

AI Supported Software Development: Moving Beyond Code Completion

by

Rohith Pudari

B.Tech., Jawaharlal Nehru Technological University Hyderabad, 2019

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

© Rohith Pudari, 2022
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

AI Supported Software Development: Moving Beyond Code Completion

by

Rohith Pudari

B.Tech., Jawaharlal Nehru Technological University Hyderabad, 2019

Supervisory Committee

Dr. Neil A. Ernst, Supervisor
(Department of Computer Science)

Dr. Jens H. Weber, Departmental Member
(Department of Computer Science)

ABSTRACT

AI-supported programming has arrived, as shown by the introduction and successes of large language models for code, such as Copilot/Codex (Github/OpenAI) and AlphaCode (DeepMind). Above-average human performance on programming challenges is now possible. However, software development is much more than solving programming contests. Moving beyond code completion to AI-supported software development will require an AI system that can, among other things, understand how to avoid code smells, follow language idioms, and eventually (maybe!) propose rational software designs.

In this study, we explore the current limitations of Copilot and offer a simple taxonomy for understanding the classification of AI-supported code completion tools in this space. We first perform an exploratory study on Copilot’s code suggestions for language idioms and code smells. Copilot does not follow language idioms and avoid code smells in most of our test scenarios. We then conduct additional investigation to determine the current boundaries of Copilot by introducing a taxonomy of software abstraction hierarchies where ‘basic programming functionality’ such as code compilation and syntax checking is at the least abstract level, software architecture analysis and design are at the most abstract level. We conclude by providing a discussion on challenges for future development of AI-supported code completion tools to reach the design level of abstraction in our taxonomy.

Table of Contents

Supervisory Committee	ii
Abstract	iii
Table of Contents	iv
List of Tables	vii
List of Figures	viii
Acknowledgements	ix
1 Introduction	1
1.1 Motivation	2
1.2 AI-supported code completion tools	3
1.3 Problem Statement and Research Questions	4
1.4 Research Design and Methodology	5
1.5 Contributions	6
1.6 Thesis Outline	7
2 Background & Related Work	8
2.1 Introduction	8
2.2 Evolution of AI-supported code completion tools	8
2.2.1 Statistical Language Models	8
2.2.1.1 N-grams	9
2.2.1.2 SLANG	9
2.2.1.3 CACHECA	9
2.2.1.4 Problems with N-grams	9
2.2.2 Neural Language Models (NLMs)	10
2.2.2.1 Transformer Models	10

2.2.2.2	Contextualized Word Representations	10
2.3	Code completion with NLMs	11
2.3.1	GitHub Copilot	12
2.3.2	Alternatives to Copilot	13
2.4	Chapter Summary	14
3	Challenges with Copilot	16
3.1	Introduction	16
3.2	Background & Motivation	17
3.3	Methodology	18
3.3.1	Pythonic Idioms	19
3.3.1.1	Sampling Approach	20
3.3.2	Code Smells	20
3.3.2.1	Sampling Approach	21
3.3.3	Input to Copilot	21
3.3.4	Evaluation of Copilot code suggestions	22
3.4	Results	22
3.4.1	Pythonic Idioms	23
3.4.2	Code Smells	27
3.4.3	Summary of Findings	32
3.5	Chapter Summary	32
4	Framework	35
4.1	Introduction	35
4.1.1	Motivation	36
4.2	Taxonomy	37
4.2.1	Syntax	39
4.2.2	Correctness	40
4.2.3	Paradigms and Idioms	41
4.2.4	Code Smells	42
4.2.5	Design level	44
4.2.5.1	Module level design	44
4.2.5.2	System level design	45
4.3	AI-supported Software Development	46
4.3.1	Evolution of design over time	48

4.4	Chapter Summary	50
5	Discussion, Future Work and Conclusion	52
5.1	Introduction	52
5.2	Discussion	53
5.2.1	Copilot code suggestions	53
5.2.2	Copilot code recitation	54
5.3	Implications	55
5.3.1	Implications for practitioners	55
5.3.1.1	Pre-training the LLM	55
5.3.1.2	Code completion time	56
5.3.1.3	Over-reliance	56
5.3.1.4	Ethical Considerations	56
5.3.2	Implications for researchers	57
5.3.2.1	Moving Beyond Tokens	57
5.3.2.2	Explainability	58
5.3.2.3	Control	58
5.4	Threats to Validity	59
5.4.1	Internal Validity	59
5.4.2	Construct Validity	59
5.4.3	Bias, Fairness, and Representation	60
5.5	Conclusion	60
	Bibliography	62

List of Tables

Table 3.1	List of all Pythonic idioms tested on Copilot.	28
Table 3.2	List of all JavaScript best practices tested on Copilot.	33

List of Figures

Figure 2.1 Transformer Architecture [74]	11
Figure 3.1 List comprehension Pythonic idiom and Copilot top suggestion.	24
Figure 3.2 Find index of every word in input string Pythonic idiom and Copilot top suggestion.	25
Figure 3.3 Open and write to a file Pythonic idiom and Copilot top suggestion.	26
Figure 3.4 Best practice for copying array contents and Copilot top suggestion.	29
Figure 3.5 Best practice for creating two references and Copilot top sugges- tion.	30
Figure 3.6 Best practice for returning string and variable name and Copilot top suggestion.	31
Figure 4.1 Koopman’s Autonomous Vehicle Safety Hierarchy of Needs [43]. SOTIF = safety of the intended function.	36
Figure 4.2 Hierarchy of software abstractions. Copilot cleared all green lev- els and struggled in red levels.	38
Figure 4.3 Code suggestion of AI-supported code completion tools at syntax level.	39
Figure 4.4 Code suggestion of AI-supported code completion tools at cor- rectness level.	40
Figure 4.5 Code suggestion of AI-supported code completion tools at paradigms and idioms level.	42
Figure 4.6 Code suggestion of AI-supported code completion tools at code smells level.	43

ACKNOWLEDGEMENTS

I would not have finished this thesis without my professors, family, friends, and colleagues' help and support.

First and foremost, I would like to thank my advisor, Dr. Neil Ernst, who is always supportive and patient, provides guidance, and challenges me. I could have never been here without his countless and significant support. I also thank him for the resources and funding provided to support my work.

I would like to thank GitHub for providing me access to Copilot. This research would not have been possible without it.

Moreover, I thank my fellow colleagues at Octera group for making me feel home and motivated at workplace and for all the activities and fun we had together.

All of this wouldn't have been possible without the equal efforts from the administrative staff of the University of Victoria and the Department of Computer Science. I thank them for all the administrative services they provided me during my M.Sc.

I would like to thank my wonderful and perfect parents: Renuka and Srinivas, who give me endless and selfless love and always cheer me on, just as they have every step of the way. They gave me the ability to maintain hope for a brighter morning, even during our darkest nights.

Also, I take this opportunity to thank my friends outside the research lab: Francis, Susan, Kealey, Anthony, and Sean. Without their support, my life in Victoria would be boring and lonely. Thank you for sticking by me.

Thank you all.

Chapter 1

Introduction

Programming is a powerful and ubiquitous problem-solving tool. Developing systems that can assist software developers or even generate programs independently could make programming more productive and accessible [70]. With increasing pressure on software developers to produce code quickly, there is considerable interest in tools and techniques for improving productivity [52]. Code completion is one such feature that predicts what a software developer tries to code and offers predictions as suggestions to the user [76]. All modern IDEs feature intelligent code completion tools in different forms that are used by both new and experienced software developers [51]. Developing AI systems that can effectively model and understand code can transform these code completion tools and how we interact with them [51].

Recent large-scale pre-trained language models such as Codex [12] have demonstrated an impressive ability to generate code and can now solve programming contest-style problems [54]. However, software development is much more than writing code. It involves complex challenges like following the best practices, avoiding code smells, using design patterns, and many more decisions before writing code.

The scope of capabilities for AI-supported code completion tools is uncertain. Identifying the nature of Copilot capabilities when it comes to more complex challenges, i.e., AI-supported software development (as opposed to programming tasks, such as coding or solving competitive programming problems). Delineating where AI-supported code completion tools are currently best able to perform, and where more complex software development tasks overwhelm them helps answer questions like exactly which software problems can current AI-supported code completion tools solve? If AI-supported code completion tools make a suggestion, is that suggestion accurate and optimal? Should a user intervene to correct it? However, identifying

these boundaries is a challenging task. In the next section, we discuss this challenge and the research opportunity it creates as this study’s motivation.

1.1 Motivation

In recent years, there have been considerable improvements in the field of AI-supported code completion tools. Copilot [27], an in-IDE recommender system that leverages OpenAI’s Codex neural language model (NLM) [12] which uses a GPT-3 model [10] has been at the forefront and is particularly impressive in understanding the context and semantics of code with just a few lines of comments or code as input and can suggest the next few lines or even entire functions in some cases [12].

The biggest challenge with using tools like Copilot is their training data. These tools are trained on existing software source code, and training costs are expensive. Several classes of errors have been discovered (shown in section 3.2), which follow from the presence of these same errors in public (training) data. There is a need to stay cautious about common bugs people make, creeping into Copilot suggestions and keeping Copilot updated to the ever-changing best practices or new bug fixes in public repositories.

Understanding the current limitations of Copilot will help developers use AI-supported code completion tools effectively. The advantage of knowing where Copilot is good/bad lets users use these tools more efficiently by letting AI-supported code completion tools take over where they excel and focus more on tasks where AI-supported code completion tools are shown to have struggled with the quality of code suggestions.

A taxonomy of software abstractions would help create a formal structure to better understand the capabilities and limitations of AI-supported code completion tools like Copilot. A taxonomy helps describe the hierarchical stages of software abstractions at which AI-supported code completion tools like Copilot are operating. For example, Copilot is capable of suggesting syntactical code but cannot suggest architectural tactics. Creating a taxonomy of software abstractions can be useful in developing new AI-supported code completion tools and measuring their capabilities.

A taxonomy of software abstractions can help in making code suggestions better in complex situations, shifting research focus to make AI-supported code completion tools better at tasks shown to be challenging, minimizing the input required by the user to make AI-supported code completion tools create meaningful, quality

suggestions.

In this thesis, we investigate the current limitations of Copilot with an empirical study on Copilot suggestions. Using the findings, we introduce a taxonomy of software abstraction hierarchy, modeled on the ideas in the SAE taxonomy and Koopman’s extension for self-driving vehicles. The insights gained from this study can help developers understand how to best use tools like Copilot and provide insights to researchers trying to develop and improve the field of AI-supported code completion tools.

1.2 AI-supported code completion tools

The purpose of code completion as an IDE feature is to save the user’s time and effort by suggesting code before manually inputting it. Code completion can also aid in project exploration by giving new users a chance to view entities from various areas of the code base. The IDE typically offers a list of potential completions for the user because the process of developing code can vary greatly.

In-IDE code completion tools have improved a lot in recent years. Early code completion techniques include suggesting variables or method calls from user code bases [46], based on alphabetically sorted lists that show the next token to write given the registered characters. Followed by tools capable of suggesting entire code blocks [13] utilizing statistical language models, such as n-gram models, is the origin of the use of data-driven strategies for code recommendation.

Karampatsis et al. [35] suggested that the most effective code-completion approach is based on neural network models. The authors used byte pair encoding [67] as a method of overcoming the lack of vocabulary issue, demonstrating that their best model performs better than n-gram models and is easily transferable to other domains resulting in a deep learning resurgence that has led to strong advances in the field of AI-supported code completion tools.

Pre-trained large language models such as GPT-3 [10], CodeBERT [22], Codex [12], AlphaCode [45] have dramatically improved the state-of-the-art on a code completion. For example, even when the large language model’s only input is a plain language description of the developer’s desired implementation, these models are now capable of predicting the entire method with reasonable accuracy [12].

1.3 Problem Statement and Research Questions

The overarching goal of this study is to:

Identify the boundaries of the capabilities of GitHub Copilot. We compare Copilot code suggestions against well-known Pythonic idioms and JavaScript code smells toward this goal. Additionally, we introduce a simple taxonomy of software abstraction hierarchies to show different capability levels of AI-supported code completion tools.

The capabilities and the limitations of current AI-supported code completion tools like Copilot are unknown. Identifying the boundaries of AI-supported code completion tools would help the users use the tool effectively and shift the research focus more on the tasks AI-supported code completion tools are proven to be not helpful.

This study aims to understand the areas where Copilot performs better than a human and where Copilot performs worse than a human. We conduct an exploratory study with the following research objectives:

RQ-1: What are the current boundaries of AI-supported code completion tools?

Approach - We use GitHub’s Copilot as a representative for current AI-supported code completion tools. We explore Copilot’s code suggestions for code smells and usage of Pythonic idioms. We conduct additional investigation to determine the current boundaries of Copilot by introducing a taxonomy of software abstraction hierarchies where ‘basic programming functionality’ such as code compilation and syntax checking is at the least abstract level. Software architecture analysis and design are at the most abstract level.

RQ-1.1: How do AI-supported code completion tools manage programming idioms?

Approach - We investigate Copilot code suggestions on the top 25 Pythonic idioms used in open source projects. These Pythonic idioms are sampled from the work of Alexandru et al. [3] and Farook et al. [21], which identified Pythonic idioms from presentations given by renowned Python developers. We investigate how Copilot’s top code suggestion compares to Python idioms from Alexandru et al. [3] and Farook et al. [21]. In addition, we report if the Pythonic idiom is listed in any of the ten viewable suggestions from Copilot.

RQ-1.2: How do AI-supported code completion tools manage to suggest non-smelly code?

Approach - We investigate Copilot code suggestions on 25 different best practices in JavaScript. We sampled best practices from the AirBNB JavaScript coding style guide [1]. We explore how Copilot’s top code suggestion compares to the best practices recommended in the AirBNB JavaScript coding style guide [1]. Additionally, we report if the best practice is listed in any of the ten viewable suggestions from Copilot.

RQ-2: Given the current boundary, how far is it from suggesting design decisions which seem much beyond the boundary?

Approach - Based on our findings in RQ-1, we discuss how far current AI-supported code completion tools are from the design level in our taxonomy. We look at the current limitations of Copilot and provide recommendations on how to make current AI-supported code completion tools reach the design abstraction level. Additionally, we report on ethical considerations, explainability, and control of AI-supported code completion tools like Copilot.

1.4 Research Design and Methodology

The current AI-supported code completion tools approaches focus on programming-in-the-small [16] i.e, on individual lines of code. Code language models have focused (effectively!) on source code as natural language [31]. This models the software development task as predicting the next token or series of tokens.

Introduced in June 2021, GitHub’s Copilot [27] is an in-IDE recommender system that leverages OpenAI’s Codex neural language model (NLM) [12] which uses a GPT-3 model [10] and is then fine-tuned on code from GitHub to generate code suggestions that are uncannily effective and can perform above human average on programming contest problems [54]. As Copilot’s webpage says, Copilot aims to produce “safe and effective code [with] suggestions for whole lines or entire functions” as “your AI pair programmer” [27]. Thus, support from language models is currently focused on software *programming*, rather than software *development* (in-the-large). Copilot can generate code in various languages given some context, such as comments, function names, and surrounding code. Copilot is the largest and most capable model currently available. We perform all our experiments on GitHub Copilot as a representative for

AI-supported code completion tools.

We begin by sampling the top 25 language idioms used in open source projects from Alexandru et al. [3] and Farook et al. [21]. We then compare and contrast Copilot’s code suggestion for each idiom and report if Copilot suggested the idiom listed in Alexandru et al. [3] and Farook et al. [21]. We also sampled 25 coding scenarios for detecting code smells from AirBNB JavaScript coding style guide [1], a widely used coding style and code review standard. We then compare and contrast Copilot’s code suggestion for each scenario and report if Copilot suggested the best practice listed in the AirBNB JavaScript coding style guide. We base our taxonomy of software abstraction hierarchies on the findings from Copilot suggestions on language idioms and code smells.

We then report on the current limitations of AI-supported code completion tools using Copilot as a representative tool and introduce a taxonomy of software abstraction hierarchies (see figure 4.2), inspired by an analogous concept in the more developed (but still nascent) area of autonomous driving. We also present an example coding scenario for every level of abstraction in our taxonomy, showing the requirements to be fulfilled by AI-supported code completion tools to satisfy that level of abstraction.

We conclude by providing a discussion for future development of AI-supported code completion tools to reach the design level of abstraction in our taxonomy and discuss the limitations of our work.

1.5 Contributions

This thesis contributes the following:

- We demonstrate the current limitations of Copilot’s code suggestions. We show that Copilot does not perform well in suggesting best practices and language idioms. We present our discussion on challenges by using AI-supported code completion tools like Copilot.
- We release our coding experiments on Copilot for best practices and language idioms. We make this available in our replication package [60].
- Using the findings from Copilot’s code suggestions on best practices and language idioms, we present a taxonomy of software abstraction hierarchy where

‘basic programming functionality’ such as code compilation and syntax checking is at the least abstract level, including the set of requirements for AI-supported code completion tools to satisfy each level of abstraction.

- Based on our experiences in this study, we present future directions for moving beyond code completion to AI-supported software engineering, which will require an AI system that can, among other things, understand how to avoid code smells, follow language idioms, and eventually propose rational software designs.

1.6 Thesis Outline

This thesis is organized as follows:

Chapter 2 elaborates the background information and some related work on AI-supported code completion tools and GitHub Copilot. It further introduces the challenges with using AI-supported code completion tools that will be explored in this thesis.

Chapter 3 introduces our study and the methodology showing the sampling approach, input, and evaluation criteria we used to address **RQ-1** (What are the current boundaries of code completion tools). We then present the results of Copilot code suggestions for language idioms and code smells.

Chapter 4 introduces a taxonomy of software abstraction hierarchy, inspired by SAE autonomous driving safety levels. It then presents a set of requirements for AI-supported code completion tools to satisfy each level of abstraction.

Chapter 5 addresses **RQ-2** (Given the current boundary, how far is it from suggesting design decisions?) with a discussion on the complex nature of design decisions. In addition, we discuss future directions for AI-supported code completion tools to reach the design abstraction level in the taxonomy. We conclude by discussing the implications and the limitations of this study.

Chapter 6 presents the conclusion of this research study.

Chapter 2

Background & Related Work

2.1 Introduction

In this chapter, we first highlight some related work that leads towards current AI-supported code completion tools like Copilot [27]. Further, we discuss existing research on Copilot. Finally, we provide details regarding alternative AI-supported code completion tools like Copilot.

2.2 Evolution of AI-supported code completion tools

A variety of data science and machine learning methodologies are applied in the field of AI-supported code completion tools. The following is an overview of current state-of-the-art methods for both text and code completion and how these approaches have evolved over time.

2.2.1 Statistical Language Models

Statistical language models (LMs) are mathematical models that express a probability distribution across a set of words. From a training dataset, LMs learn the probability of word occurrences, and LMs can predict the next word from a sequence of words based on the probability distribution of their training data. Most common LMs like N-grams are currently utilized to support many complicated models in modern approaches like Neural Language Models (NLMs).

2.2.1.1 N-grams

N-grams [66] are probabilistic language models represented by an (n-1)-order Markov source (a sequence of n-1 of random variables defined by a Markov chain). A Markov chain is based on the idea that the probability of the next word depends only on its n-1 predecessors. Since this property mostly holds for natural language, the Markov source is a good approach for modeling natural language. Code completion tools like SLANG [64] and CACHECA [24] are based on N-grams.

2.2.1.2 SLANG

SLANG [64] is a tool for completing gaps (missing API calls) in an API call sequence. It combines RNNME [50] with N-grams (n=3, trigrams). The evaluation of the study reveals that using the average of the probabilities generated by the two models for filling in gaps in API call sequences performs better than using either model alone, highlighting the applicability of N-gram LMs in support of more sophisticated models (RNNME in this case).

2.2.1.3 CACHECA

CACHECA [24] is an Eclipse IDE [18] plugin that tries to improve the code completion suggestions provided by Eclipse's default tool. It is created using a Cache language model [72], which offers words that have already been in the text a higher likelihood of occurrence. The localization of code asserts that code has local regularities (a variable will generally occur multiple times in a block of code) [72].

2.2.1.4 Problems with N-grams

N-grams are rarely employed as a stand-alone solution in modern natural language processing applications but rather as a support for a more complex model, which leads us to common problems with N-gram language models that make them unsuitable for practical code completion tools.

Short distance dependencies - N-grams can only represent relationships between a word and its n-1 predecessors, which implies they don't have long-distance dependencies and lose context after few lines of code. Because it would be difficult to foresee how an imported library function or an initialized variable would be used,

this restricts the potential use of N-grams for advanced code completion tasks such as suggesting variables or functions initialized in another class in the same file.

Out of Vocabulary (OOV) problem - In the specific application of code completion, it is possible that a developer utilizes a variable or library in our particular application that is not included in the training set. As a result, N-gram models would never predict that term, preventing us from using it as a prediction. This makes the code completion tools restricted to the knowledge of the training set, making it impossible to generalize.

2.2.2 Neural Language Models (NLMs)

N-grams are a simple statistical LM that is simple to comprehend and compute. However, it has no way of modeling multi-sentence context and cannot generalize to unseen data. Neural language models are introduced to tackle the drawbacks of LMs. The idea underlying NLMs is to project words from a corpus onto a continuous space and then apply a probability estimator that operates on that space. Initially, feedforward architecture was used in neural networks for language modelling [9].

2.2.2.1 Transformer Models

The transformer architecture is built using stacked self-attention mechanisms [74]. There are six levels in both the encoder and the decoder [74]. A feed-forward neural network and a self-attention mechanism (looking both at previous and future tokens) are the two components in each encoder layer. The decoder contains an additional attention mechanism that enables it to pay attention to specific encoder segments, in contrast to the encoder layer. In a machine translation test, it was demonstrated that the transformer design outperformed several NLM-based models [74]. Figure 2.1 shows the architecture of a transformer.

2.2.2.2 Contextualized Word Representations

The encoding of words as fixed-length vectors allowed for the use of Deep Learning in NLP. The need that all word senses have the same representation presented a challenge. Context-sensitive contextualised word representations were presented by LMs such as BERT [17], ELMo [57] and GPT-3 [10]. BERT has a variant for general-purpose representations that support downstream applications such as natural language code-search and code documentation generation called CodeBERT [22]

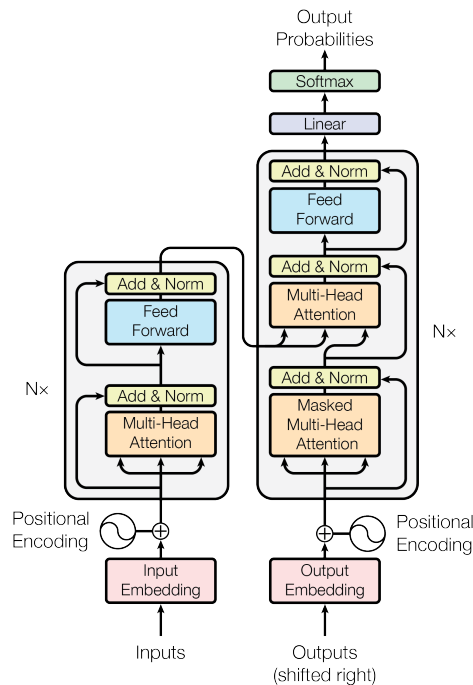


Figure 2.1: Transformer Architecture [74]

implying that representations of the same word vary depending on the context. Metrics in numerous NLP tasks have significantly improved because of these new word representations [20].

2.3 Code completion with NLMs

Language modeling faces a complex problem when it comes to code completion. There are local (to only one block of code) and very long-distance dependencies in coding, such as initializing a variable only once per function. So, approaches like pointer networks (Ptr-Net) [75] were introduced, which involved merging a model employing pointer networks with attention, which was shown to perform better at simulating long-distance dependencies than traditional NLMs [44].

By predicting words from the input sequence, Ptr-Nets offer a solution to the challenge of anticipating words that are not in one's vocabulary. However, Ptr-Nets can only expect words to appear in the local context because they cannot predict words outside the input sequence (like a block of code). Due to this restriction, a mixture of Attention (long dependencies) and Ptr-Nets (local dependencies) called Pointer Mixture Network [44] was introduced. This model predicts words from the

global vocabulary or copies one from a local context. In addition, a modified version of the Attention mechanism using an AST-based programming language was developed in this study [44]. These improvements to transformer architecture improved the performance of AI-supported code completion tools and led to the creation of large language models like Codex [12], which is the model used in the creation of Github Copilot [27].

2.3.1 GitHub Copilot

Introduced in June 2021, GitHub’s Copilot [27] is an in-IDE recommender system that leverages OpenAI’s Codex neural language model (NLM) [12] which uses a GPT-3 model [10]. This model comprises ≈ 13 billion parameters consuming hundreds of petaflop days to compute on Azure cloud platform [12] and is then fine-tuned on code from GitHub to generate code suggestions that are uncannily effective and can perform above human average on programming contest problems [54]. While many other research projects have attempted to do something similar [29, 31, 78], Copilot’s availability and smooth integration with GitHub’s backup have unavoidably generated a “hype” in the tech community, with many developers either already using it through its technical preview or started using it after its recent public launch as a paid subscription [27].

Using Copilot requires authorization using a GitHub account and is currently available for use on Visual Studio Code and other IDEs [27]. GitHub Copilot provides suggestions for numerous languages and a wide variety of frameworks but works especially well for Python, JavaScript, TypeScript, Ruby, Go, C# and C++ [27]. GitHub Copilot will automatically start giving suggestions when the user starts typing in the IDE, where the user has the option to use the suggestion, see alternative suggestions, or ignore the suggestion [27].

Currently, Copilot provides three key functionalities: autofill for repetitious code, suggest tests based on the implementation of code, and comment to code conversion [27]. For the scope of this thesis, we focus on the feature of turning comments into code when a user adds a remark describing the logic they want to utilise [27]. Although Copilot code suggestion can be triggered by adding a comment in natural language, it is advised that users add comments and meaningful names for function parameters to receive useful recommendations [27]. The human input we used to trigger code suggestions from Copilot concatenates the natural language comment,

function name, and function parameters.

Vaithilingam et al. [73] conducted an exploratory study of how developers use Copilot, finding that Copilot did not solve tasks more quickly but did save time in searching for solutions. More importantly, Copilot solved the writer's block problem of not knowing how to get started. This notion of seeding an initial, if incorrect, solution is often how design proceeds in software.

Recent work shows initial investigations on how large language models for code can add architecture tactics by using program synthesis [68, 32] and structure learning [37]. This thesis complements these earlier approaches by focusing on moving beyond code completion, where most research effort is currently concentrated. To provide deeper insights into the overall effectiveness of the instrument, it is crucial to evaluate the quality of Copilot's suggestions and understand its limitations.

2.3.2 Alternatives to Copilot

There are a handful of code completion systems currently being used. Below is a list of few such systems:

- Jedi [34] - An open source Python static analysis tool aimed at automatic code completion. Jedi has a focus on autocompletion and goto functionality. Other features include refactoring, code search, and finding references. However, performance issues and limitations on project size and complexity hamper its effectiveness.
- Kite [41] - Kite is an AI-powered code completion plugin that works with 16 languages and 16 IDEs. It uses machine learning to rank completions more intelligently based on your current code context [41].
- Deep TabNine [71] - Deep TabNine is a source code completion system which is based on OpenAI's GPT-2 Model [61]. It is trained on GitHub public repositories with permissive open-source licenses. Trained code is also filtered to ensure quality and avoid outdated, esoteric, auto-generated code and other edge cases [71].
- AlphaCode [45] - An AI-supported code completion tools by DeepMind, which uses a transformer language model to generate code, pre-trained on selected GitHub code and fine-tuning on a curated set of competitive programming

problems [45]. The training process for AlphaCode included tests for code cloning, sensitivity to problem descriptions, and metadata. AlphaCode model focuses on creating novel solutions to problems that require deeper reasoning.

- CodeBERT [22] - A bimodal pre-trained model for programming language (PL) and natural language (NL) by Microsoft. it uses multi-layer bidirectional Transformers for code completion [22]. CodeBERT learns general-purpose representations that support downstream NL-PL applications such as natural language code search, code documentation generation, etc. CodeBERT improved state-of-the-art performance on natural language code search and code documentation generation [22].
- Amazon CodeWhisperer [4] - A machine learning powered service that helps improve developer productivity by generating code recommendations based on their comments in natural language and code. It supports Python, Java and JavaScript [4]. The code recommendations provided by CodeWhisperer are based on models trained on various data sources, including Amazon and open-source code [4]. CodeWhisperer also examines the code of the current file and other files in the developer’s project to produce its recommendations [4].

2.4 Chapter Summary

In this chapter, we first provided some background on different language models used for AI-supported code completion tools and their limitations. We reviewed early developments in AI-supported code completion tools with statistical language models like N-grams. Followed by a discussion on N-gram based AI-supported code completion tools and the limitations of statistical language models, resulting in using neural language models for AI-supported code completion tools.

We established the importance of transformers for AI-supported code completion tools which is a crucial component of OpenAI’s Codex Model [12]. To further explore the role of transformer architecture in AI-supported code completion tools, we reviewed studies showing context-sensitive contextualized word representations were presented by LMs such as BERT [17].

We then discussed GitHub Copilot, the AI-supported code completion tools we would use as the basis for our study in this thesis. Additionally, we reviewed the key functionalities of Copilot. Furthermore, we discussed some related works on

Copilot about its usage [73] and its effectiveness in solving programming contest-style problems [54]. We concluded by introducing some of the other AI-supported code completion tools that provide similar functionality to Copilot.

In the following chapters, we discuss the problems with using AI-supported code completion tools like Copilot, which are harder to fix, and straightforward corrections that may not exist, like language idioms and code smells. We try to address **RQ-1** (What are the current boundaries of code completion tools) using the methodology and present our results (Chapter 3). We then introduce a taxonomy of software abstraction hierarchy to help with finding the current boundaries of AI-supported code completion tools like Copilot (Chapter 4).

We address **RQ-2** (Given the current boundary, how far is it from suggesting design decisions?) with a discussion of the complex nature of design decisions and the challenges with trying to use AI-supported code completion tools to make design decisions. Finally, we discuss some of the practical implications and limitations of our findings and also provide some future directions to help further research in AI-supported code completion tools (Chapter 5).

Chapter 3

Challenges with Copilot

3.1 Introduction

Useful AI-supported code completion tools should always suggest recommended coding best practices in its first suggestion. In this chapter, we test if Copilot suggests the optimal solution for a programming task. “Optimal ways” here means recommended ways of solving a programming task sampled from popular sources. We begin by explaining the current challenges with AI-supported code completion tools like Copilot, showing recent research works on common problems faced with using Copilot and the motivation to find the limitations of current AI-supported code completion tools like Copilot (section 3.2).

In section 3.3, we explain our approach to **RQ-1** (What are the current boundaries of AI-supported code completion tools?). We describe our sampling approach to collecting Pythonic idioms (section 3.3.1.1) and best practices in JavaScript (section 3.3.2.1). We then describe the input given to Copilot for triggering the generation of code suggestions (section 3.3.3). Finally, we explain our evaluation method to compare Copilot suggestions to the recommended practices (section 3.3.4).

In section 3.4, we present our results on performance of Copilot in suggesting recommended practices for 50 different coding scenarios (25 Pythonic idioms + 25 code smells), which answers **RQ-1.1** (How do AI-supported code completion tools manage programming idioms?), and **RQ-1.2** (How do AI-supported code completion tools manage to write non-smelly code?). We observe that Copilot had the recommended practices in its top 10 suggestions for 18 out of 50 coding scenarios (36% of all tests performed).

3.2 Background & Motivation

Code completion tools are very useful but are often limited to the generation of single elements (e.g., method calls and properties) and the usage of templates. Furthermore, too many recommendations can decrease the usefulness of the tool [59]. AI-supported code completion tools must be accurate with its code suggestions while minimizing the number of different code suggestions recommended to the user.

Copilot can make simple coding mistakes, such as not allowing for an empty array in a sort routine¹. Copilot does not understand security vulnerabilities, so it will suggest code that allows for a `log4shell` vulnerability², or common SQL injection attacks. A recent study by Pearce et al. [56] showed that approximately 40% of the code suggested by Copilot is vulnerable when tested on 89 different scenarios for Copilot to complete. Some of these basic programming challenges have been already documented and are, we suspect, very much under consideration by the corporate teams behind Copilot and Codex. Since these tools are trained on existing software source code and training costs are expensive, several classes of errors have been discovered, which follow from the presence of these same errors in public (training) data. Similarly, concerns have been raised about Copilot license compliance and copyright violation [14]; with similar input data, Copilot suggests identical code to existing code on GitHub, which may be under copyright. As it is trained on public data collected in May 2020 from 50 million public repositories on GitHub [12], any code uploaded after that date is absent in the knowledge base of Copilot.

Karampatsis et al. [36] showed that for the 1000 most popular open-source Java repositories on GitHub, there is a frequency of one single statement bug per 1600-2500 lines of code and about 33% of all the bugs match a set of 16 bug templates. So, any software flaws present in large numbers on GitHub will tend to dominate the learning of the model. But these challenges are not surprising and have straightforward fixes. These fixes might include better data engineering and filtering to remove known problems, like a filter introduced by GitHub to suppress code suggestions containing code that matches public code on GitHub, although the exact filtering process has not been publicly disclosed.

Similarly, it seems viable to conduct security scans or linting runs before suggesting removing obvious problems like SQL injection. Clone detection techniques

¹all examples are documented in our replication package [60].

²<https://www.wiz.io/blog/10-days-later-enterprises-halfway-through-patching-log4shell/>

can help find places where code violates the copyright. Better machine learning approaches, using active learning or fine-tuning, might help learn local lessons [48] for customization in the case of identifier naming or formatting. In most of these cases, good tools exist already for this.

Although these are clearly challenges, Copilot seems already to be on its way to fixing them, like a filter introduced by GitHub to suppress code suggestions containing code that matches public code on GitHub. However, what is more difficult to envision are the problems that are harder to fix because straightforward corrections may not exist, and rules for finding problems are more challenging to specify than those in smell detectors or linters [19] like language idioms and code smells.

Developers often discuss software architecture and actual source code implementations in online forums, chat rooms, mailing lists, or in person. Programming tasks can be solved in more than one way. The best way to proceed can be determined based on case-specific conditions, limits, and conventions. Strong standards and a shared vocabulary make communication easier while fostering a shared understanding of the issues and solutions related to software development. However, this takes time and experience to learn and use idiomatic approaches [3].

AI-supported code completion tools can help steer users into using more idiomatic approaches with its code suggestions or vice-versa. This makes it crucial to find the boundaries of AI-supported code completion tools like Copilot (**RQ 1**) and create a clear understanding of where can we use AI-supported code completion tools like Copilot and where should the user be vigilant in using AI-supported code completion tools code suggestions. To achieve this, we conduct an exploratory study to find if AI-supported code completion tools tools like Copilot suggest the recommended best coding practices in their suggestions.

3.3 Methodology

In this section, we explain the methodology we used to address **RQ-1** (What are the current boundaries of AI-supported code completion tools?). We perform our experiments Copilot suggestions on Pythonic Idioms (section 3.4.1) and code smells in JavaScript (section 3.3.2).

Additionally, we explain how 25 coding scenarios for Pythonic idioms (section 3.3.2.1) and code smells in JavaScript (section 3.3.1.1) were sampled. Finally, we discuss how the input is shaped to trigger Copilot to generate code suggestions (section 3.3.3)

and how Copilot suggestions are evaluated (section 3.3.4). The following analysis was carried out using the Copilot extension in Visual Studio Code. We use the most recent stable release of the Copilot extension available at the time of writing (version number 1.31.6194) in Visual Studio Code.

3.3.1 Pythonic Idioms

A software language is more than just its syntax and semantics; it is also a set of known effective ways to address real-world issues using it. The “Zen of Python” [58], a well-known set of high-level coding rules written by Tim Peters, states, “There should be one, and preferably only one, obvious way to accomplish anything.” This “one way to do it” is frequently referred to as the “Idiomatic” manner: the ideal solution to a particular programming task. A survey by Alexandru et al. [3] showed that developers learn idiomatic ways by reading code in repositories of other projects, online forums like StackOverflow, books, seminars, and coworkers. However, this takes time and experience to always use idiomatic approaches [3]. A good AI-supported code completion tools should always use idiomatic approaches in its code suggestions. To answer **RQ-1.1** (How do AI-supported code completion tools manage programming idioms?), we chose Python, one of the most popular programming languages, because Copilot’s base model Codex performs best in Python [12].

The definition for the term *Pythonic* in Python found in official Python glossary³ as follows:

An idea or piece of code follows the most common idioms of the Python language rather than implementing code using concepts common to other languages. For example, a common idiom in Python is to loop over all elements of an iterable using a for statement. Many other languages do not have this construct, so people unfamiliar with Python sometimes use a numerical counter instead, instead of the cleaner, Pythonic method³.

This definition indicates a broad meaning, referring to both concrete code and also *ideas* in a general sense. Many Python developers argue that coding the *Pythonic way* is the most accepted way to code by the Python community [3]. We consider an *idiom* to be any reusable abstraction that makes Python code more readable by shortening or adding syntactic sugar. Idioms can also be more efficient than a basic

³<https://docs.python.org/3/glossary.html#term-pythonic>

solution, and some idioms are more readable and efficient. The Pythonicity of a piece of code stipulates how concise, easily readable, and generally good the code is. This concept of Pythonicity, as well as the concern about whether code is Pythonic or not, is notably prevalent in the Python community but also in other languages, e.g. Go and gofmt, Java, C++ etc. Notably, Perl would be an exception as the philosophy for perl was "there are many ways to do it".

We sampled idioms from the work of Alexandru et al. [3], and Farook et al. [21], which identified idioms from presentations given by renowned Python developers that frequently mention idioms, e.g., Hettinger [30] and Jeff Knupp [42] and popular Python books, such as "Pro Python" [2], "Fluent Python" [63], "Expert Python Programming" [33].

3.3.1.1 Sampling Approach

We sampled the top 25 popular Pythonic idioms found in open source projects based on the work of Alexandru et al. [3], and Farook et al. [21]. The decision to sample *most popular* Pythonic idioms is taken to give the best chance for Copilot to suggest the Pythonic way as its top suggestion. As a result, Copilot will have the Pythonic way more frequently in its training data and more likely to suggest the Pythonic way in its suggestions. However, Copilot is closed source, and we cannot determine if the frequency of code snippets in training data affects Copilot's suggestions. Research by GitHub shows that Copilot can sometimes recite from its training data in "generic contexts"⁴, which may lead to potential challenges like license infringements (shown in section 3.2). Sampling the most frequently used idioms will also help understand if Copilot can recite idioms present in its training data (GitHub public repositories), which is the ideal behavior for AI-supported code completion tools.

3.3.2 Code Smells

A standard style guide is a set of guidelines that explain how code should be written, formatted and organized. Using a style guide ensures that code can be easily shared among developers. As a result, any new developer may immediately become familiar with a specific piece of code and write code that other developers will quickly and easily comprehend. A good AI-supported code completion tool should only suggest code consistent with coding style and pass code reviews by humans.

⁴<https://github.blog/2021-06-30-github-copilot-research-recitation/>

To answer **RQ-1.2** (How do AI-supported code completion tools manage to suggest non-smelly code?), we chose JavaScript to generalize our experiments with Copilot. We relied on the AirBNB JavaScript coding style guide [1], a widely used coding style and code review standard introduced in 2012, described as a “primarily reasonable approach to JavaScript” [1].

3.3.2.1 Sampling Approach

The AirBNB JavaScript coding style guide [1] is one of the first style guides for JavaScript and is one of the most popular coding style guides used in approximately a million repositories on GitHub [1]. It is a comprehensive list of best practices covering nearly every aspect of JavaScript coding like objects, arrays, modules, and iterators. However, it also includes project-specific styling guidelines like naming conventions, commas, and comments. Since we are testing Copilot for widely accepted best practices and not project-specific styling in JavaScript. We sampled 25 best practices from the AirBNB JavaScript coding style guide [1], which were closer to the design level rather than the code level. For example, selecting logging practices as a sample coding standard rather than trailing comma use in JavaScript as a coding standard. This sampling approach ensures Copilot is not tested against personalized styling guidelines of one specific project or a company. In contrast, our goal for Copilot here is to be tested against practices that bring performance or efficiency to the code base.

3.3.3 Input to Copilot

The input for Copilot to trigger code suggestions consists of two parts. First, the title of the coding scenario is tested as the first line as a comment to provide context for Copilot to trigger relevant suggestions while stating the motive of the code scenario. Second, minimal input of code is required to trigger the code suggestion. Moreover, the code input was restricted to being able to derive the best practice from the information. This is to ensure Copilot is deciding to suggest the good/bad way in its suggestions and not being restricted by the input to suggest a certain way.

Copilot does not have the functionality to override or update the input, and it will only suggest code that matches the input. So, it is important to restrict the input to accurately test Copilot without limiting its possibility of creating different coding scenarios, which may include the best practice we desired. For example, figure 3.1

shows the example of list comprehension idiom where human input is restricted to just declaring a variable “result_list”. If the input included initializing the variable with some integers or an empty list, then Copilot is forced to use a for loop in the next line to perform list comprehension eliminating the possibility of suggesting the idiomatic approach. Although, it is a desirable feature for AI-supported code completion tools to override or update the input, Copilot does not support it yet. So, we restrict the input to being able to derive the best practice from the information.

This input style also mimics a novice user, who is unaware of the idioms or best practices. Useful AI-supported code completion tools like Copilot should drive the novice user to use best practices to perform a task in their codebases and improve the quality of their code.

3.3.4 Evaluation of Copilot code suggestions

We compare Copilot code suggestions against Pythonic idioms and best practices retrieved from our sources (Alexandru et al. [3] and Farook et al. [21] for Pythonic idioms and AirBNB JavaScript coding style guide [1] for JavaScript code smells). when Copilot manages to match the Pythonic idiom or the best practice as its first suggestion, we considered it as Copilot suggested the desired approach and passed the coding scenario. In contrast, if Copilot did not have Pythonic idiom or the best practice in any of its all 10 code suggestions currently viewable using Copilot extension in Visual Studio Code, we considered Copilot did not suggest the desired approach and failed the coding scenario.

We assume that AI-supported code completion tools like Copilot are productivity tools, and the user should be saving time as opposed to writing the optimal way without using AI-supported code completion tools, scrolling through all the suggestions to deduce the idiomatic approach or the best practice that follows the coding style guide defeats this purpose. For this reason, We restricted ourselves to the first suggestion of Copilot to be considered in determining the Pass/Fail status of the coding scenario. However, we note if the best practice appeared in any of its ten suggestions.

3.4 Results

In section 3.3, we discussed our sources, the sampling approach for Pythonic Idioms, and the JavaScript coding style guide. We then discussed how the input for Copilot

to trigger code suggestions is restricted to ensure Copilot is deciding to suggest the desired way or vice-versa. Finally, we discussed the evaluation method used to compare Copilot’s code suggestions to the idiomatic approaches and the best practices listed in the coding style guide (section 3.3.4).

In this section, we show the results of the study comparing Copilot suggestions against Pythonic idioms (section 3.4.1) addressing **RQ-1.1** (How do AI-supported code completion tools manage programming idioms?) and JavaScript coding style guide (section 3.3.2) addressing **RQ-1.2** (How do AI-supported code completion tools manage manage to suggest non-smelly code?).

3.4.1 Pythonic Idioms

Using the sampling approach described in section 3.3.1.1, we sampled the 25 most popular Python idioms from the work of Alexandru et al. [3], and Farook et al. [21]. We then compared Copilot suggestions when prompted with an input (shown in section 3.3.3) to trigger a code suggestion from Copilot and present our results using the evaluation approach (shown in section 3.3.4).

Copilot suggested the idiomatic approach as the first suggestion in 2 of the 25 idioms we tested, i.e., 2 out of 25 instances, Copilot had the recommended idiomatic approach as its top suggestion. However, 8 out of those remaining 23 Idioms had the idiomatic way in Copilot’s top 10 suggestions. Copilot did not have the idiomatic way in its top 10 suggestions for 15 idioms out of 25 we tested.

The results show that Copilot did not suggest the optimal way as its first suggestion in the majority (92%) of the idioms we tested. This indicates that current AI-supported code completion tools like Copilot cannot suggest the idiomatic way even though they are the top most frequently used Python idioms in public repositories on GitHub [3, 21].

Copilot being closed source, we cannot investigate the potential reasons behind this behavior. However, one plausible explanation for this behavior is that idiomatic ways may not be as frequent as non-idiomatic ways in Copilot’s training data of public repositories on GitHub, making the non-idiomatic way rank higher than the idiomatic way.

Figure 3.1 shows the example of list comprehension Pythonic idiom, showing user input (i.e., human input), the top suggestion by Copilot, and the idiomatic way from Alexandru et al. [3].


```

List Comprehension
Human Input:
1 #list comprehension
2 result_list =
-----
Copilot Suggestion:
1 result_list = []
2 for i in range(1,11):
3     result_list.append(i)
-----
Pythonic waya:
1 result_list = [e1 for e1 in range(11)]
-----
asource [3]

```

Figure 3.1: List comprehension Pythonic idiom and Copilot top suggestion.

The list comprehension suggested by Copilot may be accepted in other languages. However, in Python there is a concise way saving 2 lines of code in this context, and it is also more readable. Additionally, if we make the input size bigger we notice that Pythonic list comprehension is faster as well. (≈ 1 sec faster when tested on a sample of 100 million).

Another example of Pythonic ways being faster than Copilot code suggestions is shown in Figure 3.2. The code suggested by Copilot finds the index of every word in a given input string. However, there is not only a lot of visual noise that comes from needing to initialize and build the list, but also the fact that when this function is called, it needs to build the entire list before returning. If we want, say, the first 2 elements only, then we wasted resources unnecessarily to build the entire rest of the list.

The Pythonic way of finding the index of every word in a given input string is to use a generator and when “next()” is called, we use the next “yield” statement. If a yield statement is encountered, the state of the function is frozen, and the value of is returned to .next()’s caller. By “frozen” we mean that all local state is retained. This creates a lazy approach to finding index of every word in input string. The time taken to execute the Copilot suggested function for the text shown in Figure 3.2 is 6.91 seconds. The Pythonic approach completed the function for the same text in

1.1e-05 seconds using its lazy approach of finding index of each word in input string, making it significantly faster than the Copilot suggested code.

```

Find index of every word in input string

Human Input:
1 # find index of every word in input string
2 text = "this is Copilot testing...."
3 def index_words(text):

-----

Copilot Suggestion:
1 def index_words(text):
2     result = []
3     for i, word in enumerate(text.split()):
4         result.append((i, word))
5     return result

-----

Pythonic waya:
1 def index_words(text):
2     if text:
3         yield 0
4     for index, letter in enumerate(text):
5         if letter == ' ':
6             yield index + 1

-----
asource [3]

```

Figure 3.2: Find index of every word in input string Pythonic idiom and Copilot top suggestion.

The most Pythonic way of performing a task is the clearest, the best, or even the most optimal in a computational sense. Another important characteristic of idiomatic code is to make the code easier to understand and maintain. Being Pythonic helps to detect errors and even make fewer mistakes.

Figure 3.3 shows Copilot code suggestion for opening and writing to a file. If we use code suggested by Copilot and if there is an exception while writing, the file will not be closed as the `f.close()` line will not be reached by the interpreter. However, in Pythonic way, the file is closed regardless of what happens inside the indented block of code. Clearly, Pythonic way is superior.

```

Open and write to a file

Human Input:
1 #Open and write to a file

-----

Copilot Suggestion:
1 f = open('idioms.txt', 'w')
2 f.write('Hello World')
3 f.close()

-----

Pythonic waya:
1 with open('idioms.txt', 'w') as f:
2     f.write('Hello, World!')

-----

asource [3]

```

Figure 3.3: Open and write to a file Pythonic idiom and Copilot top suggestion.

Table 3.1 shows the list of all the 25 Pythonic idioms we tested and the ranking of the Pythonic way in Copilot suggestions (if it exists). All the Idioms shown in Table 3.1 can be found in the replication package [60] including the code used as input (i.e., human input), the top suggestion by Copilot, and the Pythonic way suggested in Alexandru et al. [3], and Farook et al. [21].

Copilot had the Pythonic way in its top 10 suggestions for 8 coding scenarios, where Copilot ranked the non-idiomatic approach as the top suggestion. The ranking methodology of Copilot is not disclosed. However, the results suggest that it is heavily influenced by the frequency of the approach in the training data. Copilot successfully suggested the idiomatic approach as its top suggestion in ‘set comprehension’ and ‘if condition check value’ (idiom 7 & 10 in table 3.1), which are one of the most frequently occurring idioms in open source code [3].

Copilot is more likely to have the idiomatic approach in its top 10 suggestions when there are only a few ways of performing a task. For example, consider the ‘Boolean comparison idiom’; there are only two most common ways of performing the task, i.e., ‘if boolean:’ or ‘if boolean == True.’ Copilot had the non-idiomatic approach higher than the idiomatic approach in this case.

AI-supported code completion tools like Copilot should learn to detect idiomatic ways in public repositories and rank them higher than the most frequently used way

in public repositories so that the first suggestion would be the idiomatic way rather than the non-idiomatic way, which is the desired behavior for AI-supported code completion tools like Copilot. For the scope of this thesis, we leave resolving this problem as future work.

3.4.2 Code Smells

Using the sampling approach described in section 3.3.1.1, we sampled 25 best practices in JavaScript from the AirBNB JavaScript coding style guide [1]. We sampled best practices that are closer to the design level rather than the code level. For example, selecting logging practices as a sample coding standard rather than trailing comma use in JavaScript as a coding standard.

We then compared Copilot suggestions when prompted with an input (shown in section 3.3.3), which includes the title of the coding scenario being tested as the first line to provide context for Copilot and help trigger a code suggestion. We then present our results using the evaluation approach (shown in section 3.3.4).

Copilot suggested the best practice from the AirBNB JavaScript coding style guide [1] for 3 out of the 25 coding standards we tested, i.e., 3 out of 25 instances Copilot had the recommended best practice as its top suggestion. Moreover, only 5 of the remaining 22 coding scenarios had the best practice in Copilot's top 10 suggestions currently viewable. Copilot did not have the best practice in its top 10 suggestions for 17 scenarios out of 25 coding scenarios we tested.

The results show that Copilot did not suggest the recommended best practice as its first suggestion in the majority (88%) of the best practices we tested. As Copilot is closed source, we cannot find the reason behind this, but one could argue that lack of data for JavaScript compared to Python could be a reason for this behavior.

We did not test Copilot for suggesting project-specific coding styles because Copilot does not have the feature to customize its suggestions based on preferences [27]. However, AI-supported code completion tools like Copilot should follow coding style guides and adapt their code suggestions to match the coding style used in the project. For example, if a user is working on a project where one of the coding style guidelines says to leave a blank line after blocks and before the next statement [1]. As a productivity tool, the ideal behavior for AI-supported code completion tools like Copilot is to detect the coding style guideline from existing code or a coding style document in the project and always suggest code that follows the guidelines.

S No.	Idiom Title	Copilot Suggestion Matched? (out of 10 suggestions)
1	List comprehension	No
2	Dictionary comprehension	No
3	Mapping	9 th
4	Filter	7 th
5	Reduce	9 th
6	List enumeration	No
7	Set comprehension	1st
8	Read and print from a file	5 th
9	Add int to all list numbers	No
10	If condition check value	1st
11	Unpacking operators	No
12	Open and write to a file	6 th
13	Access key in dictionary	No
14	Print variables in strings	No
15	Index of every word in input string	No
16	Boolean comparision	2 nd
17	Check for null string	5 th
18	Check for empty list	4 th
19	Multiple conditions in if statement	No
20	Print everything in list	No
21	Zip two lists together	No
22	Combine iterable separated by string	No
23	Sum of list elements	No
24	List iterable creation	No
25	Function to manage file	No

Table 3.1: List of all Pythonic idioms tested on Copilot.

Figure 3.4 shows the Best Practice for Copying Array Contents, showing user input (i.e., Human Input), the top suggestion by Copilot, and the recommended way suggested by AirBNB JavaScript coding style guide [1]. The code suggested by Copilot uses a for loop to iterate through all the contents of the array and copy them, which is a lot slower than the spread approach ([...items]) shown as the best practice to copy array in the AirBNB JavaScript coding style guide [1]. This eliminates the need to iterate through all the elements of the array to copy each one.

Copying Array Contents

Human Input:

```
1 // copying array contents
```

Copilot Suggestion:

```
1 const len = items.length;
2 const copy = new Array(len);
3 for (let i = 0; i < len; i++) {
4     copy[i] = items[i];
5 }
```

Ideal way^a:

```
1 const itemsCopy = [...items];
```

^asource [1]

Figure 3.4: Best practice for copying array contents and Copilot top suggestion.

Another example for best practices being better than Copilot code suggestions is shown in figure 3.5. The AirBNB JavaScript coding style guide [1] recommends to always use ‘const’ for all the references and avoid using ‘var’ because using ‘const’ ensures that you cannot reassign your references, which in turn can lead to bugs and difficulty in code comprehension. Copilot suggested to use ‘var’ as its first suggestion. This shows that code suggested by Copilot has flaws and does not follow the recommended best practices.

Figure 3.6 shows the best practice to return a string and a variable name from a function. The AirBNB JavaScript coding style guide [1] suggests to use template strings instead of concatenation, for programmatically building up strings because template strings give a readable, concise syntax with proper newlines and string

```

Create two references
Human Input:
1 // create two references
-----
Copilot Suggestion:
1 var a = 1;
2 var b = 2;
-----
Ideal waya:
1 const a = 1;
2 const b = 2;
-----
asource [1]

```

Figure 3.5: Best practice for creating two references and Copilot top suggestion.

interpolation features. Copilot suggested concatenation as its first suggestion (shown in figure 3.6).

Table 3.2 shows the complete list of all the best practices we tested on Copilot sampled from the AirBNB Coding Style guide [1] and the ranking of the best practice in Copilot suggestions (if it exists).

All the best practices shown in Table 3.2 can be found in the replication package [60] including the code used as input (i.e., human input), the top suggestion by Copilot, and the best practice from AirBNB JavaScript coding style guide [1].

The results show that Copilot performed worse than the language idioms. This indicates that current AI-supported code completion tools like Copilot are not yet capable of suggesting the best practices in their suggestions, even though the best practices are sampled from a widely accepted coding style guide.

There could be many reasons for this performance, like the public repositories do not always follow coding standards, and Copilot cannot detect coding styles from repositories with contribution guides, including the coding standards followed in the project. Copilot being closed source, we cannot investigate the potential reasons behind this behavior and recommend ways to fix this issue, improving the performance of Copilot. However, improving the frequency of best practice usage in training data and including metrics such as repository popularity in ranking of code suggestions could be some potential areas to explore for improving performance of Copilot.

```

Return string and variable name
Human Input:
1 // return string and variable name
2 function sayhi(name){
-----
Copilot Suggestion:
1 function sayhi(name){
2     return "Hello " + name;
3 }
-----
Ideal waya:
1 function sayHi(name) {
2     return "Hello, ${name}";
3 }
-----
asource [1]

```

Figure 3.6: Best practice for returning string and variable name and Copilot top suggestion.

Copilot had the recommended best practice in its top 10 suggestions for 5 coding scenarios, where Copilot did not rank the recommended best practice as the top suggestion. The ranking methodology of Copilot is not disclosed. However, the results suggest that it is heavily influenced by the frequency of the approach in the training data. Copilot successfully suggested the recommended best practice as its top suggestion in ‘accessing properties,’ ‘converting an array-like object to an array,’ and ‘Check boolean value’ (best practice 7, 15 & 23 in table 3.2), which are one of the most common practices used by beginners to perform the task [1].

Based on the results shown in table 3.2, Copilot is more likely to have the recommended best practice in its top 10 suggestions when it is a common beginner programming task like finding ‘sum of numbers’ or ‘importing a module from a file.’ We also observed that Copilot did not always generate all 10 suggestions like in the case of Pythonic idioms, and it struggled to come up with 10 suggestions to solve a programming task. This shows that Copilot does not have enough training data compared to Python to create more relevant suggestions, which may include the recommended best practices in JavaScript.

The ideal behavior for AI-supported code completion tools like Copilot is to suggest best practices extracted from public code repositories (training data) to avoid code smells. Additionally, AI-supported code completion tools like Copilot should detect the project’s coding style and adapt its code suggestions to be helpful for a user as a productivity tool. For the scope of this thesis, we leave resolving this problem as future work.

3.4.3 Summary of Findings

In an attempt to find the boundaries of AI-supported code completion tools like Copilot, we analyzed Copilot code suggestions for Pythonic idioms and JavaScript best practices. We identified that Copilot did not suggest the idiomatic way as its top suggestion for 23 out of 25 coding scenarios in Python. Furthermore, we identified that Copilot did not suggest the recommended best practice for 22 out of 25 coding scenarios in JavaScript.

Although Copilot is very good at solving well-specified programming contest style problems [54], our experiments show that it does not do well in following idioms and recommending best practices in its code suggestions. Additionally, AI-supported code completion tools like Copilot being a productivity tool, should be able to suggest idiomatic approaches and recommended best practices in its code suggestions to be helpful for the user. Studies like ours might help use this delineation to understand what might help turn AI-supported code completion tools such as Copilot into full-fledged AI-supported software development tools.

3.5 Chapter Summary

In summary, we start this chapter by showing the methodology used in addressing textbfRQ-1 (What are the current boundaries of AI-supported code completion tools?). We first introduced Pythonic idioms and best practices in JavaScript. We then present our sampling approach for sampling 25 coding scenarios to analyze Copilot code suggestions. Furthermore, we discussed the input given to Copilot to trigger a code suggestion and how the input was restricted to deriving the desired way from the input. Finally, we described our evaluation approach for Copilot code suggestions.

We sampled 25 Pythonic idioms from Alexandru et al. [3], and Farook et al. [21]. We identified that Copilot did not suggest the idiomatic way as its top suggestion

S No.	Best Practice Title	Copilot Suggestion Matched? (out of 10 suggestions)
1	Usage of Object method shorthand	No
2	Array Creating Constructor	6 th
3	Copying Array Contents	No
4	Logging a Function	No
5	Exporting a Function	No
6	Sum of Numbers	9 th
7	Accessing Properties	1st
8	Switch case usage	No
9	Return value after condition	No
10	Converting Array-like objects	No
11	Create two references	5 th
12	Create and reassign reference	No
13	Shallow-copy objects	No
14	Convert iterable object to an array	No
15	Converting array like object to array	1st
16	Multiple return values in a function	No
17	Return string and variable name	No
18	Initialize object property	No
19	Initialize array callback	No
20	Import module from file	6 th
21	Exponential value of a number	No
22	Increment a number	2 nd
23	Check boolean value	1st
24	Type casting constant to a string	No
25	Get and set functions in a class	No

Table 3.2: List of all JavaScript best practices tested on Copilot.

for 23 out of 25 coding scenarios in Python, which addressed **RQ-1.1** (How do AI-supported code completion tools manage programming idioms?). Furthermore, we sampled 25 best practices in JavaScript from the AirBNB JavaScript coding style guide [1]. We identified that Copilot did not suggest the recommended best practice for 22 out of 25 coding scenarios in JavaScript, which addressed **RQ-1.2** (How do AI-supported code completion tools manage to suggest non-smelly code?).

In this chapter, we showed that Copilot struggles to detect and follow coding style guides present in public repositories of GitHub and always suggests code that follows those coding style guides. We also observed that Copilot struggles to detect and most common idiomatic ways present in public repositories of GitHub and rank them higher than the non-idiomatic ways. Identifying this delineation could help in urn AI-supported code completion tools such as Copilot into full-fledged AI-supported software engineering tools.

In the next chapter (chapter 4), we illustrate our taxonomy inspired by autonomous driving levels on the software abstraction hierarchy in AI-supported software development and use the results shown in this chapter to delineate where AI-supported code completion tools like Copilot currently stands in the taxonomy.

Chapter 4

Framework

4.1 Introduction

Copilot works best in creating boilerplate and repetitive code patterns [27]. However, the code suggested by AI-supported code completion tools like Copilot are found to have simple coding mistakes and security vulnerabilities. Several classes of errors have been discovered, which follow from the presence of these same errors in training data of Copilot (shown in section 3.2). In Chapter 3, we identified that Copilot does not perform well in detecting and suggesting Pythonic idioms and best practices in JavaScript. The scope of capability and the quality of code suggestions made by AI-supported code completion tools like Copilot is uncertain.

In this chapter, we try to create a metric for answering **RQ-1** (What are the current boundaries of code completion tools) with a taxonomy of six software abstraction levels to help access the current capabilities of AI-supported code completion tools such as Copilot. We explain each software abstraction level in the taxonomy and the capabilities required by AI-supported code completion tools to satisfy the software abstraction level. We try to delineate where current AI-supported code completion tools such as Copilot, are best able to perform and where more complex software engineering tasks overwhelm them using a software abstraction hierarchy where “basic programming functionality” such as code compilation and syntax checking is the lowest abstraction level, while software architecture analysis and design are at the highest abstraction level. Additionally, we use a sorting routine as an example scenario to show how a code suggestion from AI-supported code completion tool looks like in every level of abstraction in our taxonomy.

Finally, we try to address **RQ-2** (Given the current boundary, how far is it from suggesting design decisions?) with a discussion on the level of complexities and challenges involved in creating AI-supported code completion tools that can satisfy design level compared to AI-supported code completion tools satisfying code smells level in our taxonomy.

4.1.1 Motivation

To center our analysis on creating a software abstraction hierarchy to create a metric for answering **RQ-1** (What are the current boundaries of code completion tools), we leverage an analogous concept in the more developed (but still nascent) field of autonomous driving. Koopman has adapted the SAE Autonomous Driving safety levels [23] to seven levels of autonomous vehicle safety hierarchy of needs shown in figure 4.1.



Figure 4.1: Koopman’s Autonomous Vehicle Safety Hierarchy of Needs [43]. SOTIF = safety of the intended function.

The pyramid concept is derived from that of Maslow [47], such that addressing aspects on the top of the pyramid requires the satisfaction of aspects below. For example, before thinking about system safety (such as what to do in morally ambiguous scenarios), the vehicle must first be able to navigate its environment reliably (“Basic Driving Functionality”).

We think that a similar hierarchy exists in AI-supported software development. For example, the basic driving functionality in Figure 4.1 is satisfied when the vehicle works in a defined environment without hitting any objects or other road users. This could be equivalent to code completion tools being able to write code without any obvious errors like syntax. Hazard analysis level requires vehicles to analyze and mitigate risks not just from driving functions, but also potential technical malfunctions, forced exits from the intended operational design domain, etc. This could be equivalent to writing bug free code and avoiding code smells in software development perspective. The system level safety could be equivalent of software design, where tools need move beyond code completion and satisfy system quality attributes such as performance and following idiomatic approaches.

Addressing aspects on the top of the pyramid requires the satisfaction of aspects below. Similarly, for AI-supported software development tools, before worrying about software architecture issues, that is, satisfying system quality attributes such as performance and following idiomatic approaches, AI-supported software development tools need to exhibit “basic programming functionality”. This basic functionality is where most research effort is concentrated, such as program synthesis, AI-supported code completion tools, and automated bug repair.

4.2 Taxonomy

Our taxonomy is a software abstraction hierarchy where “basic programming functionality” such as code compilation and syntax checking is the lowest abstraction level, while Software architecture analysis and design are at the highest abstraction level. As we ascend the levels, just as with Koopman’s pyramid (shown in figure 4.1), software challenges rely more on human input and become more difficult to automate (e.g., crafting design rules vs. following syntax rules).

Figure 4.2 shows the taxonomy of autonomy levels for AI-supported code completion tools. The more abstract top levels depend on the resolution of the lower ones. As we move up the hierarchy, we require more human oversight of the AI; as we move

down the hierarchy, rules for detecting problems are easier to formulate. Green levels are areas where AI-supported code completion tools like Copilot works reasonably well, while red levels are challenging for Copilot based on tests shown in Chapter 3.

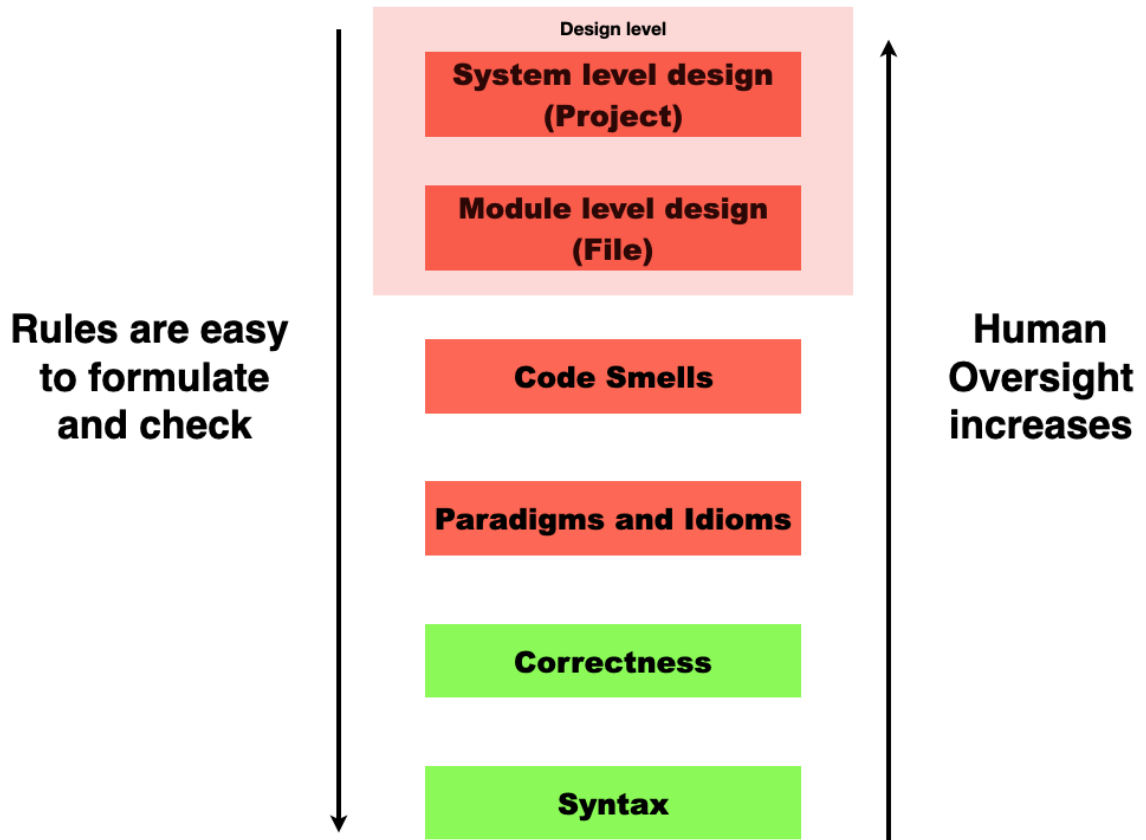


Figure 4.2: Hierarchy of software abstractions. Copilot cleared all green levels and struggled in red levels.

Based on our tests with Copilot for Pythonic idioms and JavaScript best practices (shown in Chapter 3), Copilot was able to generate syntactically correct code that solves the given programming task in the coding scenario¹. This functionality covers the syntax and the correctness level in our software abstraction hierarchy. As a result, Copilot stands at the correctness level of our taxonomy.

The challenges further up the hierarchy are nonetheless more important for software quality attributes (QA) [19] and for a well-engineered software system. For example, an automated solution suggested by AI-supported code completion tools to the top level of the taxonomy would be able to follow heuristics to engineer a

¹all coding scenarios tested are documented in our replication package [60].

well-designed software system, which would be easy to modify and scale to sudden changes in use.

4.2.1 Syntax

The syntax level is the lowest software abstraction level in our taxonomy. This level includes the most basic programming functionality like suggesting code that has correct syntax and has no errors in code compilations. This level does not require the AI-supported code completion tools suggested code to successfully solve the programming task but to suggest code without any obvious errors like syntax or code compilation errors.

For example, consider a programming task of performing a sorting operation on a list of numbers. To satisfy this level of abstraction, AI-supported code completion tools should suggest code that is syntactically correct without any compilation errors and the code is not required to perform the sorting operation correctly. Figure 4.3 shows the sorting example and Python syntax suggestions from AI-supported code completion tools at this abstraction level.

```

Syntax level suggestion for sort routine

Human Input:
1 # sorting a list
2 arr = [2,4,1,3,7,5]

-----

AI-supported code completion tools suggestion:
1 for i in range( ):

```

Figure 4.3: Code suggestion of AI-supported code completion tools at syntax level.

The goal of this software abstraction level in our taxonomy is for a AI-supported code completion tools to be able to suggest code without any syntactical errors. The capabilities required by AI-supported code completion tools to satisfy this level of abstraction are as follows:

1. Suggested code should be syntactically correct.
2. Suggested code should not produce any errors in code compilation.

4.2.2 Correctness

Correctness is the second level of software abstraction in our taxonomy. AI-supported code completion tools at this level should be capable of suggesting code that is not only syntactically correct but also solves the programming task. This level does not require AI-supported code completion tools to suggest the best possible coding solution for the programming task but to be able to suggest a decent solution to the programming task which may or may not resolve all the edge cases of the given programming task. Here “correctness” refers to performing the intended function specified by the user.

For example, consider the programming task of performing a sorting operation on a list of numbers. To satisfy this level of abstraction, AI-supported code completion tools should suggest a syntactically correct list sorting code, which is not required to be the most efficient way of sorting a list. Figure 4.4 shows the list sorting example and the Python code suggestion from AI-supported code completion tools at this abstraction level, which performs the sorting operation.

Correctness level suggestion for sort routine

Human Input:

```

1 # sorting a list
2 arr = [2,4,1,3,7,5]

```

AI-supported code completion tools suggestion:

```

1 n = len(arr)
2 for i in range(n):
3     for j in range(n-1):
4         if arr[j] > arr[j+1]:
5             temp = arr[j]
6             arr[j] = arr[j+1]
7             arr[j+1] = temp
8 print(arr)

```

Figure 4.4: Code suggestion of AI-supported code completion tools at correctness level.

The goal of this software abstraction level in our taxonomy is for AI-supported code completion tools to be able to suggest a solution instead of the best one. The

capabilities required by AI-supported code completion tools to satisfy this level of abstraction are as follows:

1. Suggest a solution for a given programming task that may not be the optimal solution for that programming task.
2. The solution suggested is not required to cover all the edge cases for that programming task.
3. Satisfy requirements of all the levels below correctness in our taxonomy.

4.2.3 Paradigms and Idioms

Paradigms and idioms are our taxonomy's third level of software abstraction. This level requires the code suggested by AI-supported code completion tools to satisfy all the previous levels of abstractions and use common paradigms and language idioms in its code suggestions. These include common practices of solving a programming task.

Returning to our running example of performing a sorting operation on a list of numbers. To satisfy this level of abstraction, AI-supported code completion tools should suggest a syntactically correct list sorting code, using idiomatic ways in its code suggestions, like the Pythonic way of swapping items in a list (line 5 in figure 4.5), As opposed to suggesting non-idiomatic approaches like creating another temporary variable to swap items in a list shown in correctness level (figure 4.4).

Figure 4.5 shows the sorting example and the Python code suggestions from AI-supported code completion tools at this abstraction level.

The goal of this software abstraction level in the taxonomy is for AI-supported code completion tools to detect and use commonly known idiomatic approaches and paradigms that occur in public code in its suggestions for suggesting code to solve a programming task.

The capabilities required by AI-supported code completion tools to satisfy paradigms and idioms level of software abstraction are as follows:

1. Identify common patterns like paradigms and language idioms in public code repositories (training data).
2. Use paradigms and language idioms in suggesting solutions for a programming task.

```

Idioms level suggestion for sort routine
Human Input:
1 # sorting a list
2 arr = [2,4,1,3,7,5]
-----
AI-supported code completion tools suggestion:
1 n = len(arr)
2 for i in range(n):
3     for j in range(n-1):
4         if arr[j] > arr[j+1]:
5             arr[j], arr[j+1] = arr[j+1], arr[j]
6 print(arr)

```

Figure 4.5: Code suggestion of AI-supported code completion tools at paradigms and idioms level.

3. Satisfy requirements of all the levels below paradigms and idioms in our taxonomy.

4.2.4 Code Smells

The code smells level is the next level of software abstractions after our taxonomy's paradigms and idioms level. Code smells level requires the suggested code to satisfy all the previous levels of abstractions and avoid common code smells in its suggestions. These include common bad practices found in public code. This software abstraction level also requires AI-supported code completion tools to suggest the most optimized version of all of the code suggestions AI-supported code completion tools could generate for the programming task.

Returning to our running example of performing a sorting operation on a list of numbers. To satisfy this level of abstraction, AI-supported code completion tools should suggest a syntactically correct list sorting code, using common patterns like idioms (line 8 in figure 4.6) and not including common code smells that occur in public code like edge case scenarios.

Figure 4.6 shows the example and the suggestion from AI-supported code completion tools at this abstraction level.

The goal of this level of software abstraction in our taxonomy is for AI-supported

```

Code smells level suggestion for sort routine

Human Input:

1 # sorting a list
2 arr = [2,4,1,3,7,5]

-----

AI-supported code completion tools suggestion:

1 swapped = True
2 n = len(arr)
3 j = 0
4 while(swapped==True and n>1):
5     swapped = False
6     for i in range(n-j-1):
7         if arr[i] > arr[i+1]:
8             arr[i], arr[i+1] = arr[i+1], arr[i]
9             swapped = True
10    n-=1
11    j+=1
12 print(arr)

```

Figure 4.6: Code suggestion of AI-supported code completion tools at code smells level.

code completion tools to be able to detect and avoid bad practices such as code smells that commonly occur in public code in its code suggestions to a problem and to suggest the most optimized version as its first suggestion to solve a programming task.

The capabilities required by AI-supported code completion tools to satisfy this level of abstraction are as follows:

1. Identify common bad practices such as code smells that occur in public code (training data).
2. Suggest solutions that do not have code smells and unresolved edge cases.
3. Suggested code should be the most optimized version of all the possible suggestions AI-supported code completion tools could create for a given problem.
4. AI-supported code completion tools should not suggest code that needs to be immediately refactored.

5. Satisfy requirements of all the levels below code smells in our taxonomy.

4.2.5 Design level

Software design is the highest level of abstraction in our taxonomy. The goal of this level is to make AI-supported code completion tools support the user in every software development process and suggest improvements. To simplify the taxonomy of overall design processes in software development, we divided it into two subcategories: Module level design and System level design. AI-supported code completion tools at the Module level design requires more user involvement in making design choices at the file level. In system level design, AI-supported code completion tools are more autonomous and require minimal input from the user in making design choices.

4.2.5.1 Module level design

Module level design is the first half of our taxonomy's design level of software abstraction. This level requires the suggested code to be free of all known vulnerabilities, include test cases and continuous integration (CI) methods such as automating the process of performing build and testing code of the software when applicable. Code suggestions should also cover all the functional requirements of a given programming task.

AI-supported code completion tools at this level should be able to pick and suggest the best applicable algorithm for a given coding scenario and be capable of following user-specified coding style guidelines. For example, consider the task of sorting operation on a list of numbers. To satisfy this level of abstraction, AI-supported code completion tools should suggest a syntactically correct list sorting code, using an algorithm that gives the best performance for that particular input scenario, like suggesting a quick sort algorithm (avg time complexity = $n \log n$) instead of bubble sort algorithm (avg time complexity = n^2) unless specifically requested by the user.

The goal of this level in the taxonomy is for AI-supported code completion tools to be able to suggest appropriate design choices at the file level, considering the input from the user, like coding style guidelines, and help the user make design choices that satisfy all the functional requirements of the given programming task.

The capabilities required by AI-supported code completion tools to satisfy this level of abstraction are as follows

1. Picking and suggesting the best applicable algorithm for a given scenario.

2. Identify file level concerns in code files.
3. Code suggestions should be free from all recognized vulnerabilities and warn the user if a vulnerability is found.
4. Code suggestions should cover all the functional requirements of the given programming task.
5. AI-supported code completion tools should be able to suggest code with appropriate tests and Continuous Integration (CI) when applicable.
6. Code suggestions should follow user-specified coding style guidelines.
7. Satisfy requirements of all previous levels of abstractions.

4.2.5.2 System level design

System level design is the second half of the design level in our taxonomy. This level is the highest abstraction level with the highest human oversight and the most complex to define rules. AI-supported code completion tools at this level can suggest design decisions at the project level, like suggesting design patterns and architectural tactics with minimal input from the user.

This level requires the suggested code to suggest rational design practices in its code suggestions for a problem and satisfy all the previous levels of abstractions. Design practices depend on many factors like requirements and technical debt. AI-supported code completion tools should be capable of considering all the relevant factors before suggesting a design practice and providing the reasoning for each choice to the user.

The main goal of this level in the taxonomy is for a AI-supported code completion tools to help the user in every part of the software development process with minimal input from the user.

The capabilities required by a AI-supported code completion tools to satisfy this level of abstraction are as follows

1. Identify system level concerns in code files.
2. Suggest design patterns and architectural tactics when prompted.
3. Code suggestions should cover all the project's non-functional requirements.

4. AI-supported code completion tools should be able to identify the coding style followed and adapt its code suggestions.
5. AI-supported code completion tools should be able to make design decisions based on requirements and inform the user about those decisions.
6. Satisfy requirements of all previous levels of abstractions.

To make a AI-supported code completion tools suggest design decisions is a very challenging task. Software design is very subjective, and software design concerns are still challenging to comprehend. This is because software design is one of the least concrete parts of the software development lifecycle, especially compared to testing, implementation, and deployment [7]. Software design is typically carried out heuristically by drawing on the design team’s knowledge, the project context (such as architecturally significant needs), and a constantly evolving set of patterns and styles from the literature. We discuss more on these challenges in section 4.3.

4.3 AI-supported Software Development

We began this thesis by analyzing Copilot code suggestions on Pythonic idioms and best practices in JavaScript to understand the current boundaries of AI-supported code completion tools like Copilot using a software abstraction taxonomy. In this section, we try to address **RQ-2** (Given the current boundary, how far is it from suggesting design decisions?) with a discussion on the complex nature of design decisions involving factors ranging from requirement analysis to maintenance, making it difficult for AI-supported code completion tools like Copilot to detect the information from code files and suggest design decisions to satisfy the top software abstraction level of our taxonomy. Additionally, we discuss our vision for AI-supported code completion tools like Copilot to satisfy the design level in our taxonomy and outline the difficulty its underlying Codex LLM approach might run into. Finally, we discuss how design choices change over time and outline the difficulties of AI-supported code completion tools like Copilot to keep updating its suggestions and reflect the current design practices (section 4.3.1).

Software development is a challenging, complex activity: It is common for tasks to be unique and to call for the analysis of ideas from other domains. Solutions must be inventively modified to address the requirements of many stakeholders. Software

design is a crucial component of the software development activity since it determines the various aspects of the system, such as its performance, maintainability, robustness, security, etc. Design is typically viewed in the context of software engineering as both a process [25] that a development team engages in and the specifications [62] that the team produces. Software design is typically carried out heuristically by pulling from the project context (such as architecturally significant needs), the design team’s knowledge, and a constantly evolving set of patterns and styles from the literature.

Automating this software design process, which is the most abstract element in the software development lifecycle, will be challenging. First, sufficient software design knowledge has to be collected to use as training data to create good AI-supported code completion tools that can suggest relevant architectural patterns. Software design generally occurs in various developer communication channels such as issues, pull requests, code reviews, mailing lists, and chat messages for multiple purposes such as identifying latent design decisions, design challenges, design quality, etc. Gathering all this data and generalizing those design decisions in training data to suggest relevant design choices to a user would be the vision for AI-supported code completion tools to satisfy the design level.

Stack Overflow², the most popular question and answer (Q&A) forum used by developers for their software development queries [6]. Software developers of all levels of experience conduct debates and deliberations in the form of questions, answers, and comments on Stack Overflow’s extensive collection of topics about software development. Due to these qualities, Stack Overflow is a top choice for software developers looking to crowdsource design discussions and judgments, making it a good source of training data for AI-supported code completion tools for design choices.

Additionally, AI-supported code completion tools at the design level should be capable of capturing design and module level concerns. These include capturing design patterns (such as Observer) and architectural tactics (such as Heartbeat) to improve and personalize suggestions. The general understanding of a system’s design that a software developer has is frequently susceptible to “evaporation,” which causes the developers to gradually lose knowledge of the design over time [65] making the process of gathering design data to train AI-supported code completion tools a significant challenge.

Organizing software design information is an active research area. Previously, this design knowledge was organized largely manually because the information was heavily

²<https://stackoverflow.com/>

context-specific and a lack of large datasets. A study by Gorton et al. [28] showed a semi-automatic approach to populate design knowledge from internet sources for a particular (big data) domain, which can be a helpful approach for collecting software design relevant data to train AI-supported code completion tools.

Over the natural evolution of a software system, small changes accumulate, which can happen for various reasons, such as refactoring [55], bug fixes [15], implementation of new features, etc. These changes can be unique. However, they frequently repeat themselves and follow patterns [53]. Such patterns can provide a wealth of data for studying the history of modifications and their effects [39], modification histories of fault fixes [26], or the connections between code change patterns and adaptive maintenance [49]. However, to use this data, AI-supported code completion tools should be able to identify these complex patterns existing in public code (training data). Current AI-supported code completion tools like Copilot struggled to detect much simpler patterns like Pythonic idioms. There is no evidence currently to suggest they can identify even more complex design patterns.

Additionally, current AI-supported code completion tools like Copilot do not support multi-file input. It is not possible to evaluate its current performance in design suggestions, as the software development process may include multiple folders with a file structure. For example, MVC pattern generally includes multiple files acting as Model, View, and Controller. Using the current limitations of input on Copilot, i.e., a code block or a code file, it is not possible for AI-supported code completion tools to deduce that the project is using the MVC pattern and adapt its suggestion to follow the MVC pattern and not suggest code where Model communicated directly with View. AI-supported code completion tools must be capable of making suggestions in multiple program units to accommodate these more abstract design patterns.

AI-supported code completion tools should be able to adapt their suggestions to context-specific issues such as variable naming conventions and formatting. This would be challenging as the existing guidelines are not standard in this space and mostly depend on context.

4.3.1 Evolution of design over time

Software design is an ever-changing field that evolves along with technology, languages, and frameworks. As a result, either new design patterns are developed, or some existing ones are depreciated. AI-supported code completion tools need to up-

date their code suggestions regularly to reflect the changes in design practices. This requires regularly updating the training data, and training costs are expensive.

Design patterns are solutions to recurring design issues that aim to improve reuse, code quality, and maintainability. Design patterns have benefits such as decoupling a request from particular operations (Chain of Responsibility and Command), making a system independent from software and hardware platforms (Abstract Factory and Bridge), and independent from algorithmic solutions (Iterator, Strategy, Visitor), or preventing implementation modifications (Adapter, Decorator, Visitor). These design patterns are integral to software design and are used regularly in software development. However, these design patterns evolve. For instance, with React Framework's introduction, many new design patterns were introduced, such as Redux and Flux, which were considered to be an evolution over the pre-existing MVC design pattern. AI-supported code completion tools trained before this evolution will not have any data of the new design patterns such as Redux and Flux, making them incapable of suggesting those design patterns to the user.

Similarly, coding practices evolve. For example, in JavaScript, callbacks were considered the best practice in the past to achieve concurrency, which was replaced by promises. When the user has a goal to achieve asynchronous code, there are two ways to create async code: callbacks and promises. Callback allows us to provide a callback to a function, which is called after completion. With promises, you can attach callbacks to the returned promise object. One common issue with using the callback approach is that when we have to perform multiple asynchronous operations at a time, we can easily end up with something known as “callback hell”. As the name suggests, it is harder to read, manage, and debug. The simplest way to handle asynchronous operations is through promises. In comparison to callbacks, they can easily manage many asynchronous activities and offer better error handling. This makes AI-supported code completion tools be updated regularly to reflect new changes in coding practices and design processes of software development.

Additionally, Bad Practices in using promises for asynchronous JavaScript like not returning promises after creation and forgetting to terminate chains without a catch statement, which are explained in documentation³ and StackOverflow⁴ are not known to Copilot and suggested code with those common anti-patterns as they could have occurred more frequently in Copilot training data. While testing, Copilot sug-

³https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises

⁴<https://stackoverflow.com/questions/30362733/handling-errors-in-promise-all/>

gested code specifically mentioned in the JavaScript documentation as a common bad practice and anti-pattern³. However, this is beyond the scope of this study and will be part of future work.

In conclusion, Software design is an abstract field of software development, where humans struggle to make correct design decisions using all their previous experience and various sources of information to satisfy the requirements of a system. Creating AI-supported code completion tools to automate the software design process requires gathering relevant training data and regular updates to the training data to reflect the new changes in the evolution of the software development process. Further, the current Copilot approach of token-level suggestions needs to be upgraded to move beyond tokens (shown in 5.3.2.1) to facilitate multi-file input to help make AI-supported code completion tools capable of satisfying the design level of our taxonomy. So, current AI-supported code completion tools are far from satisfying design level of software abstraction and require further research in gathering training data, having multi-file input and making regular updates to reflect new changes in the evolution of the software development process.

4.4 Chapter Summary

In this chapter, we introduced the taxonomy of software abstraction hierarchy. We explain each software abstraction level in the taxonomy and the capabilities required by AI-supported code completion tools to satisfy each software abstraction level. Additionally our analysis of Copilot performance in suggesting Pythonic idioms and best practices in JavaScript led us to create a metric for all AI-supported code completion tools to help answer **RQ-1** (What are the current boundaries of code completion tools). We found that Copilot satisfies syntax and correctness levels in our taxonomy.

Finally, we addressed **RQ-2** (Given the current boundary, how far is it from suggesting design decisions?). We began with a discussion on the existing capabilities of Copilot and the desired capabilities of AI-supported code completion tools at the design abstraction level. We then observed that software design evolves over time and AI-supported code completion tools needs to be updated often to keep its code suggestions up-to-date.

Identifying the limitations of current AI-supported code completion tools like Copilot could help in resolving the issues of found on Copilot and also help in the development of new tools that can satisfy design level of our taxonomy and help users

make better software. We will discuss this and other potential solutions in more detail in the next chapter (Chapter 5).

Chapter 5

Discussion, Future Work and Conclusion

5.1 Introduction

We began this thesis with an analysis of Copilot code suggestions on Pythonic idioms, and Javascript best practices to understand the current boundaries of AI-supported code completion tools like Copilot using a software abstraction taxonomy. In this chapter, we first begin by extending this discussion by comparing Copilot performance on Pythonic idioms and JavaScript best practices. In section 5.2.1, we discuss the differences in the performance and ranking of Copilot code suggestions on Pythonic idoms and JavaScript best practices. We also discuss how Copilot was able to suggest idiomatic code for some coding scenarios.

Furthermore, having established the software abstraction hierarchy to help assess the capabilities of AI-supported code completion tools, in section 5.2.2, we discuss what it means to recite code from training data of AI-supported code completion tools like Copilot. Additionally, we discuss how code recitation is an ideal behavior for AI-supported code completion tools like Copilot to suggest idiomatic code but not for code smells.

In this second part of this thesis, we discussed our taxonomy of software abstractions and the challenges involved in creating AI-supported code completion tools that are capable of satisfying the design level of our taxonomy. In section 5.3, we report on some implications for researchers and practitioners. Finally, in section 5.4, we report on the threats to the validity of the research presented in this thesis.

5.2 Discussion

In this section, we discuss the differences in the performance and ranking of Copilot code suggestions on Pythonic idioms and JavaScript best practices. We then discuss what is code recitation and how it affects AI-supported code completion tools like Copilot.

5.2.1 Copilot code suggestions

Copilot code suggestions for our coding scenarios in Pythonic idioms and JavaScript best practices (shown in Chapter 3) has three possible outcomes: Copilot suggesting the recommended approach as its top suggestion (ideal behavior), Copilot having the recommended approach in its top 10 code suggestions but not the top suggestion (less ideal) and Copilot does not have the recommended approach in any of its suggestions (worst case).

Copilot recommending the idiomatic approach as its *top suggestion* is the ideal behavior for all AI-supported code completion tools. This may require the recommended approach to be the most popular method to solve the programming task. For example, Copilot successfully suggested the idiomatic approach as its top suggestion for idioms 7 (set comprehension) and 10 (if condition check value). Both those programming tasks have the idiomatic approach as one of the common approaches of solving the problem. Similarly, for code smells, Copilot successfully suggested the JavaScript best practice as its top suggestion for best practices 7 (accessing properties), 15 (converting array-like object to array), and 23 (check boolean value). These programming tasks also have the best practice approach as a common approach for solving the programming task. To make AI-supported code completion tools like Copilot suggest recommended approach as the top code suggestion, the approach must be the most common way of solving the given programming task.

Copilot recommending the idiomatic approach in its *top 10 suggestions* is not the ideal behavior but better than not having the idiomatic approach in its suggestions at all. Copilot had 8 idiomatic approaches and 5 best practices in its top 10 suggestions. This case is a result of AI-supported code completion tools having the recommended approach in its training data, but the ranking metrics such as the popularity of the approach made the idiomatic approach rank below the non-idiomatic approach. To resolve this case, AI-supported code completion tools like Copilot should update their ranking approach to have multiple metrics such as repository popularity or

acceptance of the approach in online forums such as StackOverflow to make the idiomatic approach rank higher than the non-idiomatic approaches.

When the idiomatic approach is *not in any of its top 10 suggestions*, AI-supported code completion tools like Copilot does not have the recommended approach in its training data, making it unaware of the approach or the recommended approach is very rare that it didn't even make it to top 10 suggestions, this is the worst case of all three outcomes of our coding scenarios with Copilot. This also suggests that people do not use the recommended approach in their code, and efforts should be made to improve awareness of such recommended approaches to solve those common programming tasks.

5.2.2 Copilot code recitation

AI-supported code completion tools like Copilot are trained on billions of lines of code. The code suggestions Copilot makes to a user are adapted to the specific coding scenario, but the processing behind each code suggestion is ultimately taken from training (public) code. A recent study conducted by Bender et al. [8] showed that LLMs like GPT-3 [10] and Codex [12] could recite code suggestions identical to the code present in training data, which in turn can cause issues like license infringements [14].

Traditional N-gram LMs like SLANG [64] and CACHECA [24] (shown in section 2.2.1.1) can only model relatively local dependencies, predicting each word given the preceding sequence of N words (usually 5 or fewer). However, more advanced models like the Transformer LMs used in Codex [12] capture much larger windows and can produce code that is seemingly not only fluent but also coherent even over large code blocks. We were able to generate a whole 980 lines of a code file using Copilot with an input of the first 15 lines of that file¹.

This code recitation behavior of LLMs like Codex [12] can help with satisfying paradigms and idioms level of our taxonomy. Idioms are the most accepted approaches to solve a programming task. The ideal behavior of AI-supported code completion tools like Copilot is to recognize these idiomatic approaches to solve a programming task from training data and use them in code suggestions.

Similarly, Copilot can also recite common code smells in public (training) data. AI-supported code completion tools like Copilot needs to recognize the difference

¹all experiments are documented in replication package [60]

between an idiomatic approach and a code smell in training data. Metrics like code repository popularity and StackOverflow upvotes on the code can help AI-supported code completion tools like Copilot to distinguish between idiomatic approaches and code smells.

5.3 Implications

This research helps guide future AI-supported code completion tools to support software development. Good AI-supported code completion tools has many potential uses, from recommending expanded code completions to optimizing code blocks. Automating code production could increase the productivity of current programmers.

Future code generation models may enable developers to work at a higher degree of abstraction that hides specifics, similar to how contemporary software engineers no longer frequently write in assembly. Good AI-supported code completion tools may improve accessibility to programming or aid in training new programmers. Models could make suggestions for different, more effective, or idiomatic methods to implement programs, enabling one to develop their coding style.

5.3.1 Implications for practitioners

5.3.1.1 Pre-training the LLM

For pre-training the LLM (e.g., Codex), AI-supported software development tools will need higher-quality training data. This might be addressed by carefully engineering training examples and filtering out known flaws, code smells, and bad practices. Careful data curation seems to be part of the approach already [45]. However, there is little clarity on how this process happens and how to evaluate suggestions, particularly for non-experts. One approach is to add more verified sources like well-known books and code documentation pages to follow the best practices. Pre-training might rank repositories for training input according to code quality (e.g., only repositories with acceptable coding standards).

5.3.1.2 Code completion time

AI-supported software development tools could collaborate with, or be used in conjunction with, existing tools for code smells like SonarQube² or other code review bots to potentially improve the quality of suggestions. Since developers expect to wait for a code suggestion, the results could be filtered for quality. Amazon’s code completion tool ‘CodeWhisperer’ comes with a ‘run security scan’ option, which performs a security scan on the project or file that is currently active in VS Code [4]. Active learning approaches which learn a user’s context (e.g., the company coding style) would also improve suggestion acceptability.

5.3.1.3 Over-reliance

Over-reliance on generated outputs is one of the main hazards connected to the use of code generation models in practice. Codex may provide solutions that seem reasonable on the surface but do not truly accomplish what the user had in mind. Depending on the situation, this could negatively impact inexperienced programmers and have serious safety consequences. Human oversight and vigilance are always required to safely use AI-supported code completion tools like Copilot. Empirical research is required to consistently ensure alertness in practice across various user experience levels, UI designs, and tasks.

5.3.1.4 Ethical Considerations

Packages or programs created by third parties are frequently imported within a code file. Software engineers rely on functions, libraries, and APIs for the majority of what called as “boilerplate” code rather than constantly recreating the wheel. However, there are numerous choices for each task: For machine learning, use PyTorch or TensorFlow; for data visualization, use Matplotlib or Seaborn; etc.

Reliance on import suggestions from AI-supported code completion tools like Copilot may increase as they get used to using AI-supported code completion tools. Users may employ the model as a decision-making tool or search engine as they get more adept at “prompt engineering” with Codex. Instead of searching the Internet for information on “which machine learning package to employ” or “the advantages and disadvantages of PyTorch vs. Tensorflow,” a user may now type “# import machine

²<https://www.sonarqube.org>

learning package” and rely on Codex to handle the rest. Based on trends in its training data, Codex imports substitutable packages at varying rates [12], which may have various effects. Different import rates set by Codex may result in subtle mistakes when a particular import is not advised, increased robustness when an individual’s alternative package would have been worse, and/or an increase in the dominance of an already powerful group of people and institutions in the software supply chain.

As a result, certain players may solidify their position in the package market, and Codex may be unaware of any new packages created following the first collection of training data. The model may also recommend deprecated techniques for packages that are already in use. Additional research is required to fully understand the effects of code creation capabilities and effective solutions.

5.3.2 Implications for researchers

With a wide range of applications, including programming accessibility, developer tools, and computer science education, effective code generation models have the potential to have a positive, revolutionary effect on society. However, like most technologies, these models may enable applications with societal drawbacks that we need to watch out for, and the desire to make a positive difference does not, in and of itself, serve as a defense against harm. One challenge researchers should consider is that as capabilities improve, it may become increasingly difficult to guard against “automation bias.”

5.3.2.1 Moving Beyond Tokens

Another research challenge is to move beyond token-level suggestions and work at the code block or file level (e.g., a method or module). Increasing the model input size to span multiple files and folders would improve suggestions. For example, when there are multiple files implementing the MVC pattern, Copilot should never suggest code where **Model** communicates directly with **View**. AI-supported software development tools will need to make suggestions in multiple program units to accommodate these more abstract design concerns.

One suggestion is to use recent ML advances in helping language models ‘reason’, such as the chain of thought process by Wang et al. [77]. Chain-of-thought shows the model and example of reasoning, allowing the model to reproduce the reasoning pattern on a different input. Such reasoning is common for design questions. Shokri [69]

explored this with framework sketches.

For example, using architectural scenarios helps (humans) reason about which tactic is most suitable for the scenario [40]. This is a version of the chain of thought for designing systems. However, we have an imperfect understanding of the strategies that drive human design approaches for software [5].

5.3.2.2 Explainability

Copilot is closed source, and it is currently not possible to determine the source or the reason behind each suggestion, making it difficult to detect any problems (access is only via an API). However, engineering software systems are laden with ethical challenges, and understanding why a suggestion was made, particularly for architectural questions such as system fairness, is essential. Probes, as introduced in [38], might expand technical insight into the models.

Another challenge is understanding the basis for the ranking metric for different suggestions made by Copilot. This metric has not been made public. Thus, we cannot determine why Copilot ranks one approach (e.g., non-idiomatic) over the idiomatic (preferred) approach. However, large language model code suggestions are based on its training data [11], so one explanation is that the non-idiomatic approach is more frequent in the training data [8]. Better characterization of the rankings would allow users to better understand the motivation.

5.3.2.3 Control

Being generative models, tools like Copilot are extremely sensitive to input with stability challenges, and to make them autonomous raises control concerns. For example, if a human asks for a N^2 sorting algorithm, should Copilot recommend one or the $N\log N$ alternative? Ideally, tools should warn users if prompted to suggest sub-optimal code. AI-supported software development should learn to differentiate between optimal and sub-optimal code. One direction to look at is following commit histories of files, as they are the possible places to find bug fixes and performance improvements.

5.4 Threats to Validity

Copilot and its underlying OpenAI Codex LLM are not open sources. We base our conclusions on API results, which complicate versioning and clarity. There are several threats to the validity of the work presented in this thesis. In this section, we summarize the dangers and also present the steps taken to mitigate them. We use the stable Copilot extension release (version: 1.30.6165) in Visual Studio Code.

5.4.1 Internal Validity

Copilot is sensitive to user inputs, which hurts replicability as a different formulation of the problem might produce a different set of suggestions. Because Copilot uses Codex, a generative model, its outputs cannot be precisely duplicated. Copilot can produce various responses for the same request as Copilot is a closed-source, black-box application that runs on a distant server and is therefore inaccessible to general users (such as the author of this thesis). Thus a reasonable concern is that our (human) input is unfair to Copilot, and with some different inputs, the tool might generate the correct idiom. For replicability, we archived all examples in our replication package at replication package [60].

5.4.2 Construct Validity

The taxonomy of the software abstraction hierarchy presented in this thesis relies on our view of software abstractions. Other approaches for classifying software abstractions (such as the user's motivation for initiating AI-supported code completion tools) might result in different taxonomy. The hierarchy of software abstractions presented in this thesis relies on our understanding of software abstractions, and the results of Copilot code suggestions on language idioms and code smells. Further, we present our results using Python and JavaScript. It is possible that using some other programming language or AI-supported code completion tools might have different results.

We intended to show where Copilot cannot consistently generate the preferred answer. We biased our evaluation to highlight this by choosing input that simulates what a less experienced programmer might enter. But we argue this is reasonable: for one, these are precisely the developers likely to use Copilot suggestions and unlikely to know the idiomatic usage. More importantly, a lack of suggestion stability seems

to come with its own set of challenges, which are equally important to understand.

5.4.3 Bias, Fairness, and Representation

Code generation models are susceptible to repeating the flaws and biases of their training data, just like natural language models [10]. These models can reinforce and maintain societal stereotypes when trained on a variety of corpora of human data, having a disproportionately negative effect on underprivileged communities. Additionally, bias may result in outdated APIs or low-quality code that reproduces problems, compromising performance and security. This might result in fewer people using new programming languages or libraries.

Codex has the ability to produce code that incorporates stereotypes regarding gender, ethnicity, emotion, class, name structure, and other traits [12]. This issue could have serious safety implications, further motivating us to prevent over-reliance, especially in the context of users who might over-rely on Codex or utilise it without properly thinking through project design.

5.5 Conclusion

Chapter 3 has shown the current challenges of AI-supported code completion tools like security issues and license infringements. We also showed that AI-supported code completion tools like Copilot struggle to use Pythonic idioms and JavaScript best practices in its code suggestions.

Chapter 4 represents a continuation of our previous work in chapter 3 by introducing a taxonomy of software abstraction hierarchy to delineate the limitations of AI-supported code completion tools like Copilot. We also show that Copilot stands at correctness level of our taxonomy. Finally, we discussed how AI-supported code completion tools like Copilot can reach the highest level of software abstraction in our taxonomy (design level).

The possible applications of large LLMs like Codex are numerous. For instance, it might ease users' transition to new codebases, reduce the need for context switching for seasoned programmers, let non-programmers submit specifications, have Codex draught implementations, and support research and education.

GitHub's Copilot and related large language model approaches to code completion are promising steps in AI-supported software development. However, Software

systems need more than coding effort. These systems require complex design and engineering work to build. We showed that while the coding syntax and correctness level of software problems is well on their way to useful support from AI-supported code completion tools like Copilot, the more abstract concerns, such as code smells, language idioms, and design rules, are far from solvable at present. Although far off, we believe AI-supported software development, where an AI supports designers and developers in more complex software *development* tasks, is possible.

Bibliography

- [1] Airbnb. Airbnb javascript style guide, 2012. URL: <https://github.com/airbnb/javascript>.
- [2] Marty Alchin. *Pro Python*. Apress, 2010. doi:10.1007/978-1-4302-2758-8.
- [3] Carol V. Alexandru, José J. Merchante, Sebastiano Panichella, Sebastian Proksch, Harald C. Gall, and Gregorio Robles. On the usage of pythonic idioms. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward!* 2018, page 1–11, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3276954.3276960.
- [4] Amazon. Amazon codewhisperer, 2022. URL: <https://aws.amazon.com/codewhisperer/>.
- [5] Maryam Arab, Thomas D. LaToza, Jenny Liang, and Amy J. Ko. An exploratory study of sharing strategic programming knowledge. In *CHI Conference on Human Factors in Computing Systems*, New York, NY, USA, April 2022. ACM. doi:10.1145/3491102.3502070.
- [6] Sebastian Baltes, Lorik Dumani, Christoph Treude, and Stephan Diehl. SO-Torrent: reconstructing and analyzing the evolution of stack overflow posts. In *Proceedings of International Conference on Mining Software Repositories*, pages 319–330. ACM, 2018. doi:10.1145/3196398.3196430.
- [7] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. SEI series in software engineering. Addison-Wesley, 2003. URL: <https://books.google.ca/books?id=mdiIu8Kk1WMC>.

- [8] Emily M. Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. On the dangers of stochastic parrots: Can language models be too big? In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, FAccT '21, page 610–623, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3442188.3445922.
- [9] Yoshua Bengio, Réjean Ducharme, and Pascal Vincent. A neural probabilistic language model. In T. Leen, T. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems*, volume 13. MIT Press, 2000. URL: <https://proceedings.neurips.cc/paper/2000/file/728f206c2a01bf572b5940d7d9a8fa4c-Paper.pdf>.
- [10] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020. URL: <https://proceedings.neurips.cc/paper/2020/file/1457c0d6bfc4967418bfb8ac142f64a-Paper.pdf>.
- [11] Nicholas Carlini, Florian Tramèr, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, Alina Oprea, and Colin Raffel. Extracting training data from large language models, 2020. URL: <https://arxiv.org/abs/2012.07805>, doi:10.48550/ARXIV.2012.07805.
- [12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen

- Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL: <https://arxiv.org/abs/2107.03374>, doi:10.48550/ARXIV.2107.03374.
- [13] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. An empirical study on the usage of BERT models for code completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 108–119. IEEE, May 2021. doi:10.1109/msr52588.2021.00024.
- [14] Matteo Ciniselli, Luca Pascarella, and Gabriele Bavota. To what extent do deep learning-based code recommenders generate predictions by cloning code from the training set?, 2022. URL: <https://arxiv.org/abs/2204.06894>, doi:10.48550/ARXIV.2204.06894.
- [15] Domenico Cotroneo, Luigi De Simone, Antonio Ken Iannillo, Roberto Natella, Stefano Rosiello, and Nematollah Bidokhti. Analyzing the context of bug-fixing changes in the openstack cloud computing platform, 2019. URL: <https://arxiv.org/abs/1908.11297>, doi:10.48550/ARXIV.1908.11297.
- [16] F. DeRemer and H.H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, SE-2(2):80–86, June 1976. doi:10.1109/tse.1976.233534.
- [17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. URL: <https://aclanthology.org/N19-1423>, doi:10.18653/v1/N19-1423.
- [18] Eclipse-Foundation. Eclipse ide, 2022. URL: <https://www.eclipse.org/ide/>.

- [19] Neil A. Ernst, Stephany Bellomo, Ipek Ozkaya, and Robert L. Nord. What to fix? distinguishing between design and non-design rules in automated tools. In *2017 IEEE International Conference on Software Architecture (ICSA)*. IEEE, April 2017. doi:10.1109/icsa.2017.25.
- [20] Kawin Ethayarajh. How contextual are contextualized word representations? Comparing the geometry of BERT, ELMo, and GPT-2 embeddings. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 55–65, Hong Kong, China, November 2019. Association for Computational Linguistics. URL: <https://aclanthology.org/D19-1006>, doi:10.18653/v1/D19-1006.
- [21] Aamir Farooq and Vadim Zaytsev. There is more than one way to zen your python. In *Proceedings of the 14th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2021*, page 68–82, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3486608.3486909.
- [22] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020. URL: <https://arxiv.org/abs/2002.08155>, doi:10.48550/ARXIV.2002.08155.
- [23] Society for Automotive Engineers. Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles, 2021. URL: https://www.sae.org/standards/content/j3016_202104/.
- [24] Christine Franks, Zhaopeng Tu, Premkumar Devanbu, and Vincent Hellen-doorn. Cacheca: A cache language model based code suggestion tool. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 705–708, 2015. doi:10.1109/ICSE.2015.228.
- [25] Peter Freeman and David Hart. A science of design for software-intensive systems. *Commun. ACM*, 47(8):19–21, aug 2004. doi:10.1145/1012037.1012054.
- [26] Daniel M. German. An empirical study of fine-grained software modifications. *Empirical Softw. Engg.*, 11(3):369–393, sep 2006. doi:10.1007/s10664-006-9004-6.

- [27] GitHub. Github copilot, 2021. URL: <https://copilot.github.com>.
- [28] Ian Gorton, John Klein, and Albert Nurgaliev. Architecture knowledge for evaluating scalable databases. In *2015 12th Working IEEE/IFIP Conference on Software Architecture*, pages 95–104, 2015. doi:10.1109/WICSA.2015.26.
- [29] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 933–944, 2018. doi:10.1145/3180155.3180167.
- [30] Raymond Hettinger. Transforming code into beautiful, idiomatic python. URL: <https://www.youtube.com/watch?v=OSGv2VnCOgo>.
- [31] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering, ICSE '12*, page 837–847. IEEE Press, 2012.
- [32] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering, ICSE '22*, page 1219–1231, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3510003.3510203.
- [33] M. Jaworski and T. Ziadé. *Expert Python Programming: Master Python by learning the best coding practices and advanced programming concepts, 4th Edition*. Packt Publishing, 2021. URL: <https://books.google.ca/books?id=2tAwEAAAQBAJ>.
- [34] Jedi. Jedi, 2022. URL: <https://github.com/davidhalter/jedi>.
- [35] Rafael-Michael Karampatsis and Charles Sutton. Maybe deep neural networks are the best choice for modeling source code, 2019. URL: <https://arxiv.org/abs/1903.05734>, doi:10.48550/ARXIV.1903.05734.
- [36] Rafael-Michael Karampatsis and Charles Sutton. *How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset*, page 573–577. Association for Computing Machinery, New York, NY, USA, 2020. URL: <https://doi.org/10.1145/3379597.3387491>.

- [37] Anjan Karmakar and Romain Robbes. What do pre-trained code models know about code? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1332–1336, Los Alamitos, CA, USA, 2021. IEEE. doi:10.1109/ASE51524.2021.9678927.
- [38] Anjan Karmakar and Romain Robbes. What do pre-trained code models know about code? In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1332–1336, 2021. doi:10.1109/ASE51524.2021.9678927.
- [39] David Kawrykow and Martin P. Robillard. Non-essential changes in version histories. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE '11*, page 351–360, New York, NY, USA, 2011. Association for Computing Machinery. doi:10.1145/1985793.1985842.
- [40] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere. The architecture tradeoff analysis method. In *International Conference on Engineering of Complex Computer Systems*, pages 68–78. IEEE Comput. Soc, Aug 1998. doi:10.1109/ICECCS.1998.706657.
- [41] kite. Kite, 2022. URL: <https://www.kite.com>.
- [42] J. Knupp. *Writing Idiomatic Python 3.3*. Createspace Independent Pub, 2013. URL: https://books.google.ca/books?id=EtdF4NMi_NEC.
- [43] Philip Koopman. Maturity levels for autonomous vehicle safety, 2022. URL: <https://safeautonomy.blogspot.com/2022/04/maturity-levels-for-autonomous-vehicle.html>.
- [44] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. Code completion with neural attention and pointer networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence, IJCAI'18*, page 4159–25. AAAI Press, 2018.
- [45] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d’Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli,

- Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode, 2022. URL: <https://arxiv.org/abs/2203.07814>, doi:10.48550/ARXIV.2203.07814.
- [46] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: Helping to navigate the api jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05*, page 48–61, New York, NY, USA, 2005. Association for Computing Machinery. doi:10.1145/1065010.1065018.
- [47] A. H. Maslow. A theory of human motivation. *Psychological Review*, 50(4):370–396, July 1943. doi:10.1037/h0054346.
- [48] Tim Menzies, Andrew Butcher, David Cok, Andrian Marcus, Lucas Layman, Forrest Shull, Burak Turhan, and Thomas Zimmermann. Local versus global lessons for defect prediction and effort estimation. *IEEE Transactions on Software Engineering*, 39(6):822–834, June 2013. doi:10.1109/tse.2012.83.
- [49] Omar Meqdadi and Shadi Aljawarneh. A study of code change patterns for adaptive maintenance with ast analysis. *International Journal of Electrical and Computer Engineering (IJECE)*, 10:2719, 06 2020. doi:10.11591/ijece.v10i3.pp2719-2733.
- [50] Tomáš Mikolov, Anoop Deoras, Daniel Povey, Lukáš Burget, and Jan Černocký. Strategies for training large scale neural network language models. In *2011 IEEE Workshop on Automatic Speech Recognition and Understanding*, pages 196–201, 2011. doi:10.1109/ASRU.2011.6163930.
- [51] G.C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the eclipse ide? *IEEE Software*, 23(4):76–83, 2006. doi:10.1109/MS.2006.105.
- [52] Emerson Murphy-Hill, Ciera Jaspán, Caitlin Sadowski, David Shepherd, Michael Phillips, Collin Winter, Andrea Knight, Edward Smith, and Matthew Jorde. What predicts software developers’ productivity? *IEEE Transactions on Software Engineering*, 47(3):582–594, 2021. doi:10.1109/TSE.2019.2900308.
- [53] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N. Nguyen, and Hriday Rajan. A study of repetitiveness of code changes in software evolu-

- tion. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 180–190, 2013. doi:10.1109/ASE.2013.6693078.
- [54] Nhan Nguyen and Sarah Nadi. An empirical evaluation of GitHub Copilot’s code suggestions. In *Proceedings of the 19th ACM International Conference on Mining Software Repositories (MSR)*, pages 1–5, 2022.
- [55] Fabio Palomba, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. An exploratory study on the relationship between changes and refactoring. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 176–185, 2017. doi:10.1109/ICPC.2017.38.
- [56] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri. Asleep at the keyboard? assessing the security of github copilot’s code contributions. In *2022 IEEE Symposium on Security and Privacy (SP) (SP)*, pages 980–994, Los Alamitos, CA, USA, may 2022. IEEE Computer Society. URL: <https://doi.ieeecomputersociety.org/10.1109/SP46214.2022.00057>, doi:10.1109/SP46214.2022.00057.
- [57] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 2227–2237, New Orleans, Louisiana, June 2018. Association for Computational Linguistics. URL: <https://aclanthology.org/N18-1202>, doi:10.18653/v1/N18-1202.
- [58] Tim Peters. *The Zen of Python*, pages 301–302. Apress, Berkeley, CA, 2010. doi:10.1007/978-1-4302-2758-8_14.
- [59] Sebastian Proksch, Johannes Lerch, and Mira Mezini. Intelligent code completion with bayesian networks. *ACM Transactions on Software Engineering and Methodology*, 25(1):1–31, December 2015. doi:10.1145/2744200.
- [60] Rohith Pudari. Replication package, 2022. URL: <https://figshare.com/s/d39469927e2c620ec5d2>.
- [61] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners, 2019.

- [62] Paul Ralph and Yair Wand. A proposal for a formal definition of the design concept, 01 2009. doi:10.1007/978-3-540-92966-6_6.
- [63] L. Ramalho. *Fluent Python: Clear, Concise, and Effective Programming*. O’Reilly Media, 2015. URL: <https://books.google.co.in/books?id=bIZHCgAAQBAJ>.
- [64] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, June 2014. doi:10.1145/2594291.2594321.
- [65] Martin P. Robillard. Sustainable software design. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016*, page 920–923, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2950290.2983983.
- [66] R. Rosenfeld. Two decades of statistical language modeling: where do we go from here? *Proceedings of the IEEE*, 88(8):1270–1278, 2000. doi:10.1109/5.880083.
- [67] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany, August 2016. Association for Computational Linguistics. URL: <https://aclanthology.org/P16-1162>, doi:10.18653/v1/P16-1162.
- [68] Ali Shokri. A program synthesis approach for adding architectural tactics to an existing code base. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1388–1390, Los Alamitos, CA, USA, nov 2021. IEEE. doi:10.1109/ase51524.2021.9678705.
- [69] Ali Shokri. A program synthesis approach for adding architectural tactics to an existing code base. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1388–1390, 2021. doi:10.1109/ASE51524.2021.9678705.
- [70] Margaret-Anne Storey, Thomas Zimmermann, Christian Bird, Jacek Czerwonka, Brendan Murphy, and Eirini Kalliamvakou. Towards a theory of software devel-

- oper job satisfaction and perceived productivity. *IEEE Transactions on Software Engineering*, 47(10):2125–2142, 2021. doi:10.1109/TSE.2019.2944354.
- [71] Tabnine. Deep tabnine, 2022. URL: <https://www.tabnine.com>.
- [72] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 269–280, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2635868.2635875.
- [73] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts*, New York, NY, USA, April 2022. Association for Computing Machinery. doi:10.1145/3491101.3519665.
- [74] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL: <https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf>.
- [75] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks, 2015. URL: <https://arxiv.org/abs/1506.03134>, doi:10.48550/ARXIV.1506.03134.
- [76] Stefan Wagner and Melanie Ruhe. A systematic review of productivity factors in software development. *Arxiv*, 2018. URL: <https://arxiv.org/abs/1801.06475>, doi:10.48550/ARXIV.1801.06475.
- [77] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models, 2022. URL: <https://arxiv.org/abs/2203.11171>, doi:10.48550/ARXIV.2203.11171.

- [78] Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. Coacor: Code annotation for code retrieval with reinforcement learning. In *The World Wide Web Conference, WWW '19*, page 2203–2214, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3308558.3313632.