that compromises the security of the code. SonarQube's Security Module is used to assess code security by calculating the number of vulnerabilities in a given code. To evaluate code maintainability, SonarQube counts the code smells present in the code. For code reliability, the number of bugs in the code is counted also using SonarQube.

They found that all code generation tools can produce valid code 90% of the time, but there is a chance that the generated code may be invalid 10% of the time, with similar types of issues. To improve Code Correctness scores, continuous input from practitioners is necessary. Longer and more complex prompts led to lower scores, while simpler instructions yielded higher scores.

ChatGPT was the most successful tool, while Amazon CodeWhisperer was the least successful. The text reports that practitioners of GitHub Copilot, Amazon CodeWhisperer, and ChatGPT should expect code with some bugs and code smells, but they are not very common. If code smells are present, the average time to solve them is reported as 9.1 minutes for GitHub Copilot, 5.6 minutes for Amazon CodeWhisperer, and 8.9 minutes for ChatGPT. The text also suggests that practitioners should not expect secure code from the generators.

They found that new version of GitHub Copilot had 12% more passed-unit tests than its older version, while the updated version of Amazon CodeWhisperer resulted in 28% more passed-unit tests than its previous version, indicating that both tools have made significant improvements. They suggested that when using code generation tools, it is important to provide clear problem descriptions to obtain valid and correct code. Whenever possible, programmers should include a comprehensive explanation of the problem, along with sample unit tests in the form of docstrings, comments, or other forms of documentation during the solution generation process.

Another study is this one conducted by Sarkar [35]. They did a survey collecting papers to answer several questions. They composed 8 sections and with each section they are trying to collect set of papers to answer each section.

First section was Prior conceptualisations of intelligent assistance for programmers. They tried to answer Do humans only choose features powered by technologies that the AI research community would consider to be AI? Do we include those that use rules and guidelines created by experts? Systems that are able to make decisions that a human may not agree with, or those with a possibility of making mistakes?

Second section, A brief overview of large language models for code generation. This section is divided into two subsections, the transformer architecture and big datasets enable large pre-trained models. They introduced several LLMs starting from Word2Vec in 2013 to BERT in 2019. Second subsection, language models tuned for source code generation, they introduced here several publications, AlphaCode, codex and GPT-3.

Third section is Commercial programming tools that use large language models, They introduced here Github copilot and amazon codeWhisperer. Apart from Copilot, they introduced other commercial applications of AI-based autocompletion features such as Visual Studio Intellicode and Tabnine. They are less comprehensive than Copilot, and the user experience is similar to using the ‘traditional’ auto-complete, which is driven by static analysis, syntax and heuristics.

The fourth section, they examined the implications of AI models that generate code in terms of reliability, safety and security. These models have the potential to bring a variety of serious challenges, and determining the accuracy of the output is difficult due to the complexity of the generated software artifacts.

Existing methods such as HumanEval, MBPP and CodeContests are limited in their ability to evaluate code readability, completeness and potential errors. The next section, Usability and design studies of AI-assisted programming, they collected several studies that compared code generation technologies to human performance.

Next section, they collected several feedback from using code generation tools and they presented in the study summary of the challenges that these programmers found. First one was Writing effective prompts is hard. second issue was The activity of programming shifts towards checking and unfamiliar debugging. Third finding These tools are useful for boilerplate and code reuse.

Section 8 was The inadequacy of existing metaphors for AI-assisted programming, which they were searching the AI assistance in different scenarios. First, AI assistance as search using the tool to search for examples. AI assistance as compilation is the second scenario, is that AI assistance is more akin to a compiler. In this way, programming using natural language prompts and queries is a type of higher-level specification which is "compiled" by the model into a source code in a lower level language. Last scenario is AI assistance as pair programming.

In contrast to search and compilation, which are comparatively impersonal tools, the comparison with pair programming suggests a more customized experience with assistance from a partner who understands the particular context and the desired outcome. AI-assisted programming can be more personalized to the extent that it can take into consideration the specific source code and project files.

Last section is the Issues with application to end-user programming. In this section they cited work that investigated, The advantages and difficulties of programming with LLMs that have been discussed not related to the professional programmer or a learner who is new to programming. These individuals usually don't have a formal background in coding and understand the potential flaws that can be found in AI-generated code. Most people who do programming, however, are not in this category. They are everyday computer users who code to pursue a certain goal and these individuals typically lack the relevant knowledge and motivation to acquire the necessary programming skills.

They concluded that thee use of commercially available tools based on these models has shown that they are a new form of programming. LLM assistance has an effect on almost all aspects of programming, from planning and writing to reuse, editing, comprehension, and debugging. It is similar to a highly advanced compiler, a partner in pair programming, and a streamlined search-and-reuse feature, but also has its own unique aspects which bring both challenges and possibilities for programming research that focuses on the needs of people. Additionally, it poses a big challenge in helping non-experts take advantage of these tools.

Kuttal [18] investigated the difficulties associated with pair programming, such as scheduling conflicts, collocation issues, role imbalance, and power dynamics. The trade-offs of replacing a human with an agent to provide benefits and alleviate the obstacles of pair programming were also examined. Gender-balanced studies with human-human pairs in a remote lab with 18 programmers and Wizard-of-Oz studies with 14 programmers were conducted, and the results were analyzed both qualitatively and quantitatively.

The aim of this research was to assess the possibility of a conversational agent as a partner in a pair programming environment. Questions posed in this research included: Can the advantages of pair programming be maintained by replacing a human programmer with an agent? What sort of knowledge is exchanged between human-human and human-agent pairs? Do human programmers consider the agent as their partner?

This study compared the effectiveness of a human-human pair and a human-agent pair in a Wizard of Oz study. The wizards simulated the components of a conversational agent (such as speech recognition, intent understanding, dialogue state tracking, dialogue policy, response generation) using a constrained Wizard of Oz protocol.

The second study design was similar to the human-human study, except that a participant completed the task with the agent and were given an instructional tutorial on the agent itself, pair jelling was not incorporated, and the agent's gender presentation was switched halfway through each study. After the task was completed, the participants were asked interview questions to get their opinions on the agent.

To evaluate the trade-offs of utilizing an agent in a pair programming environment, they compared and contrasted the transcripts, productivity, code quality, and questionnaire results from the human-human and human-agent studies, and the findings were verified with interviews. They examined each research question and its associated findings, with each finding being accompanied by relevant key takeaways.

They found that the comparison of the two studies showed no substantial differences in productivity, code quality, and self-efficacy. Agents were found to facilitate knowledge transfer, but unlike humans, they were unable to provide logical explanations or engage in discussions. Human partners showed trust and humility towards agents. This research demonstrates that agents can act as effective pair programming partners and opens up possibilities for new research on conversational agents for programming.

Robe [33] tested if the implementation of an interactive pair-programming conversational agent has encountered three difficulties: lack of benchmark datasets, absence of software engineering specific labels, and difficulty understanding developer conversations. To address these, a Wizard of Oz study involving 14 participants was conducted, and 4,443 developer-agent utterances were collected. An open coding process yielded 26 software engineering labels and a hierarchical classification scheme.

Three state-of-the-art transformer-based language models (BERT, GPT-2, and XLNet) were utilized to comprehend the labeled developer-agent conversations, and results were found to be interchangeable. Furthermore, developer-developer conversations on video hosting websites were investigated and a publicly available dataset (3,436 utterances) was labeled. Transfer-learning increased accuracy, however, developer-developer conversations were discovered to be more implicit, neutral, and opinionated than developer-agent conversations.

This paper offers developer-agent conversational data, software engineering specific labels, and examines the utility of transformer-based language models and transfer learning for pair programming conversations. It includes four main contributions: presenting a hierarchical label scheme, manually labeling transcripts, gathering and classifying questions, and demonstrating the usefulness of transfer learning.

The findings have implications for programming virtual Q/A assistants, intelligent tutoring systems, and bots. It emphasizes possibilities for researchers and practitioners to build a pair-programming benchmark dataset, continue to investigate methods that will enhance the generalizability of training data, and finalize the remaining components of a pair-programming agent.

Almonte [1] created an overview of the systematic mapping review of recommender systems (RSs) for modeling and model-driven engineering (MDE) tasks in this work. Four categories domain, tooling, suggestion, and evaluation—were examined for classifying the 66 relevant studies. The majority of RSs were discovered to be knowledge-based, linguistically unrestricted, and employed for the completion and maintenance of artifacts.

There aren't many RSs for model transformations or code generators, and offline experiments are still the most used evaluation method. In addition to outlining research prospects, the study predicts that RSs will become more significant in software engineering. Understanding the tasks that can be recommended, the suitable recommendation techniques and their evaluation procedures, as well as the open difficulties in the industry, is also helpful for tool developers and researchers.

In this research, recommender systems (RSs) for modeling and model-driven engineering (MDE) activities are reviewed systematically. A total of 151 papers were chosen after four reviewers looked over the abstracts of 1,456 different papers. These papers all made recommendations for modeling jobs. Based on standards for language, focus, and quality, the documents were chosen.

In the second phase, 89 papers were considered irrelevant, 9 papers were unavailable, and 53 papers were thought to be relevant. A final collection of 66 relevant papers was produced through a snowballing process in which 13 additional papers were chosen.

These 66 documents, which span nearly 16 years of research and 51 distinct methodologies, range from 2004 to September 2020. There is a rise in the quantity of papers on this subject, indicating a rise in interest in the subject.

The paper can help tool developers and researchers understand the tasks that can be recommended for, the recommendation techniques that can be used and how to evaluate them, and the unresolved issues in the field. The findings help to clarify the opportunities and difficulties that RSs for modeling and MDE tasks present as well as the opportunity for the creation of novel tools.

7 CONCLUDING REMARKS

7.1 CONCLUSION

In this section we will use the results from the evaluation chapter and the discussion to answer the research questions.

RQ1 What is the current state of the automated code generation in the current research?

RQ1.1 How did the number of publications from the automated code generation change over time?

> The publications in automated code generation for pair programming varied, starting from just one study in 2021, increasing to 15 studies in 2022, and then to 9 publications during the initial three months of 2023.

RQ1.2 What are the most used research methodologies in automated code generation?

> There are three types of methodologies used in this field Quantitative, Qualitative and mixed. The majority of these studies are done quantitatively because the nature of testing such tools requires datasets, correct ratio or how much memory did it save etc. While still some studies used also qualitative or mixed results specially when it came to developer experience because it can be represented with qualitative data as well as quantitative like explaining difference in productivity.

RQ1.3 What are the research objectives in automated code generation?

> The majority of the objectives used in automated code generation for pair programming were code correctness, how correct is the code produced by automated code generation tools. Code efficiency, how much is the code generated more efficient in terms of lines of code, better structure and space consumption, Code security, how secure was the code generated by the tool, Code usability, how usable and readable the code generated by the tools. Developer experience, how productive or how was his experience while using such a tool, was also an objective in this field of research .

RQ1.4 What is researched task in automated code generation?

The tasks in this field, varied from solving problems to develop a task using the tool or make the tool solve a scenario, to suggest code for different tasks.

RQ1.5 What are the used technologies in automated code generation?

The most used technologies in automated code generation, first most used one is GitHub copilot the state of the art of this field, CodeWhisperer was released second half of 2022 but it is expected to have place in this market. AlphaCode is used more for solving contest problems.

Then we are trying to answer the second main question"

RQ2 What are possible research gaps in the field of automated code generation?

There is multiple gaps in the field at the moment. First, is the technologies tested, the most used technology is Github copilot, even the other technologies is Codex and Gpt which all of them belongs to the same language model. Codex is the underline model for copilot and Codex is a modified GPT to produce code. So most of the studies are biased towards this language model. It is expected in the future with the release of CodeWhisperer that it encourage more research for other tools than the Openai tools but at the moment this remains a gap.

The second gap is the experiment design so majority of the designs were Lab studies designs or in a simulated environments designs. These environments can give you a good understanding of how the tool can behave in everyday problems. Another stage of testing still needed in this field which is closer to production and working with developers' everyday problems.

Another gap is that they were no research that used iso standards when reviewing code quality. They were mostly using tools to assess the code quality aspects like usability. but there is no research that tackling the 6 aspects of code quality presented in this standard.

7.2 FUTURE WORK

There is two types of future works in this study. First future work within the study, like repeating the search process at the end of this year 2023, it is expected to have more papers that fits with the research scope. Also repeating the study when more research that covers CodeWhiperer comes to light. Repeating the search process when new tool emerge in this area. Also copilot would have been released for 2 years now so it is expected to have more field studies testing the tool in the near future.

Second future work is related to the field code generation for pair programming in general. Testing other tools than the ones developed by Openai or has language models developed by Openai in its core so we can have a better vision how the technology of code generation can be used for pair programming. Second is following ISO standards when testing for code quality. Another future work field related is doing more field studies although it is more costly but it gives a better indication about the behaviour of tools in production. Finally is doing more interviews and surveys so can get more developers first hand experience about using the tools.

A
APPENDIX

This is the Appendix. Add further sections for your appendices here.

BIBLIOGRAPHY

[1] Lissette Almonte, Esther Guerra, Iván Cantador, and Juan Lara. "Recommender systems in model-driven engineering: A systematic mapping review." In: *Software and Systems Modeling* 21 (Feb. 2022). DOI: 10.1007/s10270-021-00905-x.

[2] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. *Structural Language Models for Any-Code Generation*. 2020. URL: https://openreview.net/forum?id=HylZIT4Yvr.

[3] Owura Asare, Meiyappan Nagappan, and N. Asokan. *Is GitHub's Copilot as Bad as Humans at Introducing Vulnerabilities in Code?* 2023. arXiv: 2204.04741 [cs.SE].

[4] Olavo Barbosa12 and Carina Alves. "A systematic mapping study on software ecosystems." In: (2011).

[5] Shraddha Barke, Michael B. James, and Nadia Polikarpova. *Grounded Copilot: How Programmers Interact with Code-Generating Models*. 2022. arXiv: 2206.15000 [cs.HC].

[6] Andrew Begel and Nachiappan Nagappan. "Pair Programming: What's in It for Me?" In: *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ESEM '08. Kaiserslautern, Germany: Association for Computing Machinery, 2008, 120–128. ISBN: 9781595939715. DOI: 10.1145/1414004.1414026. URL: https://doi.org/10.1145/1414004.1414026.

[7] Christian Bird, Denae Ford, Thomas Zimmermann, Nicole Forsgren, Eirini Kalliamvakou, Travis Lowdermilk, and Idan Gazit. "Taking Flight with Copilot: Early Insights and Opportunities of AI-Powered Pair-Programming Tools." In: *Queue* 20.6 (2023), 35–57. ISSN: 1542-7730. DOI: 10.1145/3582083. URL: https://doi.org/10.1145/3582083.

[8] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. DOI: 10.48550/ARXIV.2107.03374. URL: https://arxiv.org/abs/2107.03374.

[9] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, Zhen Ming, and Jiang. *GitHub Copilot AI pair programmer: Asset or Liability?* 2023. arXiv: 2206.15331 [cs.SE].

[10] Enrique Dehaerne, Bappaditya Dey, Sandip Halder, Stefan De Gendt, and Wannes Meert. "Code Generation Using Machine Learning: A Systematic Review." In: *IEEE Access* 10 (2022), pp. 82434–82455. DOI: 10.1109/ACCESS.2022.3196347.

[11] Paul Denny, Viraj Kumar, and Nasser Giacaman. *Conversing with Copilot: Exploring Prompt Engineering for Solving CS1 Problems Using Natural Language*. 2022. arXiv: 2210.15157 [cs.HC].

[12] Hossein Hajipour, Thorsten Holz, Lea Schönherr, and Mario Fritz. *Systematically Finding Security Vulnerabilities in Black-Box Code Generation Models*. 2023. arXiv: 2302.04012 [cs.CR].

[13] Vincent J. Hellendoorn, Premkumar T. Devanbu, and Alberto Bacchelli. “Will They Like This? Evaluating Code Contributions with Language Models.” In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. 2015, pp. 157–167. DOI: 10.1109/MSR.2015.22.

[14] Saki Imai. “Is GitHub Copilot a Substitute for Human Pair-Programming? An Empirical Study.” In: *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. ICSE ’22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, 319–321. ISBN: 9781450392235. DOI: 10.1145/3510454.3522684. URL: https://doi.org/10.1145/3510454.3522684.

[15] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. *Jigsaw: Large Language Models meet Program Synthesis*. 2021. arXiv: 2112.02969 [cs.SE].

[16] Robert Feldt Kai Petersen and Shahid Mujtaba et al. “Systematic Mapping Studies in Software Engineering.” In: (), pp. 1–10. DOI: 10.14236/ewic/EASE2008.8.

[17] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. CMU/SEI-90-TR-21. Software Engineering Institute, 1990.

[18] Sandeep Kaur Kuttal, Bali Ong, Kate Kwasny, and Peter Robe. “Trade-Offs for Substituting a Human with an Agent in a Pair Programming Context: The Good, the Bad, and the Ugly.” In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. CHI ’21. Yokohama, Japan: Association for Computing Machinery, 2021. ISBN: 9781450380966. DOI: 10.1145/3411764.3445659. URL: https://doi.org/10.1145/3411764.3445659.

[19] Sila Lertbanjongngam, Bodin Chinthanet, Takashi Ishio, Raula Gaikovina Kula, Pattara Leelaprute, Bundit Manaskasemsak, Arnon Rungsawang, and Kenichi Matsumoto. *An Empirical Evaluation of Competitive Programming AI: A Case Study of AlphaCode*. 2022. arXiv: 2208.08603 [cs.SE].

[20] Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Dong Chen, Shuai Wang, and Cuiyun Gao. *CCTEST: Testing and Repairing Code Completion Systems*. 2022. arXiv: 2208.08289 [cs.SE].

[21] Naser Al Madi. “How Readable is Model-generated Code? Examining Readability and Visual Inspection of GitHub Copilot.” In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2022. DOI: 10.1145/3551349.3560438. URL: https://doi.org/10.1145%2F3551349.3560438.

[22] Ehsan Mashhadi and Hadi Hemmati. “Applying CodeBERT for Automated Program Repair of Java Simple Bugs.” In: *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 2021, pp. 505–509. DOI: 10.1109/MSR52588.2021.00063.

[23] Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. *On the Robustness of Code Generation Techniques: An Empirical Study on GitHub Copilot*. 2023. arXiv: 2302.00438 [cs.SE].

[24] Gábor Melis, Chris Dyer, and Phil Blunsom. *On the State of the Art of Evaluation in Neural Language Models*. 2017. DOI: 10.48550/ARXIV.1707.05589. URL: https://arxiv.org/abs/1707.05589.

[25] Nhan Nguyen and Sarah Nadi. “An Empirical Evaluation of GitHub Copilot’s Code Suggestions.” In: *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. 2022, pp. 1–5. DOI: 10.1145/3524842.3528470.

[26] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. *Asleep at the Keyboard? Assessing the Security of GitHub Copilot’s Code Contributions*. 2021. arXiv: 2108.09293 [cs.CR].

[27] Oscar Pedreira, Félix García, Nieves Brisaboa, and Mario Piattini. “Gamification in software engineering – A systematic mapping.” In: *Information and Software Technology* 57 (2015), pp. 157–168. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2014.08.007. URL: https://www.sciencedirect.com/science/article/pii/S0950584914001980.

[28] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. *The Impact of AI on Developer Productivity: Evidence from GitHub Copilot*. 2023. arXiv: 2302.06590 [cs.SE].

[29] Luis Perez, Lizi Ottens, and Sudharshan Viswanathan. *Automatic Code Generation using Pre-Trained Language Models*. 2021. DOI: 10.48550/ARXIV.2102.10535. URL: https://arxiv.org/abs/2102.10535.

[30] Julian Aron Prenner and Romain Robbes. *Automatic Program Repair with OpenAI’s Codex: Evaluating QuixBugs*. 2021. arXiv: 2111.03922 [cs.SE].

[31] Rohith Pudari and Neil A. Ernst. *From Copilot to Pilot: Towards AI Supported Software Development*. 2023. arXiv: 2303.04142 [cs.SE].

[32] Ben Puryear and Gina Sprint. “Github Copilot in the Classroom: Learning to Code with AI Assistance.” In: *J. Comput. Sci. Coll.* 38.1 (2022), 37–47. ISSN: 1937-4771.

[33] Peter Robe, Sandeep K. Kuttal, Jake AuBuchon, and Jacob Hart. “Pair Programming Conversations with Agents vs. Developers: Challenges and Opportunities for SE Community.” In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022. Singapore, Singapore: Association for Computing Machinery, 2022, 319–331. ISBN: 9781450394130. DOI: 10.1145/3540250.3549127. URL: https://doi.org/10.1145/3540250.3549127.

[34] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Siddharth Garg, and Brendan Dolan-Gavitt. *Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants*. 2023. arXiv: 2208.09727 [cs.CR].

[35] Advait Sarkar, Andrew D. Gordon, Carina Negreanu, Christian Poelitz, Sruti Srinivasa Ragavan, and Ben Zorn. *What is it like to program with artificial intelligence?* 2022. arXiv: 2208.06213 [cs.HC].

[36] Mohammed Latif Siddiq, Shafayat H. Majumder, Maisha R. Mim, Sourov Jajodia, and Joanna C. S. Santos. “An Empirical Study of Code Smells in Transformer-based Code Generation Techniques.” In: *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2022, pp. 71–82. DOI: 10.1109/SCAM55253.2022.00014.

[37] Mohammed Latif Siddiq and Joanna C. S. Santos. "SecurityEval Dataset: Mining Vulnerability Examples to Evaluate Machine Learning-Based Code Generation Techniques." In: *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security*. MSR4Pamp;S 2022. Singapore, Singapore: Association for Computing Machinery, 2022, 29–33. ISBN: 9781450394574. DOI: 10.1145/3549035.3561184. URL: https://doi.org/10.1145/3549035.3561184.

[38] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. "Expectation vs.nbsp;Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models." In: *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*. CHI EA '22. New Orleans, LA, USA: Association for Computing Machinery, 2022. ISBN: 9781450391566. DOI: 10.1145/3491101.3519665. URL: https://doi.org/10.1145/3491101.3519665.

[39] Michel Wermelinger. "Using GitHub Copilot to Solve Simple Programming Problems." In: SIGCSE 2023. Toronto ON, Canada: Association for Computing Machinery, 2023, 172–178. ISBN: 9781450394314. DOI: 10.1145/3545945.3569830. URL: https://doi.org/10.1145/3545945.3569830.

[40] L. Williams, R.R. Kessler, W. Cunningham, and R. Jeffries. "Strengthening the case for pair programming." In: *IEEE Software* 17.4 (2000), pp. 19–25. DOI: 10.1109/52.854064.

[41] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. "A Systematic Evaluation of Large Language Models of Code." In: *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. MAPS 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, 1–10. ISBN: 9781450392730. DOI: 10.1145/3520312.3534862. URL: https://doi.org/10.1145/3520312.3534862.

[42] Burak Yetistiren, Isik Ozsoy, and Eray Tuzun. "Assessing the Quality of GitHub Copilot's Code Generation." In: *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*. PROMISE 2022. Singapore, Singapore: Association for Computing Machinery, 2022, 62–71. ISBN: 9781450398602. DOI: 10.1145/3558489.3559072. URL: https://doi.org/10.1145/3558489.3559072.

[43] Burak Yetiştiren, Işık Özsoy, Miray Ayerdem, and Eray Tüzün. *Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT*. 2023. arXiv: 2304.10778 [cs.SE].

[44] Albert Ziegler, Eirini Kalliamvakou, Shawn Simister, Ganesh Sittampalam, Alice Li, Andrew Rice, Devon Rifkin, and Edward Aftandilian. *Productivity Assessment of Neural Code Completion*. 2022. arXiv: 2205.06537 [cs.SE].