Analyzing Prompt Influence on Automated Method Generation: An Empirical Study with Copilot

Ionut Daniel Fagadau
University of Milano - Bicocca
Milan, Italy
i.fagadau@campus.unimib.it

Leonardo Mariani
University of Milano - Bicocca
Milan, Italy
leonardo.mariani@unimib.it

Daniela Micucci
University of Milano - Bicocca
Milan, Italy
daniela.micucci@unimib.it

Oliviero Riganelli
University of Milano - Bicocca
Milan, Italy
oliviero.riganelli@unimib.it

ABSTRACT

Generative AI is changing the way developers interact with software systems, providing services that can produce and deliver new content, crafted to satisfy the actual needs of developers. For instance, developers can ask for new code directly from within their IDEs by writing natural language prompts, and integrated services based on generative AI, such as Copilot, immediately respond to prompts by providing ready-to-use code snippets. Formulating the prompt appropriately, and incorporating the useful information while avoiding any information overload, can be an important factor in obtaining the right piece of code. The task of designing good prompts is known as prompt engineering.

In this paper, we systematically investigate the influence of eight prompt features on the style and the content of prompts, on the level of correctness, complexity, size, and similarity to the developers' code of the generated code. We specifically consider the task of using Copilot with 124,800 prompts obtained by systematically combining the eight considered prompt features to generate the implementation of 200 Java methods. Results show how some prompt features, such as the presence of examples and the summary of the purpose of the method, can significantly influence the quality of the result.

CCS CONCEPTS

• **Software and its engineering → Integrated and visual development environments**.

KEYWORDS

Prompt engineering, code generation, Copilot.

ACM Reference Format:
Ionut Daniel Fagadau, Leonardo Mariani, Daniela Micucci, and Oliviero Riganelli. 2024. Analyzing Prompt Influence on Automated Method Generation: An Empirical Study with Copilot. In *32nd IEEE/ACM International Conference on Program Comprehension (ICPC '24), April 15–16, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3643916.3644409

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

ACM ISBN 979-8-4007-0586-1/24/04.
https://doi.org/10.1145/3643916.3644409

1 INTRODUCTION

Generative AI solutions, and in particular those built on Large Language Models (LLM), such as ChatGPT [23], Bard [12], and CoPilot [9], promise to become a powerful tool that can aid software developers in completing their tasks more efficiently and effectively. For instance, LLMs have already been exploited to support programming tasks by automatically generating code that responds to a given natural language request formulated by a user [8, 14, 22]. Some other recent studies investigated how developers interact with these tools during project development [2], and how usable these tools are [2].

A key concern about LLMs is that they are known to be sensitive to the prompt, that is, the quality of the result strongly depends on the query that is asked by the user to the model [16, 17, 27]. In fact, the style of the prompt, as well as its content, may determine the level of correctness of the response. This is also confirmed by our observations in the domain of code generation. For instance, when asking for the body of the Java method `String convertToBase7(int num)` Copilot generates better code when the prompt includes some basic hints about the semantics of the intended solution, such as "*The base 7 digits are 0–6 and the digit positions represent powers of 7*". Or sometimes the resulting code might be better by just considering some simple stylistic rules, such as using the imperative mood (e.g., *… implement pattern matching that…*), which is often easier to be interpreted by LLMs compared to future (e.g., *… the method shall implement pattern matching that…*). As a consequence, researchers are actively studying how to interact with LLMs, so that the prompts that are likely to provide the best results can be quickly formulated and submitted to the models, without losing time with ineffective interactions.

Only a few studies preliminary investigated prompt engineering for code generation. Denny et al. studied the impact of changes implemented in prompts used for solving Python programming problems [6]. The results show that prompt engineering has been useful in the vast majority of the cases to improve the correctness of the resulting code. White et al. proposed patterns that can be used by programmers to write prompts in different situations, however without reporting data about their effectiveness [29]. Ren et al. investigated the specific case of adjusting the prompt to obtain

proper exception handling code [25]. So far, no study systematically considered prompt engineering in code generation tasks.

In this paper, we propose a first controlled experiment that investigates the effectiveness of eight prompt features on the same prompt for 200 Java code generation problems extracted from both GitHub [10] and LeetCode [13]. For example, we systematically compare the generated code when the same request is formulated according to different grammar styles (e.g., active and passive forms) or includes different content (e.g., with and without examples). We consider both GitHub and LeetCode to investigate the impact of the same prompt features when occurring within different kinds of prompts. Prompts derived from the GitHub methods' comments usually are concise explanations of a method's behavior, such as *The function inserts the given vertex into a sorted position in the given array*. On the contrary, prompts derived from LeetCode methods' descriptions usually are lengthy descriptions about the method to be implemented, such as *The function implements wildcard pattern matching … where '?' matches a single character … return true if any match is found. Example: …*.

In our study, we focus on the Copilot-3 LLM, since it is the most adopted AI developer tool, with more than 1 million developers and 20,000 organizations that adopted it [7]. We executed Copilot with a total of 124, 800 queries discovering important information about prompt engineering for code generation. In particular, we discover that including a summary of the purpose of the method and including examples in the prompt are particularly useful to obtain code that passes the available test cases. Some stylistic rules may also have an impact. For instance, the usage of the presence tense seems beneficial in prompts.

On the other hand, including some other information has not produced beneficial results. For instance, the inclusion of boundary cases and contextual information in the prompts had no significant effect on the level of correctness of the results. Including excessive information may even have a negative effect, as reported for boundary cases that made Copilot produce code that differs more from the code implemented by the developers when included in the prompt.

The experimental materials necessary to reproduce our study have been made available for online access at https://shorturl.at/hmpBM.

The paper is structured as follows. Section 2 introduces prompt engineering and describes the prompt features that we considered in our experimental study. Section 3 reports the three research questions that we investigated and describes the methodology that we followed to answer them. Section 4 presents the results that we obtained for each research question, discusses threats to validity, and distills some advice about the definition of the prompts. Section 5 discusses related work. Section 6 provides final remarks.

2 PROMPT ENGINEERING AND PROMPT FEATURES

Although powerful, LLMs may generate results of different quality depending on the style and content of the prompt. For this reason prompt engineering, that is, "the formal search for prompts that retrieve desired outcomes from language models" [16], is now an active field of research. For example, in computer vision, studies on prompt engineering revealed that focusing on the mood and style of the keywords that occur in the prompts is more important than rephrasing the prompts themselves [16].

In a nutshell, prompt engineering can be seen as a kind of programming in natural language, where programming statements are the natural language sentences in the prompts [26]. Understanding how to write good prompts is indeed also relevant when considering code generation tasks. To derive insights about how to write prompts for code generation, we systematically considered the same prompts written according to different styles and with variations in the content. The selection of the features derives from the analysis of the comments and prompts present in the GitHub and LeetCode methods we extracted. In particular, we considered three prompt features affecting the *style* of the sentences and five prompt features affecting the *content* of the prompt, for a total of eight prompt features.

Prompt features about the *style*

Mood: *(a) Indicative*, when the prompt uses the indicative mood to specify what the function has to do, for instance *Given an integer array …, the function returns true if there are two distinct indices …*; *(b) Imperative*, when the prompt uses the imperative mood of to specify what the function has to do, for instance *Given an integer array …, return true if there are two distinct indices …*;

Sentence Mode: *(a) Active*, when the prompt is in active mode, for instance *Given an integer array …, the function returns True if …*; *(b) Passive*, when the prompt is in passive mode, for instance *An integer array and … are given to the function. True is returned by the function if …*.

Tense: *(a) Present*, when the prompt uses the present tense, for instance *Given an integer array …, the function returns true if there are two distinct indices …*; *(b) Future*, when the prompt uses the future tense, for instance *Given an integer array …, the function will return true if there are two distinct indices …*

Prompt features about the *content*

Reference to the Parameters: *(a) Implicit without names*, when the prompt implicitly refers to the parameters of the functions without using their name, for instance *Return True if there are two distinct indices … in an integer array, such that …*; *(b) Implicit with names*, when the prompt implicitly refers to the parameters but uses their names, for instance *Return True if there are two distinct indices … in an integer array nums, such that …*; *(c) Explicit without names*: when the prompt explicitly refers to the parameters without using their names, for instance *Given an integer array and …, return True if there are two distinct …*; *(d) Explicit with names*: when the prompt explicitly refers to the parameters using their names, for instance *Given an integer array nums and …, return True if there are two distinct …*.

Boundary Cases: *(a) Missing*, when the prompt does not include any boundary case; *(b) Implicit*, when the boundary case is stated but it is not stated what the method should do, for instance *Given an integer array … return true if there are … and the array is not null*; *(c) Explicit*, when the boundary case is stated, including the expected behavior, for instance *Given an integer array … return true if there are …. If the array is null, return false.*

Summary of the Method: *(a) Missing*, when the semantic of the method is not summarized in the prompt, like in all the examples reported so far; *(b) Provided Upfront*, when the semantic is stated

at the beginning of the prompt, for instance *The result is a boolean representing if there are any duplicates in the array. Given an integer …*; *(c) Provided Afterward*, when the semantic of the method is summarized at the end of the prompt, for instance *Given an integer … The result is a boolean representing if there are any duplicates in the array*.

Examples: *(a) Present*, when the prompt includes examples about the input-output behavior of the method, for instance *Given an integer array … Example Input nums=[1, 2, 3, 1], Output = true*; *(b) Absent*, when the prompt does not include any example.

Context: *(a) Present*, when the prompt includes information about the characteristics of the inputs and the outputs, for instance *Given an integer array … Constraints 1 <= nums.length <= 100000, 0<=k<=100000*; *(b) Absent*, when the prompt does not include such information.

3 METHODOLOGY

The objective of our study is to investigate how prompt features may impact the correctness of the code generated by LLMs. In particular, we consider the generation of a method body as a task, and Copilot as LLM assisting this task. We selected this task because these tools are mainly designed and trained to help developers with the implementation of code snippets, such as individual methods and functions [2, 11]. We selected Copilot because it is the most popular and used AI-based code assistant at the moment [7].

Our study is structured around three main research questions:

RQ1 - How do prompt features impact on the *correctness* of the code generated by Copilot? This research question investigates how prompt features impact on the generation of code that compiles and passes test cases.

RQ2 - How do prompt features impact on the *complexity* of the code generated by Copilot? This research question investigates how prompt features impact on the cyclomatic complexity and size of the generated code.

RQ3 - How do prompt features impact on the *similarity* between the code generated by Copilot and the code implemented by the developers? This research question investigates how prompt features impact the possibility of obtaining code that is syntactically and semantically close to the code that the developers have implemented.

We describe below how we selected the prompts for answering RQ1-3, how we obtained the alternative prompts, consistently with the identified prompt features, how we collected the responses from Copilot, and finally how we analyzed the results.

3.1 Selection of the Prompts

Our experiment involves creating many different variants for the same prompts. We have thus to limit the initial set of prompts to start from to achieve a feasible total number of prompts to be analyzed. On the other hand, we want to study prompt engineering both in the context of short prompts mostly reflecting what developers already write as comments for their code, and in the context of richer prompts designed to be more explicative of the code that must be generated. For this reason, we decided to select 100 GitHub methods with comments, representing the case of shorter prompts derived from comments, and 100 LeetCode[1] methods, representing the case of longer and more explicative prompts.

To select the prompts from GitHub, we built on top of results obtained by Mastropaolo et al. [19] who collected prompts from the Javadoc comments present in the GitHub code that is part of Java projects with at least 300 commits, 50 contributors, and 25 stars. We selected prompts corresponding to methods with at least 75% of the statements exercised by the test case available in the repository of the project. From this set of prompts and methods, to make sure to consider a variety of prompts of different lengths, we randomly selected 40 prompts in the range 36-97 characters, 30 prompts in the range 98-159 characters, and 30 prompts in the range 160-221 characters[2].

To select the prompts from LeetCode, we exploited the classification of the problems present in the platform as easy, medium, and hard tasks. We thus randomly selected 40 easy problems, 30 medium problems, and 30 hard problems. Differently from GitHub, LeetCode provides the prompts but provides neither the expected solution nor the test suite to validate the code[3]. To obtain the solutions for the selected problems, we exploited the walkccc platform (https://walkccc.me/LeetCode) that stores solutions to LeetCode problems. Finally, to obtain the test cases, we automatically generated the tests from the solutions by running the EvoSuite test generator tool (https://www.evosuite.org) with its default configuration. We obtained a total of 426 test cases, an average number of 4.3 test cases per method. The average statement coverage achieved by these tests is 99.8%. Actually 97 methods are fully covered, and three methods have a lower coverage, with a minimum equal to 88%.

At the end of this step, we obtained 100 prompts derived from GitHub and 100 prompts derived from LeetCode. Each prompt is associated with a method signature, a method implementation, and a set of test cases. In all cases, we packaged the code as a Maven project to ease the automation of the next steps.

3.2 Generation of the Alternative Prompts

So far, the collected prompts are available in a single shape in terms of prompt features, while our objective is to systematically study the impact of different combinations of prompt features. To this end, we created every possible version of each prompt, according to the prompt features reported in Section 2.

In the case of the prompts extracted from GitHub, five prompt features can be modified systematically: the three prompt features about *style*, plus the *Reference to the Parameters* and the *Boundary Cases*. The prompt features *Summary of the Method*, *Examples*, and *Context* are always absent, and we decided to not add this information ourselves in the prompts to avoid any bias in the experiment. We, however, modified the prompts considering any possible combination of the applicable features obtaining 96 alternative versions of each prompt (2 *Mood* × 2 *Sentence Mode* × 2 *Tense* × 4 *Reference*

[1] LeetCode is an online platform for programming challenges, and to train developers to get ready for technical job interviews.

[2] 36 characters are the length of the shortest prompt in the dataset and 221 characters are the length of the longest prompt.

[3] LeetCode provides test suites to validate solutions, but the test suites are not publicly available and they can be used only by submitting solutions online, which is not doable for such a large study.

to the Parameters × 3 *Boundary Cases*), for a total of 9,600 prompts studied.

In the case of the prompts extracted from LeetCode, all prompt features can be modified systematically. This leads to 1,152 prompt variants (96 prompt variants as for GitHub prompts × 3 *Summary of the Method* × 2 *Examples* × 2 *Context*) for each prompt, for a total of 115,200 prompts studied.

We obtained these so many variants of the prompts with a combination of manual effort and automatic scripts. Changes to prompt features like *Mood*, *Sentence Mode*, and *Reference to the Parameters*, require manual intervention, since part of the sentence should be rewritten. These changes have been actuated first. The remaining prompt features could be addressed semi-automatically, that is, for each prompt that has to be modified, we implemented an ad-hoc script that makes substitutions and suppressions, finally obtaining the modified prompts.

Note that we always generated the variants of the features manually, and the scripts have been used to only systematically combine their values. For instance, the scripts can include the right instance of a feature in a sentence or selectively replace verbs. The overall set of prompts studied amounts to 124,800.

3.3 Collection of the Results

To complete our benchmark, we need to collect the code generated by Copilot for all the prompts we generated. To this end, we automatically generated a query package for each prompt in the dataset. The query package consists of a Java file with the prompt that has to be studied, followed by the signature of the method whose code has to be generated, and an empty method body implementation. For the cases derived from GitHub, we included in the file with the empty method the rest of the code of the class where the method occurs, so that Copilot could still exploit contextual information.

To automatically collect the results generated by Copilot, since an API is not available, we implemented an automation tool with PyAutoGUI (www.pyautogui.readthedocs.io), which is a Python library for automating I/O operations. To implement a quick and reliable tool, we interacted with Copilot within Visual Studio Code mainly using shortcuts. In particular, our tool opens Visual Studio Code directly on the file with the empty method from the command line, it then searches that method, moves the cursor over the target method, closes the search menu, enters into the empty body method, and invokes Copilot using shortcuts. Finally, it continuously takes screenshots until Copilot has produced a response, it then moves over the suggestion, accepts it, saves the files, and closes the editor.

The experimental material, including the full set of prompts and responses produced by Copilot, are available online at https://shorturl.at/hmpBM to facilitate further studies on this subject.

3.4 Analysis of the Results

To answer RQ1-3, we analyzed the code generated by Copilot according to multiple perspectives.

To answer RQ1, we checked if the generated code compiles and, in case it compiles, we determine if it also passes the execution of the available test cases. Although testing cannot guarantee the full correctness of the generated code, a code that passes a test suite that extensively exercises the code implemented by the developer is likely a quite useful piece of code. To determine if a prompt feature has a significant influence on the level of correctness of the code, we compute the contingency table and use the χ^2 test to check significance.

To answer RQ2, we compute the cyclomatic complexity and the number of lines of code in the generated code using JaSoMe (https://github.com/rodhilton/jasome). To determine if a prompt feature has a significant influence on the complexity and size of the generated code, we use the Wilcoxon-Mann-Whitney U test [20] for prompt features with two categorical values and the Kruskal-Wallis H-test [18] for the prompt features with three or more categorical values.

To answer RQ3, we compare the generated code and the original code written by the developers according to both the normalized Levenshtein distance and the CodeBLEU metric [24]. The Levenshtein distance captures the percentage of syntactic changes necessary to obtain the desired code from the code generated by Copilot (the higher, the more diverse the two code snippets are). The CodeBLEU metric is a more sophisticated metric that captures the degree of syntactic and semantic similarity between the compared codes, considering both their abstract syntax tree and their data-flow (the higher, the more similar they are). We used the javalang library (https://pypi.org/project/javalang/) to compute the normalized Levenshtein between tokens and the CodeXGLUE tool (https://github.com/microsoft/CodeXGLUE/tree/main) to compute the CodeBLEU metric. To determine if the prompt features have a significant influence on the Levenshtein distance and on the CodeBLEU metric of the generated code, we use the Wilcoxon-Mann-Whitney U test [20] for prompt features with two categorical values and the Kruskal-Wallis H-test [18] for the prompt features with three or more categorical values.

For all the statistical tests, we consider a significance level of 0.05, but we also mention when results are nearly significant, that is, they are significant at a significance level of 0.1.

4 RESULTS

In this section, we discuss the results obtained for the three research questions.

4.1 RQ1 - Level of Correctness

This research question investigates the correlation between the level of correctness of the code generated by Copilot and the content and style of the prompts, depending on the occurrences of the prompt features.

Table 1: Level of correctness of the generated code.

Subject Cases	Tot. Prompts	No Compile	Fail Test	Pass All Tests
GitHub	9,600	6,246 (65%)	1,772 (18%)	1,582 (17%)
LeetCode	115,200	86,847 (75%)	14,434 (13%)	13,919 (12%)
Total	124,800	93,093 (75%)	16,206(13%)	15,501(12%)

Table 1 reports the number and percentage of method bodies generated by Copilot that do not compile (Column *No Compile*), that fail at least a test case (Column *Fail Test*), and that pass all the available test cases (Column *Pass All Tests*). The code generated for

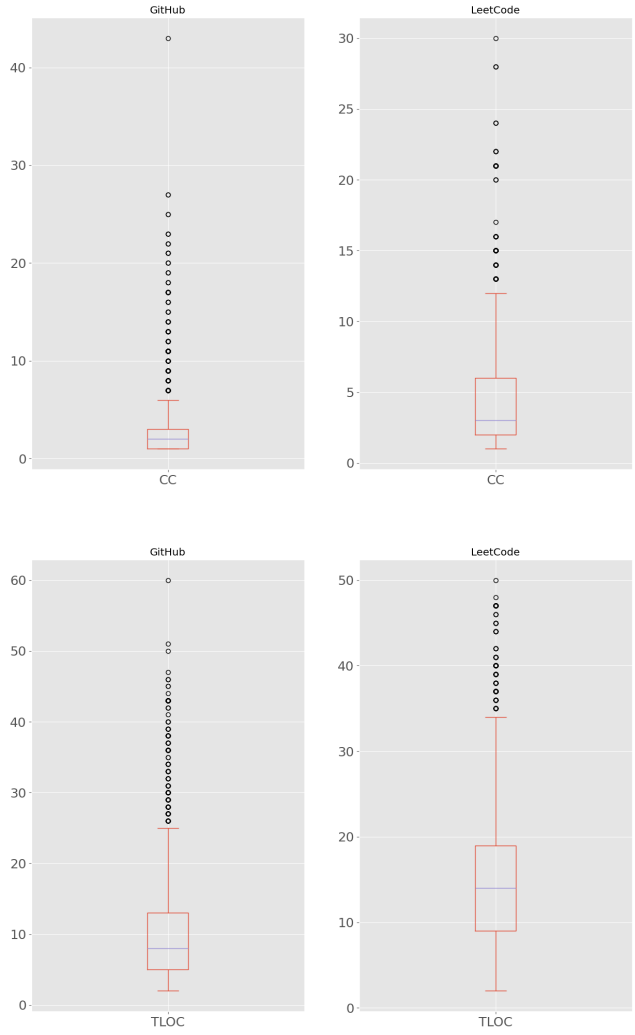

Figure 1: Complexity and size of methods in GitHub and LeetCode.

the GitHub methods compiled and passed test cases more frequently than the code generated for LeetCode methods: 35% of method bodies compiled and 17% of method bodies passed all the test cases for GitHub methods, while 25% of method bodies compiled and 12% passed all the test cases for LeetCode. This is likely due to the different characteristics of the problems present in LeetCode, compared to the code present in actual GitHub projects. Figure 1 shows the complexity and size of the GitHub and LeetCode methods in our benchmark. LeetCode methods tend to be more complex and longer than GitHub methods, likely explaining the difference in the results. GitHub methods capture the small-scale problems that developers face when writing code, while LeetCode methods capture some algorithmic problems that developers may have to face. Overall, the percentage of cases that pass all the available test cases is quite limited (12%), but consistent with previous results reported in studies about the effectiveness of Copilot. For instance, Mastropaolo et al. [19] obtained approximately 13% method bodies that pass the test cases with Copilot using both the original prompts and those generated, either manually or automatically, through specialized tools.

Table 2: P-values of prompt features significantly influencing the level of correctness of the generated code.

Prompt Features	GitHub		LeetCode	
	Compile	Pass Tests	Compile	Pass Tests
Mood	No	No	No	No
Mode	No	No	No	No
Tense	0.1	0.1	No	No
Param	No	No	No	No
B. Cases	No	No	No	No
Summary	-	-	No	**0.04**
Examples	-	-	**10^{-4}**	**10^{-4}**
Context	-	-	No	No

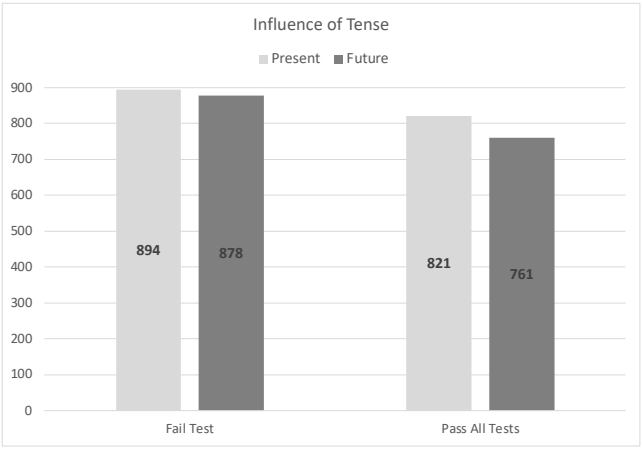

Figure 2: Influence of Tense on GitHub prompts.

In terms of the influence of the prompt features, Table 2 lists all the considered features, and their significance on the generation of code that respectively compiles and passes all the available test cases, for both GitHub and LeetCode. We report *No* when the prompt feature is not significant, we report the boldface p-value with a grey background when the feature is significant, we use a lighter background when the p-value is nearly significant, and – when the prompt feature does not apply.

None of the five studied features have a statistically significant impact on the level of correctness of the results obtained for the prompts derived from GitHub. The only prompt feature with a mild influence on the correctness is the *Tense* (p-value 0.1). Figure 2 shows how *Tense* impacts on the results. The present tense tends to be better interpreted by Copilot than the future tense, especially in terms of code that passes all the test cases.

In the case of LeetCode, two prompt features have been statistically significant: the presence of a *Summary*, for the purpose of obtaining code that passes all the available test cases only, and the

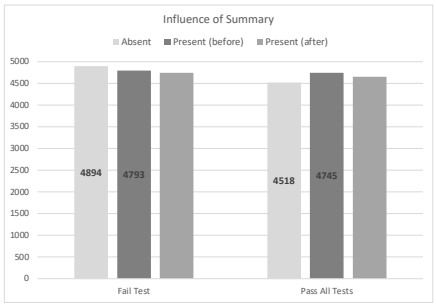

Figure 3: Influence of Summary on testing.

Figure 4: Influence of Examples on compilation.

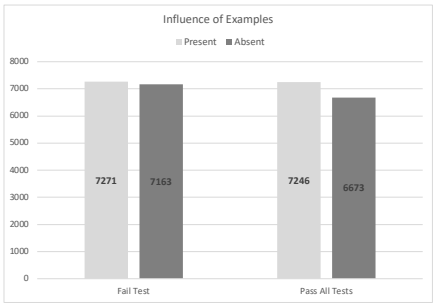

Figure 5: Influence of Examples on testing.

presence of *Examples*, for both the generation of code that compiles and passes all the tests. The *Tense* in LeetCode is not even mildly significant. Probably, the *Tense* has little impact when the description is richer, as the one derived from LeetCode, but it has a slightly more significant impact when the prompt is short.

The presence of a *Summary* of the purpose of the method has been significantly useful for generating code that passes the test cases, while it is not fundamental to simply obtain code that only compiles. Figure 3 shows the impact of *Summary* that, especially when occurring before the rest of the description, increases the number of methods that pass all the available tests.

The presence of *Examples* resulted to be a key element for the success of the code generation task. Figures 4 and 5 show its impact on the generation of code that compiles and passes all the tests, respectively. Indeed, examples provide important guidance for the generation of the right code for Copilot, especially to obtain code that passes the test cases. This is partially surprising since Copilot just interprets the text, not using the examples, for instance to run the generated code, and thus their strong impact on the results was not entirely expected.

In our experiments, the presence of *Context* information about how methods are used, the specification of *Boundary Cases*, the *Reference to Parameters*, as well as the *Mood* and the *Sentence Mode*, have not revealed as introducing any significant difference on the level of correctness of the results. Based on our observations, they might tend to overcomplicate the prompt with information that cannot be fully exploited by the LLM.

Answer to RQ1 According to our results, effective prompts start with a *summary* of the purpose of the method and include some *examples* of its input/output behavior. Prompts should preferably use the *present tense*. Providing additional contextual information does not necessarily produce significantly better results.

4.2 RQ2 - Complexity and Size

This research question investigates the correlation between the complexity and size of the generated code, and the *content* and *style* of the considered prompts.

Table 3 reports the prompt features that have a significant influence on code complexity and size. In the case of GitHub, the *Sentence Mode* has been reported as significantly influencing complexity, although generating a mild impact. In fact, the mean (and standard deviation) of the complexity when using the active mode is $2.81(\pm 2.5)$, while when using the passive mode is $2.72(\pm 2.35)$. It thus seems that using the passive form may mildly influence code complexity, when the prompt is short (e.g., the GitHub prompt) and does not include additional information such as *Examples*, *Summary of the Method*, and *Boundary Cases*, as opposed to LeetCode's prompts. Due to difficulty in explaining observations with LLM, it is hard to identify the reason for such a mild dependency.

Table 3: P-values of prompt features significantly influencing the complexity and size of the generated code.

Prompt Features	GitHub		LeetCode	
	Complexity (CC)	Size (loc)	Complexity (CC)	Size (loc)
Mood	No	No	No	No
Mode	**0.02**	No	No	No
Tense	No	No	No	No
Param	No	No	No	No
B. Cases	$\mathbf{10^{-6}}$	$\mathbf{10^{-1}}$	No	No
Summary	-	-	0.09	No
Examples	-	-	$\mathbf{10^{-11}}$	$\mathbf{10^{-23}}$
Context	-	-	No	No

The specification of the *Boundary Cases* had a significant influence on the complexity and size of the code generated for the GitHub cases.

Figure 6 shows how the presence of the *Boundary cases* induces the generation of longer and slightly more complicated code. In fact, when a boundary case is explicit, Copilot often generates code that explicitly incorporates `if` statements to handle the boundary cases specified in the prompt. For example, one of the parameters of the method `defineLigand`, which is one of the methods that we selected from GitHub, is an object of type `IAtomContainer` named `container`. The implementation is supposed to check the parameter for `null` values, and Copilot adds this check in the code only when the boundary case is specified. The presence of the check increases the level of correctness, but also the length and the complexity of the resulting code. While Copilot often outputs additional checks when boundary cases are included in the prompt, thus increasing the size and complexity of the code, these extra checks are not sufficient to significantly increase the level of correctness

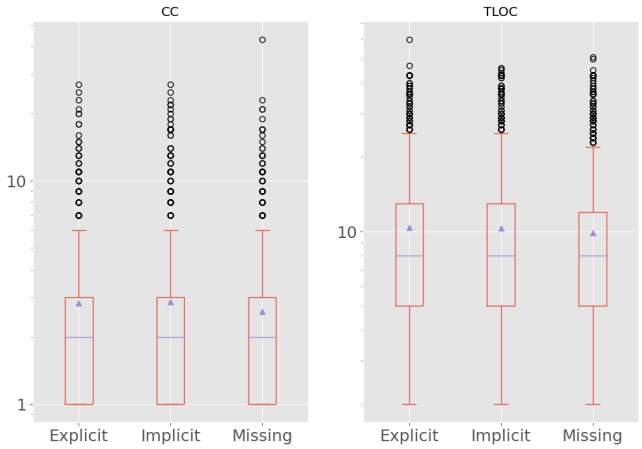

Figure 6: Influence of Boundary Cases on complexity and size.

of the code in our study, since they are sometimes unnecessary or implied by the already generated code (see the results for RQ1).

When the prompt is particularly rich, for instance it already includes summaries and examples, the inclusion of boundary cases does not have an impact on the complexity and size of the code, as reported by the lack of significance of the boundary cases on the size and complexity of the code generated for LeetCode's methods. This is likely to happen because examples and boundary cases report partially overlapping information.

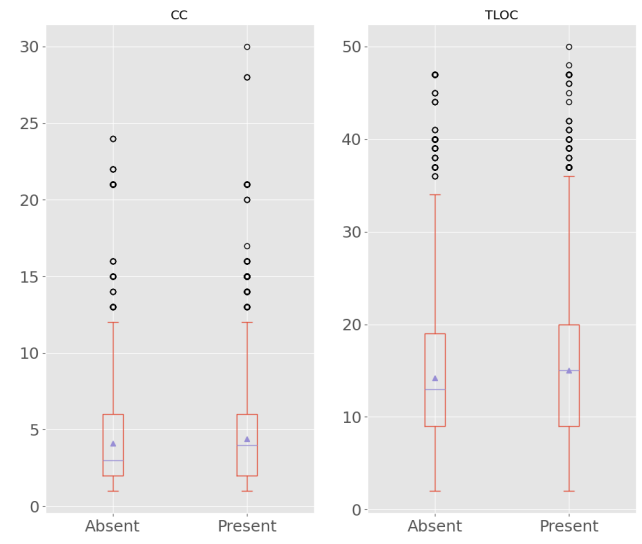

Figure 7: Influence of Examples on complexity and size.

The *Summary* and especially the *Examples* had a significant influence on code complexity and size of the generated code. Figure 7 quantifies the impact of these prompt features. This result was expected since these prompt features correlate with the correctness of the code and the largely wrong code (i.e., code that does not even compile) is often over-simplistic. Table 4 reports the complexity (row *CC*) and size (row *Locs*) for the methods generated for both GitHub and LeetCode cases, distinguishing between the methods that do/do not compile and, among the ones that compile, the ones that fail/do not fail at least a test case. These results show how the shortest code usually does not even compile. In fact, the code that does not compile has the lowest mean complexity and the smallest mean size for both GitHub and LeetCode. On the other hand, it is interesting to see how, among the code that compiles, the most complex and largest one usually fails at least a test case. In fact, the highest mean complexity and largest mean size are reported for the code that fails at least a test for both GitHub and LeetCode.

Table 4: Mean size and complexity of the generated code.

Subject	Metric	Not Compile	Compile	Fail Test	Pass All Tests
GitHub	CC	2.68	2.87	3.06	2.65
	Locs	9.48	11.15	11.42	10.84
LeetCode	CC	1.88	4.85	5.05	4.85
	Locs	6.28	16.70	17.09	16.31

Table 5: Significant relationships between size and complexity of the code to be generated and the level of correctness of the solution.

Subject	Metric	Generated Code Compiles	Generated Code Passes All the Tests
GitHub	CC of the original code	No	10^{-8}
	Locs in the original code	10^{-30}	10^{-32}
LeetCode	CC of the original code	10^{-6}	0.001
	Locs in the original code	10^{-8}	10^{-15}

We also investigated if a similar relationship holds with the code that has to be generated, that is, the complexity and length of the original code extracted from GitHub and LeetCode. Table 5 shows the significant correlations between complexity and size metrics and the resulting level of correctness of the code, distinguishing between achieving code that compiles and code that passes all the test cases. We can notice that there is a strong correlation between all these factors (all the correlations are significant with the exception of code complexity correlated to the generation of code that compiles in GitHub).

Figures 8 and 9 show the results for GitHub and LeetCode, respectively. Interestingly, they show opposite trends for GitHub and LeetCode, concerning the code that fails to compile. Although the population is spread, most of the code that fails to compile originates from the attempt to generate the body of a method that is rather simple and short in GitHub. While most of the code that fails to compile originates from the attempt to generate the body of methods that are rather complex and long in LeetCode. This can be explained by the nature of the cases. The complexity of GitHub is more on the use of appropriate APIs and less on the algorithmic aspects, making the complexity and length of a method a less relevant factor. While the length and complexity are a more important factor for the algorithmic code present in LeetCode.

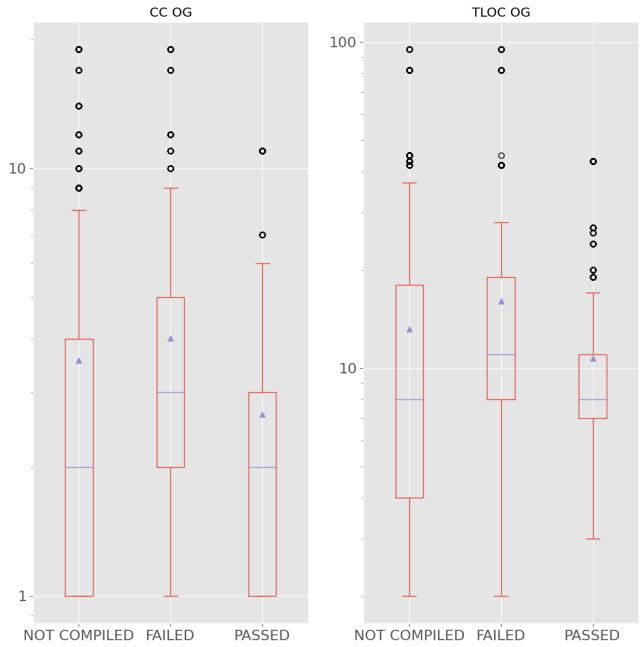

Figure 8: Complexity and size of the developers' code for the GitHub methods whose code generated by Copilot does not compile, fails at least a test case, or passes all the tests.

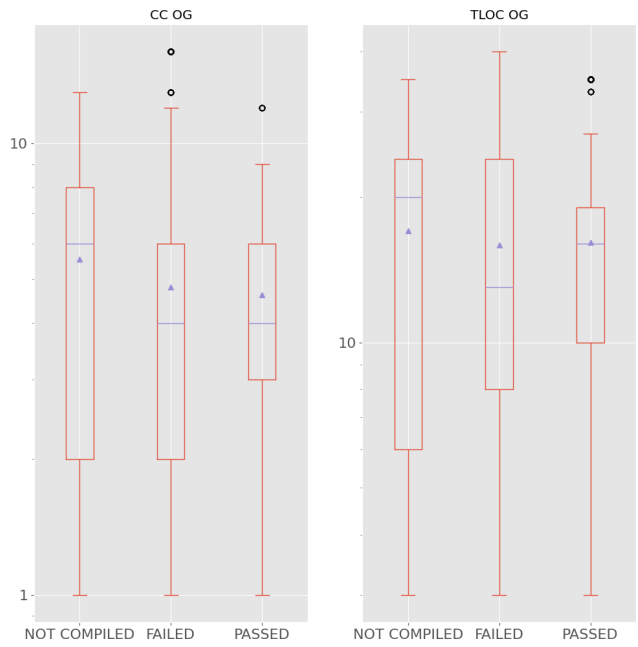

Figure 9: Complexity and size of the developers' code for the LeetCode methods whose code generated by Copilot does not compile, fails at least a test case or passes all the tests.

Once code that compiles is generated, Copilot succeeds mostly with the simpler and shorter methods in GitHub, while it is not necessarily the case in LeetCode.

> **Answer to RQ2** The complexity and length of the generated code are influenced by the *Sentence Mode* of the prompt, the presence of *Boundary Cases*, the presence of the *Summary* and the *Examples*. While the examples demonstrated beneficial for code correctness, it was not the same for the boundary cases. The mode had also a mild impact on the complexity of the generated code, with the passive form producing slightly simpler code.

4.3 RQ3 - Similarity to the Intended Code

This research question investigates how close the generated code, even if incorrect, is to the original code according to the Levenshtein and the CodeBLEU metrics. Table 6 reports the significant prompt features for these two metrics.

Results show that the *Sentence Mode* has an influence on the Levenshtein distance and the CodeBLEU distance for the LeetCode cases. In particular, when the active mode is used, the mean Levenshtein distance is 0.601 while it is 0.599 with the passive mode. Similarly, the CodeBLEU is 0.412 with the active mode and 0.415 with the passive one. The active mode generates code that is slightly more diverse from the original code according to both metrics (higher Levenshtein distance and lower CodeBLEU). However, the impact of this phenomenon is marginal.

The presence of *Boundary cases* also had a significant impact on the similarity of the resulting code compared to the original code. Figure 10 shows the influence of boundary cases. They cause the code to differ slightly more from the original code (higher Levenshtein distance and lower CodeBLEU). This suggests that boundary cases might induce the generation of conditions that should not be present in the code, increasing the level of diversity between the generated code and the original one. Considering that boundary cases do not have a positive effect on correctness (RQ1) and tend to generate longer code (RQ2), this result confirms that it might be better not to include them in the prompt, representing a possible overload of information for the LLM.

Note that improving the similarity between the generated code and the original code is important also when the generated code is not fully correct, since developers will likely have to implement fewer changes to obtain satisfactory code.

Table 6: P-values of prompt features significantly influencing the normalized Levenshtein distance and the CodeBLEU of the generated code.

Prompt Features	GitHub		LeetCode	
	Levenshtein	CodeBLEU	Levenshtein	CodeBLEU
Mood	No	No	No	No
Mode	No	No	**0.05**	**0.01**
Tense	No	No	No	No
Param	No	No	No	No
B. Cases	**0.007**	0.054	No	No
Summary	-	-	No	No
Examples	-	-	No	No
Context	-	-	No	No

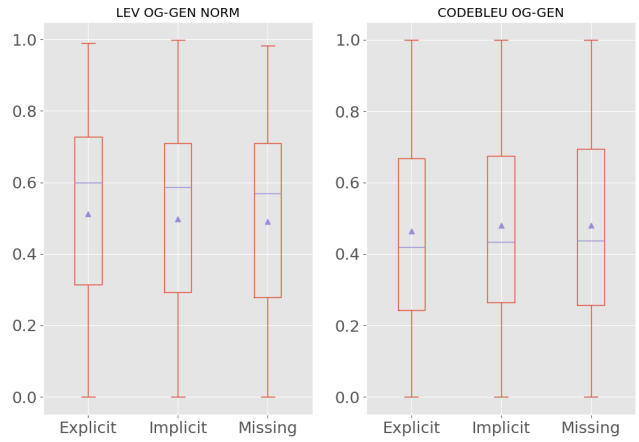

Figure 10: Influence of Boundary Cases on Levenshtein distance and CodeBLEU.

Answer to RQ3 Although none of the prompt features had a large influence on the level of similarity between the code generated by Copilot and the original code, some prompt features may have a mild influence. In particular, using the *passive mode* and not stating *Boundary Cases* explicitly may result in code with higher similarity to the original one.

4.4 Threats to Validity

As with every study, also our empirical study is affected by multiple threats to validity that we identified and addressed as follows.

The main internal threat to validity is about the actual influence that prompt features had on the results. We addressed this threat in two ways. First, we studied prompt features systematically, considering every possible combination. Second, we investigated their impact statistically, to focus on the relevant phenomena. Although we cannot claim the presence of any strong cause-effect relationship between inputs and outputs, due to the low explainability of large language models, the results reported in this paper provide an initial guidance towards the challenge of designing good prompts.

Another threat is how we measure the quality of the generated code. We relied on automated methods to assess the quality of the generated code due to the scale of the study (124,800 methods generated with Copilot). In particular, we relied on the capability to compile the code, and the execution of test cases, to capture different levels of correctness of the code. Indeed, we never claim that a method that passes all the available test cases is a fully correct method, but yet test execution provides useful information about the quality of the code. This limitation is shared with several other studies in the area [4, 15, 21, 30].

To not limit the analysis to test execution, we also consider a combination of syntactic (Levenshtein distance) and semantic (CodeBLEU) metrics that measure the differences between the two methods, disclosing information about the level of closeness of the generated code to the original developers' code.

The main external threat to validity concerns the representativeness and generalizability of the findings. Our study targets a common, although specific, use case, that is, the generation of the body of a method. Results cannot be generalized beyond this use case. Moreover, we focus on the design of a good prompt with the aim of obtaining the right code with a single request. Engineering conversations with large language models is a different problem. Again, although our findings might be useful also in the context of a conversation, the study explicitly focuses on individual and independent interactions (e.g., useful to decide how to start the conversation). Finally, we covered both the case of Java methods extracted from GitHub, representing small-scale problems that developers face daily, and the case of Java methods extracted from LeetCode, representing algorithmic problems that developers may also have to face. We do not know if our findings can be generalized beyond these two cases, which however already cover a reasonable spectrum of the interesting coding problems faced by developers.

4.5 Prompt Advices

Based on our results, there are some advices that could be exploited to design an initial prompt to request for some code. Although the study focuses on individual features, and not on the impact of multiple interacting features, we can conjecture that it is useful to include the following three sections in the prompt:

`<Summary> <Description (Present Tense)><Examples>`

The prompt should start with a short summary of the purpose of the method. It follows a description of the behavior of the method, preferably written using the present tense (with either imperative or indicative mood). The usage of the passive mode may sometimes reduce the length of the generated code. Finally, the examples should include a few input-output pairs that demonstrate how the code should behave in some specific cases.

The following listing shows an example prompt extracted from our dataset that matches this structure. We marked the three main sections with `<summary>`, `<description>`, and `<examples>`.

```
<summary>
Return true if any match is found.
</summary>

<description (present tense)>
Given an input string (s) and a pattern (p), the function
implements wildcard pattern  matching with support for '?' and '*'
where:
'?' Matches any single character.
'*' Matches any sequence of characters (including the empty
    sequence).
The matching should cover the entire input string (not partial).
</description>

<examples>
Example 1:
Input: s = "aa", p = "a"
Output: false
Explanation: "a" does not match the entire string "aa".

Example 2:
Input: s = "aa", p = "*"
Output: true
Explanation: '*' matches any sequence.

Example 3:
Input: s = "cb", p = "?a"
Output: false
Explanation: '?' matches 'c', but the second letter is 'a', which
does not match 'b'.
</examples>
```

5 RELATED WORK

The use of generative models of artificial intelligence, such as language models and code generators, has gained considerable attention in recent years, and is changing the way developers create software. Generative AI solutions, and in particular those built on LLMs have demonstrated remarkable success in understanding and generating code for various programming languages [8, 14, 22]. Some other recent studies investigated how developers interact with these tools during project development [2], and how usable these tools are [2]. However, the effectiveness of these models largely depends on the quality and informativeness of the provided prompts [6]. There is a growing interest in understanding the influence of prompt features on the quality of the generated code.

Prompts are essential for guiding language models towards specific tasks and outputs, without the need for retraining [16, 17, 27]. It can be difficult to understand a model's true capabilities and distinguish between infeasible tasks and those where the model simply misunderstood the prompt. A failed task may indicate a poorly designed prompt, rather than the model's inability to perform the task [3]. For instance, previous research has explored various aspects of prompt engineering. Brown et al. [3] observed that enriching prompts with practical examples of the desired task enhances the capabilities of these models, with more examples correlating positively with better output quality. This is consistent with our findings in the context of LLMs used to generate the body of methods. On the other hand, Reynolds and McDonell [26] studied example-free approaches, focusing on prompt engineering to improve results, outperforming, in some cases, poorly structured prompts with examples. They utilized techniques like task specification, encompassing various methods to describe the same goal, both directly and through proxies, such as analogies and synonyms for common concepts. They also explored methods to restrict unwanted outputs. Other studies [28, 29] have introduced and categorized prompt patterns, similar to software design patterns, with the goal of establishing a context for models like ChatGPT and directing them toward specific desired outcomes. While these patterns were initially developed with software development in mind, their utility extends to a wide range of model applications. Moreover, some of these patterns rely on the conversational aspect of the model, which may not align with Copilot's non-conversational nature. Building upon this prior research, our study uniquely focuses on the impact of eight prompt features on the quality of the code produced by Copilot.

In recent years, various studies have explored the capabilities and limitations of Copilot and its underlying model. Some initial evaluations focused on the classic approach of testing the correctness of the solutions provided [1, 4, 5, 21]. These assessments did not employ prompt engineering techniques, but observed that Copilot was capable of generating correct code for relatively simple tasks even without detailed implementation guidance. An alternative perspective was explored in a study that examined metamorphic testing of Copilot, focusing on prompt modifications [15]. This research investigated the effect of altering prompts, with a unique emphasis on code fragments as prompts rather than natural language descriptions. The study conducted operations involving semantically equivalent mutations, often resulting in distinct code outputs.

Finally, a recent empirical study analyzed the impact of changes made to prompts expressed in natural language through both manual and automated paraphrasing, leveraging tools such as PEGASUS [31] and translation pivoting [19]. The authors created a dataset of Java methods collected from established projects on GitHub. They maintained the original code while automatically generating code for these methods using paraphrases of the original descriptions as prompts for Copilot. The study revealed significant diversification in terms of code correctness, similarity to the original code, and complexity of the code produced, all driven by changes in the prompt. Denny et al. [6] conducted research to assess the impact of prompt modifications when tackling Python programming challenges. Their findings suggest that prompt engineering has proven beneficial in enhancing the accuracy of the resulting code across a majority of cases. White et al. [29] introduced various prompt patterns that programmers can utilize in different scenarios, although they did not provide data regarding the efficacy of these patterns. Additionally, Ren et al. [25] focused on adjusting prompts to facilitate the generation of proper exception handling code.

Nevertheless, to the best of our knowledge, prompt engineering in code generation tasks has not been studied systematically. Our work provides initial insights about prompt engineering for code generation, considering Copilot as generative AI solution.

6 CONCLUSIONS

Generative AI techniques represent sophisticated services that developers can exploit to increase their efficiency and effectiveness in several tasks, including code development. In this context, Copilot is one of the most used tools based on generative AI that can assist developers while they are coding from within the integrated development environment. A key capability of Copilot is the possibility to generate the body of a method starting from a prompt, that is, the request written by the developer as the method's comment.

To obtain a good response from generative AI tools, it is well-known that crafting an appropriate request is mandatory. This paper systematically studies this aspect by empirically investigating how eight prompt features may impact the effectiveness of the prompts submitted to Copilot. The reported study involves 200 Java methods and 124,800 prompts, whose response is assessed in terms of their correctness, complexity, size, and similarity to the intended code.

The results show how only some prompt features influence the results. In particular, we report how a good prompt should include a summary of the semantics of the method, a description of its behavior, possibly using the presence tense, and some examples of input-output pairs that exemplify the behavior of the method. On the contrary, other prompt features, such as the description of boundary cases, the reference to the parameters, and contextual information have little influence on the results. We ended up recommending a structure for the prompt.

Our results represent a starting point for a deeper investigation of this subject. In fact, our future work includes widening the study to consider conversations between the developers and the generative AI tools, and not only individual interactions. Further, we plan to extend our analysis to other tools to investigate to what extent prompt engineering, and in particular results about prompt features, generalize across generative AI solutions.

ACKNOWLEDGMENTS

This work has been partially supported by the Engineered MachinE Learning-intensive IoT systems (EMELIOT) national research project (PRIN 2020 program Contract 2020W3A5FY) and the MUR under the grant "Dipartimenti di Eccellenza 2023-2027" of the Department of Informatics, Systems and Communication of the University of Milano-Bicocca, Italy.

REFERENCES

[1] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732

[2] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proceedings of the ACM Programming Languages* 7, OOPSLA1 (2023). https://doi.org/10.1145/3586030

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared 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 M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models Are Few-Shot Learners. In *Proceedings of the International Conference on Neural Information Processing Systems (NeurIPS)*.

[4] 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. 2021. Evaluating large language models trained on code. arXiv:2107.03374

[5] Vincenzo Corso, Daniela Mariani, Leonardo Micucci, , and Oliviero Riganelli. 2024. Generating Java Methods: An Empirical Assessment of Four AI-Based Code Assistants. In *Proceedings of the International Conference on Program Comprehension (ICPC)*. https://doi.org/10.1145/3643916.3644402

[6] Paul Denny, Viraj Kumar, and Nasser Giacaman. 2023. Conversing with Copilot: Exploring Prompt Engineering for Solving CS1 Problems Using Natural Language. In *Proceedings of the ACM Technical Symposium on Computer Science Education (SIGCSE TS)*. https://doi.org/10.1145/3545945.3569823

[7] Thomas Dohmke. 2023. The economic impact of the AI-powered developer lifecycle and lessons from GitHub Copilot. https://github.blog/2023-06-27-the-economic-impact-of-the-ai-powered-developer-lifecycle-and-lessons-from-github-copilot/.

[8] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[9] GitHub. 2023. Copilot. https://github.com/features/copilot.

[10] GitHub. 2023. GitHub. https://github.com/.

[11] GitHub. 2023. GitHub Copilot in VS Code. https://code.visualstudio.com/docs/editor/github-copilot.

[12] Google. 2023. Bard. https://bard.google.com.

[13] LeetCode. 2023. LeetCode. https://leetcode.com.

[14] 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. 2022. Competition-level code generation with AlphaCode. *Science* 378, 6624 (2022), 1092–1097. https://doi.org/10.1126/science.abq1158

[15] Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Dong Chen, Shuai Wang, and Cuiyun Gao. 2023. CCTEST: Testing and Repairing Code Completion Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE48619.2023.00110

[16] Vivian Liu and Lydia B Chilton. 2022. Design Guidelines for Prompt Engineering Text-to-Image Generative Models. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*. https://doi.org/10.1145/3491102.3501825

[17] Leo S. Lo. 2023. The Art and Science of Prompt Engineering: A New Literacy in the Information Age. *Internet Reference Services Quarterly* 27, 4 (2023), 203–210. https://doi.org/10.1080/10875301.2023.2227621

[18] Thomas W. MacFarland and Jan M. Yates. 2016. Kruskal–Wallis H-test for oneway analysis of variance (ANOVA) by ranks. *Introduction to nonparametric statistics for the biological sciences using R* (2016), 177–211. https://doi.org/10.1007/978-3-319-30634-6_6

[19] Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. 2023. On the Robustness of Code Generation Techniques: An Empirical Study on GitHub Copilot. In *proceedings of the International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE48619.2023.00181

[20] Patrick Mcknight and Julius Najab. 2010. Mann-Whitney U Test. *the Corsini encyclopedia of psychology* (2010). https://doi.org/10.1002/9780470479216.corpsy0524

[21] Nhan Nguyen and Sarah Nadi. 2022. An empirical evaluation of GitHub copilot's code suggestions. In *Proceedings of the International Conference on Mining Software Repositories (MSR)*. https://doi.org/10.1145/3524842.3528470

[22] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[23] OpenAI. 2023. ChatGPT. https://openai.com/chatgpt.

[24] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. arXiv:2009.10297

[25] Xiaoxue Ren, Xinyuan Ye, Dehai Zhao, Zhenchang Xing, and Xiaohu Yang. 2023. From Misuse to Mastery: Enhancing Code Generation with Knowledge-Driven AI Chaining. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. https://doi.org/10.1109/ASE56229.2023.00143

[26] Laria Reynolds and Kyle McDonell. 2021. Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*. https://doi.org/10.1145/3411763.3451760

[27] Alberto D. Rodriguez, Katherine R. Dearstyne, and Jane Cleland-Huang. 2023. Prompts Matter: Insights and Strategies for Prompt Engineering in Automated Software Traceability. In *Proceedings of the Software and Systems Traceability Workshop (SST) at the International Requirements Engineering Conference (RE)*.

[28] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. arXiv:2302.11382

[29] Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C. Schmidt. 2023. ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design. arXiv:2303.07839

[30] Burak Yetistiren, Isik Ozsoy, and Eray Tuzun. 2022. Assessing the Quality of GitHub Copilot's Code Generation. In *Proceedings of the Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*. https://doi.org/10.1145/3558489.3559072

[31] Jingqing Zhang, Yao Zhao, Mohammad Saleh, and Peter Liu. 2020. Pegasus: Pre-training with extracted gap-sentences for abstractive summarization. In *Proceedings of the International Conference on International Conference on Machine Learning (ICML)*.