Copilot-in-the-Loop: Fixing Code Smells in Copilot-Generated Python Code using Copilot

Beiqi Zhang
School of Computer Science
Wuhan University
Wuhan, China
zhangbeiqi@whu.edu.cn

Peng Liang*
School of Computer Science
Wuhan University
Wuhan, China
liangp@whu.edu.cn

Qiong Feng
School of Computer Science
Nanjing University of Science and Technology
Nanjing, China
qiongfeng@njust.edu.cn

Yujia Fu
School of Computer Science
Wuhan University
Wuhan, China
yujia_fu@whu.edu.cn

Zengyang Li
School of Computer Science
Central China Normal University
Wuhan, China
zengyangli@ccnu.edu.cn

ABSTRACT

As one of the most popular dynamic languages, Python experiences a decrease in readability and maintainability when code smells are present. Recent advancements in Large Language Models have sparked growing interest in AI-enabled tools for both code generation and refactoring. GitHub Copilot is one such tool that has gained widespread usage. Copilot Chat, released in September 2023, functions as an interactive tool aimed at facilitating natural language-powered coding. However, limited attention has been given to understanding code smells in Copilot-generated Python code and Copilot Chat's ability to fix the code smells. To this end, we built a dataset comprising 102 code smells in Copilot-generated Python code. Our aim is to first explore the occurrence of code smells in Copilot-generated Python code and then evaluate the effectiveness of Copilot Chat in fixing these code smells employing different prompts. The results show that 8 out of 10 types of code smells can be detected in Copilot-generated Python code, among which *Multiply-Nested Container* is the most common one. For these code smells, Copilot Chat achieves a highest fixing rate of 87.1%, showing promise in fixing Python code smells generated by Copilot itself. In addition, the effectiveness of Copilot Chat in fixing these smells can be improved by providing more detailed prompts.

CCS CONCEPTS

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

KEYWORDS

Code Smell, Code Quality, Code Refactoring, GitHub Copilot

*This work is partially funded by NSFC with No. 62172311.

ACM Reference Format:
Beiqi Zhang, Peng Liang, Qiong Feng, Yujia Fu, and Zengyang Li. 2024. Copilot-in-the-Loop: Fixing Code Smells in Copilot-Generated Python Code using Copilot. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3691620.3695290

1 INTRODUCTION

Code smells refer to the symptoms of poor design and implementation decisions according to the definition by Martin Fowler in his book [9]. Code smells negatively affect the internal quality of software systems, hindering comprehensibility and maintainability [20, 24] and increasing error proneness [16, 19]. The identification of code smells suggests the potential need for code refactoring, pinpointing when and what refactoring can be applied to code [9].

Python, consistently ranked as one of the most popular programming languages [2], is increasingly used in various software development tasks. Python is a high-level, interpreted, and dynamic language that provides a simple but effective approach to object-oriented programming [27]. Due to Python's nature of flexibility and dynamism, developers often find it challenging both to write and maintain Python code [6], and abusing the short constructs of Python can expose code to bad patterns and reduce code readability [3, 17], leading to the occurrence of code smells in Python [7].

Recent advancements in Large Language Models (LLMs) have unveiled impressive capabilities in solving various Natural Language Processing (NLP) tasks [29, 30], showcasing their effectiveness in e.g., code generation [13] and refactoring[23, 26]. Released in June 2021, GitHub Copilot powered by LLM (i.e., OpenAI Codex) has been widely embraced by developers for code auto-completion, and it has evolved into the world's most widely adopted AI developer tool [1]. However, concerns arose regarding the quality of code generated by Copilot [31, 33]. Copilot's code suggestion algorithms are incentivized to propose suggestions more likely to be accepted rather than easy to read and understand, which has an adverse impact for long-term code maintainability [11]. Subsequently, in September 2023, a public beta of GitHub Copilot Chat has been released as an interactive tool for Copilot, aiming to enable natural language-powered coding [34]. Developers can utilize Copilot Chat for tasks such as code analysis and fixing security issues, democratizing software development for a new generation. Given the potential for Copilot-generated Python code to exhibit code smells and the capacity of Copilot Chat to assist in rectifying such issues, this paper delves into fixing code smells in Copilot-generated Python code using Copilot Chat. **In this paper**, to evaluate the prevalence of code smells in Copilot-generated Python code and the competence of Copilot Chat in fixing Python smells, we built a dataset with 102 code smells in Copilot-generated Python code. Specifically, we investigated two Research Questions (RQs):

- **RQ1**: To what extent does the Copilot-generated Python code contain code smells?
- **RQ2**: How effective is Copilot Chat in fixing different types of code smells in Copilot-generated Python code?

Our preliminary findings show that 14.8% Python files generated by Copilot contain code smells, with *Multiply-Nested Container* being the dominant code smell. GitHub Copilot exhibits promising potential in fixing code smells in Copilot-generated Python code, and the results indicate that Copilot Chat performs the best in fixing Python code smells by the prompt with code smell name.

2 BACKGROUND & RELATED WORK

2.1 Definition of Python Code Smells

Considering the multiple programming paradigms and flexible grammatical constructs of Python (a dynamic programming language), the types of code smells presented by Martin Fowler [9] that target in static programming language are not entirely applicable to Python code smells [6]. Chen *et al.* [7] proposed a set of 10 Python code smells (see Table 1), which we considered in this study. These Python code smells are metric-based detectable using `Pysmell` [7], and these smells have been widely used in various studies (e.g., [10, 14]) that explored code smells in Python projects.

Table 1: Definition of Python code smells.

Code Smell	Definition
Long Parameter List (LPL)	A method or function with an extensive list of parameters [9].
Long Method (LM)	A method or function that exceeds a considerable length [9].
Long Scope Chaining (LSC)	A method or function with deeply nested closures [9].
Large Class (LC)	A class that exceeds a considerable length [9].
Long Message Chain (LMC)	An expression that accesses an object through an extended chain of attributes or methods using the dot operator [5].
Long Base Class List (LBCL)	A class definition that inherits from an excessive number of base classes [7].
Long Lambda Function (LLF)	A lambda function that exceeds a considerable length [7].
Long Ternary Conditional Expression (LTCE)	A ternary conditional expression that exceeds a considerable length [7].
Complex Container Comprehension (CCC)	A container comprehension (i.e., list, set, and dict comprehension, along with generator expression) with excessive complexity [7].
Multiply-Nested Container (MNC)	A container (i.e., set, list, tuple, and dict) with deep nesting [7].

2.2 Quality of Copilot-Generated Code

Yetistiren *et al.* [31] evaluated the quality of Copilot-generated code, focusing on validity, correctness, and efficiency. They found Copilot to be a promising tool for code generation tasks. Similarly, Nguyen and Nadi [18] assessed Copilot-generated code for correctness and understandability. Their results show that Copilot-generated code exhibits low complexity, with no notable differences observed across programming languages. Pearce *et al.* [21] investigated the prevalence and conditions that can cause GitHub Copilot to recommend insecure code. Their findings revealed that about 40% of programs generated by Copilot were deemed vulnerable.

Different to the studies above, our work focuses on investigating a specific issue—code smells—in Copilot-generated Python code and accessing Copilot Chat's ability to fix these smells in the loop.

3 METHODOLOGY

3.1 Use `Pysmell` to Detect Python Code Smells

This study aims to explore Python code smells in Copilot-generated code and evaluate Copilot Chat's capability to fix Python smells. We first built a dataset consisting of 102 code smells in Copilot-generated Python code by the following steps (see Figure 1):

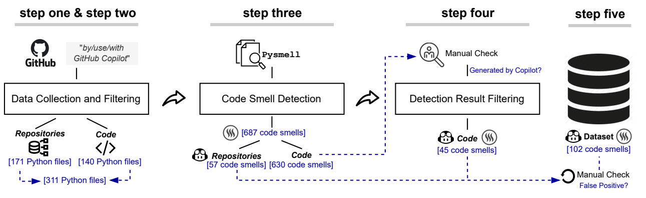


Figure 1: Overview of the dataset construction process.

In **step one**, we used a keyword-based mining approach to collect Copilot-generated Python files from GitHub. Before the search process, a pilot search was conducted using the keyword "*GitHub Copilot*" directly within GitHub's search engine. The pilot search results returned by GitHub were grouped into two categories, i.e., projects containing the keyword labeled as *Repositories* and code files containing the keyword labeled as *Code*. Under the *Repositories* label, some projects claim in their README files or project descriptions that they were entirely generated by Copilot. Similarly, under the *Code* label, certain files contain Copilot-generated code, as indicated by comments within the code. However, the pilot search using "*GitHub Copilot*" included many irrelevant results. Our observations from the pilot search showed that using "*by GitHub Copilot*", "*use GitHub Copilot*", and "*with GitHub Copilot*" as keywords could improve the accuracy of search results. Hence, we established the aforementioned three keywords as our search terms. The search was conducted on November 30, 2023, and Table 2 shows the number of retrieved repositories and code. However, a Python file under the *Code* label might contain multiple sets of keywords, implying duplicates among the 2,917 Python files we collected. After deduplication, 1,204 distinct Python files under the *Code* label were retained.

Table 2: Search terms used in GitHub.

#	Search Term	# Repositories	# Code
ST1	"*by GitHub Copilot*"	33	896
ST2	"*use GitHub Copilot*"	52	1,069
ST3	"*with GitHub Copilot*"	68	952
Total		153	2,917

In **step two**, to manually label whether the projects under the *Repositories* label were entirely generated by Copilot and whether the Python files under the *Code* label contain Copilot-generated code, the first and fourth authors conducted a pilot data labelling by randomly selecting 10 candidates under each label. The two authors independently labelled whether these projects and code files should be included based on project documentation, code comments, and other metadata in the search results. Data labelled by the authors were compared, and the level of agreement between them were calculated using the Cohen's Kappa coefficient [8]. The Cohen's Kappa coefficient was 0.79 for the projects under the *Repositories* label and 0.85 for the code files under the *Code* label, indicating a high level of agreement between the two authors. After the pilot labelling, the two authors checked all the projects and code files retrieved from GitHub. In the labelling process, if the two authors were unsure about whether a project or code file should be included, they discussed it with the second author until all three reached a consensus. After manually filtering all the candidates, we collected 51 projects from *Repositories* and 140 Python files from *Code*. The 51 repositories comprised 171 Python files that were entirely generated by Copilot, while only a portion of the 140 code files were generated by Copilot. In total, we got 311 (171 + 140) Python files after this step (see Figure 1). Note that the version information of Copilot that generated these Python files cannot be identified.

In **step three**, we utilized `Pysmell` [7], which has been widely used in various studies exploring code smells in Python [10, 14, 22, 28], to detect the 10 code smells listed in Section 2.1 in the 311 Python files obtained in **step two**. `Pysmell` has three thresholds for smell detection, and we opted for the Tuning Machine Strategy due to empirical evidence indicating its superior accuracy in detecting Python smells among the three strategies [7]. A total of 687 code smells were detected by `Pysmell` in this step. All the code smells (57) detected in the Python files under the *Repositories* label were generated by Copilot. However, under the *Code* label, not all the detected Python smells (630) were Copilot-generated ones.

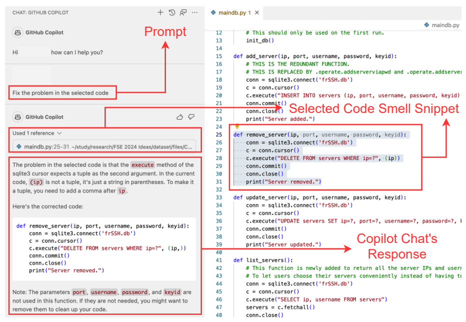
In **step four**, the first author manually checked whether the 630 instances of code smell under the *Code* label obtained in **step three** were generated by Copilot. Only code smells in the Python files with code comments explicitly indicating that they were generated by Copilot were retained. During the manual checking process, in cases where the first author had uncertainties regarding the inclusion of a Python smell in the dataset, the second author was consulted for a discussion until an agreement was reached. Among the 630 code smells under the *Code* label, 45 were generated by Copilot, resulting in a total of 102 (57 from *Repositories* and 45 from *Code*) code smells in Copilot-generated code were identified.

In **step five**, the first author rechecked all the identified code smells in **step four** to determine any potential false positives. After the manual check, all the 102 Python smells were confirmed as true positives, which was reasonable as `Pysmell` with Tuning Machine Strategy attains high precision in Python smell detection [7].

3.2 Use Copilot Chat to Fix Python Code Smells

3.2.1 Prompt Design. Referring to the foundational prompt that instructs Copilot Chat to improve non-functional requirement of accessibility [25], we initially conducted a series of pilot experiments employing different prompt structures and formulations to fix code smells using Copilot Chat. Based on our prior observations, we selected three prompts of varying detail levels that demonstrated various effectiveness in fixing Python code smells:
General Fix Prompt: *Fix the **problem** in the selected code*
Code Smell Fix Prompt: *Fix the **code smell** in the selected code*
Specific Code Smell Fix Prompt: *Fix the **[code smell name] (e.g., Long Method) code smell** in the selected code*

The three prompts mentioned above each provide more details than the one before it, which enables us to explore Copilot Chat's effectiveness in fixing Copilot-generated code smells when provided with different levels of information. We first selected the code snippets of identified code smells and provided these 3 types of prompts to Copilot Chat in the chat window of Visual Studio Code. Copilot Chat then utilized the selected code as references (input) to generate responses to our prompts (see Figure 2).



Hi how can I help you?

Fix the problem in the selected code

GitHub Copilot

Used 1 reference

maindb.py:25-31 ~/study/research/FSE 2024 Ideas/dataset/files/C...

The problem in the selected code is that the `execute` method of the sqlite3 cursor expects a tuple as the second argument. In the current code, `(ip)` is not a tuple, it's just a string in parentheses. To make it a tuple, you need to add a comma after `ip`.

Here's the corrected code:

```python
def remove_server(ip, port, username, password, keyid):
    conn = sqlite3.connect('frSSH.db')
    c = conn.cursor()
    c.execute("DELETE FROM servers WHERE ip=?", (ip,))
    conn.commit()
    conn.close()
    print("Server removed.")
```

Note: The parameters `port`, `username`, `password`, and `keyid` are not used in this function. If they are not needed, you might want to remove them to clean up your code.

```python
    # This should only be used on the first run.
    init_db()

def add_server(ip, port, username, password, keyid):
    # THIS IS THE REDUNDANT FUNCTION.
    # THIS IS REPLACED BY .operate.addserverviapwd and .operate.addserve
    conn = sqlite3.connect('frSSH.db')
    c = conn.cursor()
    c.execute("INSERT INTO servers (ip, port, username, password, keyid)
    conn.commit()
    conn.close()
    print("Server added.")

def remove_server(ip, port, username, password, keyid):
    conn = sqlite3.connect('frSSH.db')
    c = conn.cursor()
    c.execute("DELETE FROM servers WHERE ip=?", (ip))
    conn.commit()
    conn.close()
    print("Server removed.")

def update_server(ip, port, username, password, keyid):
    conn = sqlite3.connect('frSSH.db')
    c = conn.cursor()
    c.execute("UPDATE servers SET ip=?, port=?, username=?, password=?, 
    conn.commit()
    conn.close()
    print("Server updated.")

def list_servers():
    # This function is newly added to return all the server IPs and user
    # To let users choose their servers conveniently instead of having t
    conn = sqlite3.connect('frSSH.db')
    c = conn.cursor()
    c.execute("SELECT ip, username FROM servers")
    servers = c.fetchall()
    conn.close()
```

Figure 2: An example of using Copilot Chat to fix Python code smells in Visual Studio Code.

3.2.2 Code Smell Snippets Used as Referenced Code. `Pysmell` only provided the starting line and type of the code smells detected. We combined the information to determine the corresponding code snippet for each instance of the identified code smells. Consequently, 102 code smell snippets were obtained in Copilot-generated Python code at the class, method, or expression level. Notably, we observed that an individual code snippet could manifest multiple identical smells. To guide Copilot Chat in fixing code smell snippets, we constrained the selected snippets used as references to include at least one complete line of code, thereby ensuring that ample contextual information was provided to Copilot Chat. Hence, the number of code snippets that contain code smells for Copilot Chat to fix was reduced to 96. Among these, 3 exceeded the token length limit and were removed. Out of the remaining 93 instances of code smells, 91 were in separate code snippets, while 2 different code smells were located in the same code snippet. We used that particular code snippet as a reference to address the 2 code smells, resulting in 92 code snippets encompassing the 93 distinct instances of code smells. We used these 92 code smell snippets as referenced code and applied the prompts outlined in Section 3.2.1 as input to instruct Copilot Chat in fixing the smells. We recorded Copilot Chat's responses to our input for further evaluation, which are provided at [32].

3.3 Evaluation of Code Smell Fixing

To evaluate the effectiveness of Copilot Chat in fixing Python code smells generated by Copilot itself, the first author manually reviewed the code refactored by Copilot Chat utilizing the threshold

Table 4: Fixing rates for different types of code smells using different prompts.

Prompt	MNC	LPL	LM	LLF	LTCE	CC	CMC	LC	Avg
General Fix Prompt	19.4%	0.0%	50.0%	**100.0%**	80.0%	50.0%	50.0%	**100.0%**	34.4%
Code Smell Fix Prompt	58.3%	22.7%	91.7%	**100.0%**	**100.0%**	**100.0%**	**100.0%**	**100.0%**	64.5%
Specific Code Smell Fix Prompt	**69.4%**	**95.5%**	**100.0%**	**100.0%**	**100.0%**	**100.0%**	**100.0%**	**100.0%**	**87.1%**
Avg	49.1%	39.4%	80.6%	**100.0%**	93.3%	83.3%	83.3%	**100.0%**	

in the Tuning Machine Strategy of `Pysmell` [7] as the benchmark to verify if the code smell was fixed or not. If the original code smell was resolved, we labeled it as "*Fixed*", otherwise, we labelled it as "*Unfixed*". We defined "*Fixing Rate*" indicating the proportion of fixed smells to evaluate the effectiveness of code smell fixing.

Table 3: Code smell types detected in Copilot-generated Python code.

Code Smell	#	%	Code Smell	#	%
MNC	41	40.2%	LTCE	5	4.9%
LPL	22	21.5%	CCC	4	3.9%
LM	14	13.7%	LMC	2	2.0%
LLF	12	11.8%	LC	2	2.0%

4 RESULTS

4.1 Results of RQ1

4.1.1 The proportion of Copilot-generated Python files that contain code smells. In total, we collected 171 Python files from the *Repositories* label and 140 from the *Code* label. Among these 311 (171+140) Python files, 46 contain code smells generated by Copilot, accounting for 14.8%.

4.1.2 The types of code smells detected in Copilot-generated Python code. Table 3 presents the 8 types of code smells detected in Copilot-generated Python code. Among the 10 detectable Python code smells listed in Section 2.1, 2 (LSC and LBCL) were not found in Copilot-generated code. *MNC*, which accounts for over 40%, is the most common type of code smell in Copilot-generated Python code, followed by *LPL*, which represents over 20%. *LMC* and *LC* are the least identified code smells in Copilot-generated Python code, each with a proportion of 2.0% of the total.

4.2 Results of RQ2

Table 4 lists Copilot Chat's average fixing rates with different prompts. Overall, the *Specific Code Smell Fix Prompt*, which provides Copilot Chat with the particularized names of the code smells that needed to be fixed, achieved the highest average fixing rate at 87.1%. On the other hand, the average fixing rate of the *General Fix Prompt*, which instructs Copilot Chat to resolve potential issues in the referenced code without indicating the issue is a code smell, is the lowest (34.4%). This result is in line with our intuition that using more detailed prompts to instruct Copilot Chat might get more effective code fix suggestions.

Copilot Chat's fixing rates for different types of code smells using different prompts are also showed in Table 4. In general, Copilot Chat demonstrates the best effectiveness in fixing *LLF* and *LC*, both achieving a fixing rate of 100.0%. Conversely, Copilot Chat exhibits the lowest effectiveness in fixing *LPL* and *MNC*, with fixing rates of 39.4% and 49.1%, respectively. We can also find that when using the three prompts with varying levels of detail (see Section 3.2.1) to instruct Copilot Chat in fixing the Python code smells detected in Copilot-generated code, the fixing rate with *Specific Code Smell Fix Prompt* is consistently the highest for all the 8 types of code smells, while that with *General Fix Prompt* is consistently the lowest.

5 DISCUSSION

Attention to *MNC* and *LPL* in Copilot-generated Python code: According to the RQ1 results (see Section 4.1), about 15% Copilot-generated Python files contain code smells, and the top two code smells are *MNC* and *LPL*. However, based on the RQ2 results (see Section 4.2), Copilot Chat shows the worst effectiveness in fixing these two code smells. The occurrence of *MNC* reduces code readability and may obscure bugs, while *LPL* makes code more complex [7], both negatively impacting the Python code quality. Therefore, developers should pay attention to *MNC* and *LPL* when using Copilot to generate Python code and Copilot Chat to fix them.

Enhanced effectiveness through Detailed Prompts for Copilot Chat: We used three prompts of varying detail levels to instruct Copilot Chat to fix code smells in Copilot-generated Python code. The RQ2 results (see Section 4.2) show that *Specific Code Smell Fix Prompt*, providing comprehensive information, yielded the most favorable outcomes, while *General Fix Prompt*, providing minimum information, produced the least effective results. This finding aligns with our expectation that Copilot Chat would exhibit better effectiveness in fixing Copilot-generated Python smells when offered more detailed prompts. When instructing Copilot Chat to fix Copilot-generated Python smells, developers can provide the specific type of code smell to improve Copilot Chat's effectiveness.

6 CONCLUSIONS AND FUTURE WORK

In this work, we constructed a dataset of 102 code smells in Copilot-generated Python code from GitHub, and evaluated the effectiveness of Copilot Chat in fixing these code smells. The results show that 8 types of Python smell were detected in Copilot-generated code, and the dominant code smell is *MNC*. Copilot Chat demonstrates promise in fixing Python smells in Copilot-generated code.

Potential avenues for future work include: (1) explore code smells in AI-generated code with other languages such as Java, C/C++ and Rust, (2) investigate the impact of different prompt methods (e.g., few-shot learning [4] and CoT prompting [29]) and LLM-based frameworks (e.g., RAG [15] and multi-agent systems [12]) on Copilot Chat's ability to fix code smells, and (3) explore the combination of Copilot Chat with other code analysis tools to enhance its code smell detection and fixing capabilities.

Our work serves as **a starting point** for investigating the identification of code smells in AI-generated code and exploring the potential of AI-coding tools in fixing the smells by themselves.

REFERENCES

[1] 2024. *GitHub Copilot - Your AI Pair Programmer.* https://github.com/features/copilot.

[2] 2024. *TIOBE Index for January 2024.* https://www.tiobe.com/tiobe-index/.

[3] David M Beazley. 2009. *Python Essential Reference.* Addison-Wesley Professional.

[4] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. 33 (2020), 1877–1901.

[5] William H Brown, Raphael C Malveau, Hays W "Skip" McCormick, and Thomas J Mowbray. 1998. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley & Sons, Inc.

[6] Zhifei Chen, Lin Chen, Wanwangying Ma, and Baowen Xu. 2016. Detecting Code Smells in Python Programs. In *Proceedings of the 7th International Conference on Software Analysis, Testing and Evolution (SATE).* IEEE, 18–23.

[7] Zhifei Chen, Lin Chen, Wanwangying Ma, Xiaoyu Zhou, Yuming Zhou, and Baowen Xu. 2018. Understanding metric-based detectable smells in Python software: A comparative study. *Information and Software Technology* 94 (2018), 14–29.

[8] Jacob Cohen. 1960. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46.

[9] Martin Fowler. 1999. *Refactoring - Improving the Design of Existing Code.* Addison-Wesley Professional.

[10] Jiri Gesi, Siqi Liu, Jiawei Li, Iftekhar Ahmed, Nachiappan Nagappan, David Lo, Eduardo Santana de Almeida, Pavneet Singh Kochhar, and Lingfeng Bao. 2022. Code smells in machine learning systems. *arXiv preprint arXiv:2203.00803* (2022).

[11] William Harding and Matthew Kloster. 2024. *Coding on Copilot: 2023 Data Suggests Downward Pressure on Code Quality.* https://www.gitclear.com/coding_on_copilot_data_shows_ais_downward_pressure_on_code_quality.

[12] Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352* (2023).

[13] Maliheh Izadi, Jonathan Katzy, Tim Van Dam, Marc Otten, Razvan Mihai Popescu, and Arie Van Deursen. 2024. Language Models for Code Completion: A Practical Evaluation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE).* ACM, 13.

[14] Hadhemi Jebnoun, Houssem Ben Braiek, Mohammad Masudur Rahman, and Foutse Khomh. 2020. The scent of deep learning code: An empirical study. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR).* ACM, 420–430.

[15] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Proceedings of the 34th Annual Conference on Neural Information Processing Systems (NeurIPS).* ACM, 9459–9474.

[16] Wei Li and Raed Shatnawi. 2007. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software* 80, 7 (2007), 1120–1128.

[17] Mark Lutz. 2010. *Programming Python: Powerful Object-Oriented Programming.* O'Reilly Media, Inc.

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

[19] Steffen M. Olbrich, Daniela S. Cruzes, and Dag I.K. Sjøberg. 2010. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE).* ACM, 1–10.

[20] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering* 23, 3 (2018), 1188–1221.

[21] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (SP).* IEEE, 754–768.

[22] Rana Sandouka and Hamoud Aljamaan. 2023. Python code smells detection using conventional machine learning models. *PeerJ Computer Science* 9 (2023), e1370.

[23] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An analysis of the automatic bug fixing performance of chatgpt. In *Proceedings of the IEEE/ACM 4th International Workshop on Automated Program Repair (APR).* IEEE, 23–30.

[24] Zéphyrin Soh, Aiko Yamashita, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2016. Do code smells impact the effort of different maintenance programming activities?. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER).* IEEE, 393–402.

[25] Ed Summers and Jesse Dugas. 2023. *Prompting GitHub Copilot Chat to become your personal AI assistant for accessibility.* https://github.blog/2023-10-09-prompting-github-copilot-chat-to-become-your-personal-ai-assistant-for-accessibility/.

[26] Nigar M Shafiq Surameery and Mohammed Y Shakor. 2023. Use chat gpt to solve programming bugs. *International Journal of Information Technology & Computer Engineering* 3, 01 (2023), 17–22.

[27] Guido Van Rossum and Fred L Drake Jr. 1995. *Python Tutorial.* Vol. 620. Centrum Wiskunde & Informatica.

[28] Natthida Vatanapakorn, Chitsutha Soomlek, and Pusadee Seresangtakul. 2022. Python Code Smell Detection Using Machine Learning. In *Proceedings of the 26th International Computer Science and Engineering Conference (ICSEC).* IEEE, 128–133.

[29] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Proceedings of the 36th Annual Conference on Neural Information Processing Systems (NeurIPS).* Curran Associates, Inc., 24824–24837.

[30] Jingfeng Yang, Hongye Jin, Ruixiang Tang, Xiaotian Han, Qizhang Feng, Haoming Jiang, Shaochen Zhong, Bing Yin, and Xia Hu. 2024. Harnessing the Power of LLMs in Practice: A Survey on ChatGPT and Beyond. 18, 6 (2024), 32.

[31] Burak Yetistiren, Isik Ozsoy, and Eray Tuzun. 2022. Assessing the Quality of GitHub Copilot's Code Generation. In *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE).* ACM, 62–71.

[32] Beiqi Zhang, Peng Liang, Qiong Feng, Yujia Fu, and Zengyang Li. 2024. *Dataset of the Paper: Copilot-in-the-Loop: Fixing Code Smells in Copilot-Generated Python Code using Copilot.* https://doi.org/10.5281/zenodo.13335000.

[33] Beiqi Zhang, Peng Liang, Xiyu Zhou, Aakash Ahmad, and Muhammad Waseem. 2023. Demystifying Practices, Challenges and Expected Features of Using GitHub Copilot. *International Journal of Software Engineering and Knowledge Engineering* 33, 11&12 (2023), 1653–1672.

[34] Shuyin Zhao. 2023. *GitHub Copilot Chat beta now available for all individuals.* https://github.blog/2023-09-20-github-copilot-chat-beta-now-available-for-all-individuals/.