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

Priyan Vaithilingam
pvaithilingam@g.harvard.edu
Harvard University
USA

Tianyi Zhang
tianyi@purdue.edu
Purdue University
USA

Elena L. Glassman
glassman@seas.harvard.edu
Harvard University
USA

ABSTRACT

Recent advances in Large Language Models (LLM) have made automatic code generation possible for real-world programming tasks in general-purpose programming languages such as Python. However, there are few human studies on the usability of these tools and how they fit the programming workflow. In this work, we conducted a within-subjects user study with 24 participants to understand how programmers use and perceive Copilot, a LLM-based code generation tool. We found that, while Copilot did not necessarily improve the task completion time or success rate, most participants preferred to use Copilot in daily programming tasks, since Copilot often provided a useful starting point and saved the effort of searching online. However, participants did face difficulties in understanding, editing, and debugging code snippets generated by Copilot, which significantly hindered their task-solving effectiveness. Finally, we highlighted several promising directions for improving the design of Copilot based on our observations and participants' feedback.

CCS CONCEPTS

• **Human-centered computing → Empirical studies in HCI**.

KEYWORDS

large language model, github copilot

ACM Reference Format:
Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts (CHI '22 Extended Abstracts), April 29-May 5, 2022, New Orleans, LA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3491101.3519665

1 INTRODUCTION

Automatic code generation has been a long-term goal for multiple research communities including Programming Languages (PL), Software Engineering (SE), Natural Language Processing (NLP), and Machine Learning (ML). Recent attempts to achieve this have focused on two different kinds of approaches: (1) program synthesis algorithms that search over a large program space defined by a domain-specific language (DSL) [2, 7, 10, 12, 14, 19, 24, 25, 30, 31, 34, 43], and (2) deep learning models that are trained on a large amount of existing code and can generate new code given some forms of specifications such as natural language descriptions or incomplete code [5, 16, 17, 22, 38, 39, 48, 49]. Both kinds of approaches have clear drawbacks. On the one hand, existing program synthesis techniques are constrained to pre-defined DSLs and cannot scale to general-purpose programming languages [15]. On the other hand, existing generative models have a hard time learning sophisticated programming patterns from code corpora and often generate code with syntactic or semantic errors [9, 29, 40]. The recent development of Large Language Models (LLM) such as GPT-3 [32] has opened up new opportunities for addressing the limitations of existing code generation techniques. For example, Codex [50], which contains 12 billion model parameters and is trained on 54 million software repositories on GitHub, has demonstrated stunning code generation capability—solving over 70% of 164 Python programming tasks with 100 samples [8].

The performance of LLM-based code generation tools has been extensively studied using benchmarks [8, 33]. However, little is known about the usability and programmers' perception of such a tool in a real-world programming workflow. To bridge the gap, we conducted a within-subjects comparative study with 24 participants, in which participants were asked to complete Python programming tasks. In the experimental condition, participants wrote programs with the assistance of Copilot, a Visual Studio Code (VSCode) plugin powered by Codex [13]. In the control condition, participants wrote programs with the assistance of Intellisense, the default code completion plugin in VSCode. We investigated the following research questions:

- **RQ1:** How does using Copilot affect the programming experience?
- **RQ2:** How do users recognize errors in code generated by Copilot?
- **RQ3:** What coping mechanisms do users employ when they find errors in code generated by Copilot?
- **RQ4:** What are the obstacles and limitations that can prevent adoption of Copilot?

Our key findings are: (1) the majority of the participants (19 out of 24) preferred using Copilot over Intellisense (Control condition); (2) Copilot provides a useful starting point for participants to kick start the task and saved them the effort of searching online; (3) There is a need to identify better ways for participants to understand long blocks of generated code to help them edit, debug, and repair the code.

CHI '22 Extended Abstracts, April 29-May 5, 2022, New Orleans, LA, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9156-6/22/04…$15.00
https://doi.org/10.1145/3491101.3519665

2 RELATED WORK

2.1 AI-based Code Generation

There is a long history of research on automated code generation. Some of the earliest work dates back to the 1960s, where Waldinger and Lee presented a program synthesizer called PROW that automatically generated LISP programs based on user-provided specifications in the form of a predicate calculus [41].

There are two main trends in modern automatic code generation: program synthesis and machine learning. Program Synthesis primarily uses a search-based technique to generate code that fulfills a given specification. These techniques work on a subset of the language components relevant to the domain known as Domain-Specific Languages (DSLs). More recently, program synthesis has been applied to a variety of domains, e.g., low-level bit-vector implementations [36], data manipulation in excel [14], and regular expression synthesis [51]. The main limitation is that these techniques are limited to a pre-defined DSL, making it less scalable to programs written in general-purpose programming languages such as Java or Python. Because general-purpose programming languages include much more language features and syntax rules compared with DSLs and therefore define a much bigger program space to search from [15].

The second trend is using machine learning, especially deep learning. Advances in deep learning have shown promising results on automatically generating code for real-world programming tasks [5, 16, 17, 22, 38, 39, 48, 49]. For instance, Kim et al. [21] developed a transformer architecture that is aware of code structures using abstract syntax trees. Alon et al. [1] introduced structural language models that remove any restriction on the vocabulary or structure— the main limitation of program synthesis techniques. Karampatsis and Sutton [20] similarly introduced open-vocabulary models that can generate code with an arbitrary number of tokens. Though these methods have shown promising results, they still suffer from low accuracy and are less reliable [9, 29, 40]. For instance, Ciniselli et al. [9] show their RoBERTa-based model can only produce correct solutions for 7% of the tasks from the CodeSearchNet benchmark [18].

The recent advances in large language models (LLM) such as GPT-3 [32] have led to a breakthrough in automated code generation compared to prior state-of-the-art deep learning methods [4, 6, 42]. For example, Codex [50], a fine-tuned version of GPT-3, can generate fully correct code for 29% of unseen programming tasks with only one sample of generated programs and 72% of them with 100 samples, while a widely used code generation tool, TabNine [39] can only solve 3% and 8%, respectively [8].

While there has been recent work evaluating the accuracy of LLM-based code generation tools [8, 33], little is known about its usability. With such increases in accuracy, how will programmers interact with a tool that generates almost accurate yet not perfect code? How easy or difficult is it for programmers to recognize errors in a code snippet that is almost but not quite correct? Will they simply modify the incorrect part or completely rewrite the entire code themselves? This motivates us to study programmers' expectations, coping strategies, and needs for such powerful code generation tools.

2.2 Coping with Imperfect AI

Prior studies have examined how users interact with imperfect AI [11, 23, 26–28, 35, 37, 45]. Dzindolet et al. [11] showed that once people observed an automated system make errors, their distrust in the system increased unless an explanation was provided. However, these explanations may also lead to over-reliance on the system even when unwarranted, signaling the importance and difficulty of providing explanations that help people to calibrate trust appropriately. Kocielnik et al. [23] examined the effect of giving people control over the types of errors made by a scheduling assistant, either by avoiding false positives or false negatives. They found that even when the system was only 50% accurate, users who expected a reduction in the false positive rate had a lower perception of accuracy and lower acceptance of the system than the users who expected a reduction in the false negative rate. [3, 52] showed that confidence scores helped calibrate users' trust, form a good mental model of the AI, and understand the error boundaries better.

Similar to other AI techniques, AI-based code generation tools also suffer from inherent uncertainty and imperfection. They may inevitably generate code with errors or even code that wildly differs from users' expectations. However, unlike other domains, code generation demands a much higher level of correctness: code either compiles or not, and it is either correct or contains bugs such as logic errors and security vulnerabilities. Therefore, existing findings of other types of AI techniques may not generalize to the domain of code generation.

Currently, there are only a few studies on how programmers use such imperfect code generation tools [44, 47]. Xu et al. [47] did a user study with 31 participants to evaluate the usefulness of a NL-to-code plugin [46]. They found that there was no statistically significant difference in task completion time or task correctness scores when using or not using the NL-to-code plugin. Furthermore, most participants stayed neural or somewhat positive to the NL-to-code plugin. The main reason for these negative results was the correctness and quality of generated code as pointed out by many participants in the post-study survey. However, these findings may not hold as more recent large language models have significantly boosted the correctness and quality of generated code. This further motivates us to conduct the user study with Copilot.

Weisz et al. [44] interviewed 11 software engineers at IBM and solicited their feedback on a neural machine translation (NMT) model for an adjacent domain—translating code from one programming language to another. They found that the user's acceptance of the NMT model was contingent on the number of errors in the translated code. They also identified several common themes in participants' feedback such as acceptance via verification and the desire to provide guidance to the NMT model. Our study was designed to complement this knowledge but for daily programming tasks.

3 STUDY DESIGN

To understand how programmers use an LLM-based code generation tool, we designed and carried out a within-subjects comparative study with 24 participants. For the control condition, each participant was asked to complete a Python programming task in Visual Studio Code (VSCode) IDE with the default code completion tool called Intellisense. Intellisense suggests a drop-down list of valid tokens in the current code context, ordered by alphabetical order or relevance. The users can select the token they want and press the Tab button to accept the suggested token or the Esc button to reject it.

For the experiment condition, each participant finished another Python programming task in VSCode with Copilot. Similar to Intellisene, Copilot can automatically suggest code based on the current code context as a programmer is typing. While Intellisense only predicts one token at a time, Copilot is capable of generating multiple lines of code. The participants can press Tab to accept the code suggestion or Esc to reject. Though not required, participants can give prompts to Copilot by writing comments. Henceforth, when we mention prompts in the text, we refer to comments written by the participants in the code specifically to guide Copilot.

3.1 Tasks

We selected three real-world python programming tasks with different levels of difficulty from [47].

- *Task 1. Edit CSV (Easy)*: Write a program to read CSV data from the 'data.csv' file. Delete the first column and the last column. Save it to the 'output.csv' file.
- *Task 2. Web Scrapping (Medium)*: Given the URL of a web page, write a program that extracts the URLs of all hyperlinks in the web page and save the URLs to a file named 'urls.txt'.
- *Task 3. Graph Plotting (Hard)*: Write a program to draw a scatter plot of the data in 'shampoo.csv' and save it to 'shampoo.png'. The plot size should be 10 inches wide and 6 inches high. The Date column is the x-axis. The date string shown on the plot should be in the YYYY-MM-DD format. The Sales column is the y-axis. The graph should have the title "Shampoo Sales Trend".

3.2 Participants

We recruited 24 participants (4 Female, 19 Male, 1 Non-binary) through mailing lists of two research universities. Ten participants were undergraduate students, 5 were master's student, 8 were Ph.D. students, and 1 was a software engineer. Regarding their familiarity with programming, only 1 participant had less than 2 years of programming experience, 14 participants have 2-5 years of experience, and 9 participants have over 5 years of experience. Participants received a $20 Amazon gift card as compensation for their time.

3.3 Protocol

To enable easy access to the code generation tools, we set up two virtual machines (VMs) in Microsoft Azure, one with Copilot installed and the other with IntelliSense installed. We also pre-installed VSCode and several popular Python packages in both VMs. Participants can easily log into each VM from their laptop to start the user study. We recorded the audio and the screen-cast with the consent of each participant. In each study session, a participant completed one of the three tasks using Copilot (i.e. the experiment condition) and another task with Intellisense (i.e. the control condition). To emulate real-world programming experience, the participants were allowed to use Internet search or refer to any online resources anytime during the task. To mitigate the learning effect, both the order of task assignment and the order of tool assignment was counterbalanced across participants through random assignment. Therefore, for each unique combination of 3 tasks and 2 conditions, we have 8 participant data points. Before each task, the participants were given a quick tutorial of the assigned tool. We set a time limit of 20 minutes for each programming task. A task was considered failed if participants did not complete it within 20 minutes. After each task, participants answered a survey to reflect on their experience using the tool. After finishing both tasks, participants answered a final survey to directly compare the two conditions. The first author performed open-coding on participants' responses to identify themes and then discussed with co-authors to refine the themes over multiple sessions. These themes were then used to explain the results in the following sessions.

4 RESULTS

This section describes both the quantitative and qualitative results of our study. Quantitative results include the task completion time, task failure rates, and metrics from survey responses. In the qualitative results subsection, we describe the common themes that emerged through open coding of participant comments and experimenter observations made during the study.

4.1 Quantitative

Participants using Copilot failed to complete tasks *more* often than participants using Intellisense. Table 1 shows individual and average task completion times. Table cells in the orange background indicate sessions in which participants did not solve the task within 20 minutes. When using Copilot, all 8 participants working on the easiest task completed it, 6 out of 8 participants working on the medium-difficulty task completed it, and 5 out of 8 participants working on the hardest task completed it within the allotted time. In contrast, when using Intellisense, all 8 participants in both the easiest and medium-difficulty task conditions completed their tasks, and only 2 participants failed to complete the hardest task. Overall task difficulties, Intellisense users failed twice while Copilot users failed 5 times. This difference is not statistically significant.

We analyzed the session recordings to identify the root cause of these task failures. Out of the 5 task failures when using Copilot, 3 were caused by incorrect code generated by Copilot, which led participants into a time-consuming debugging rabbit hole (discussed in Section 4.2.4). The other two were caused by the participants' inexperience with the relevant Python libraries (graph plotting and HTML parsing libraries) and the debugging features of the IDE. In contrast, participants using Intellisense failed to finish the 2 tasks due to their inexperience with a graph plotting library.

While Copilot users completed fewer tasks than Intellisense users, the tasks completed with Copilot were done more quickly on average (see the last row of Table 1). The overall mean difference of task completion time using Copilot vs. Intellisense is about 1 min. Yet the mean difference is not statistically significant (student t-test, $p = 0.53$).

	Task 1 - Easy		**Task 2 - Medium**		**Task 3 - Hard**	
	Intellisense	**Copilot**	**Intellisense**	**Copilot**	**Intellisense**	**Copilot**
	9:35	1:46	7:48	12:53	13:41	11:08
	3:50	3:57	15:52	16:45	13:43	11:05
	4:49	4:55	16:28	7:26	22:42	4:04
	9:04	6:18	14:16	15:05	13:06	DNF
	5:18	1:18	7:35	13:24	23:13	19:54
	15:54	7:52	12:39	DNF	4:48	DNF
	5:27	3:12	10:47	6:02	DNF	DNF
	2:09	20:12	8:30	DNF	DNF	9:19
Average Time	**7:01**	**6:11**	**11:44**	**11:56**	**13:36**	**11:06**
Overall average time for all tasks combined					**10:23**	**9:18**

Table 1: Individual and average task completion times. Cells with an orange cell background indicate that the participant never succeeded because they were stopped after approximately 20 minutes of trying. DNF implies the participant did not finish on time.

In the post-study survey, 19 of 24 participants answered that they preferred Copilot over Intellisense. Furthermore, 23 of 24 participants answered that Copilot was more helpful than Intellisense.

We also asked participants to rate the helpfulness of code generated by both tools on a scale of 1 (not at all helpful) to 7 (very helpful). Participants found code generated by Copilot more helpful than code generated by Intellisense (6.16 vs. 4.45 on average). This difference is statistically significant (student t-test: $p < 0.001$). However, only 10 participants self-reported that they felt more confident about the code generated by Copilot than the code suggested by IntelliSense.

4.2 Qualitative

4.2.1 User Perception. Participants found Copilot helpful as it provided a starting point for the task instead of a blank canvas they usually have. Even if the code generated by Copilot is incorrect, it always points them towards a direction they can get started from. P1 said *"Copilot's function/line generation is a helpful reference; even if the generated code is not correct, it can point me in the right direction for completing the task."* This is primarily useful for the kind of tasks in which the user has no experience. P7 said, *"the generation of fully formed functions that completed a task that I wasn't sure how to approach/start was very cool."* For four of the participants, Copilot auto-completed the code for almost the whole tasks, and participants did very little to no fixes to the generated code. Though we did not see any significant difference in task completion time, seven participants explicitly mentioned that Copilot can save time in completing the task compared to Intellisense. P4 said *"[Copilot] will likely save me much more time during the coding process."* Participants also considered writing comments to guide Copilot as a way of communicating with the AI. P24 said *"Copilot behaves just like a TA and can tell me exactly what I want by reading the comments."*

However, participants pointed out several concerns about adopting Copilot in practice. First, twelve participants said they found it hard to understand and change the code generated by Copilot. P1 said, *"Copilot generated a complete function to fulfill the full task, but part of the function did not work as desired. Because I did not understand several parts of the function generated by Copilot, I did not know how to debug the function. This caused me to get rid of the whole function generated by copilot and start over."*. Due to a lack of understanding, five participants perceived a loss of control over their code. P13 said, *"I would go with Intellisense for now since it gives me more control over the code I am writing"*. Second, seven participants expressed concerns over code reliability. P7 said *"At this time I probably prefer Intellisense just because I trust my own googling and understanding code examples online rather than opaque suggestions from copilot."* P18 felt very frustrated after observing Copilot continuously generate code with errors. They said, *"Yes, I got rid of the whole snippet as I didn't want to conform to the code generated by AI as it may have unwanted bugs."* Third, eight participants said they only trusted participants for simple tasks. This is due to multiple reasons, e.g., the difficulty to understand generated code, fear of unknown bugs, failure to match the coding style, etc.

4.2.2 User Interaction Patterns. While prior code completion tools such as Intellisense only suggest one token at a time, Copilot is capable of generating even multiple lines of code at a time. While such a code generation capability is often interpreted as a powerful feature, it causes significant cognitive overload in practice, especially when the generated code has errors. A long piece of generated code forces the user to switch back and forth between program reading and writing. When the generated code has errors, the user needs to further enter into the debugging mode. This constant context switching puts significant mental demand on the users.

Another common interaction pattern is to use Copilot as a substitute for Internet search. P3 said, *"for certain tasks that follow very routine structures, and which I always have to look up on Stack Overflow, a tool like Copilot eliminates a lot of the tedious searching on Google"*. However, we have to note that unlike code examples from Stack Overflow, which are vetted by human programmers, the code generated by Copilot may contain errors. P10 wrote, *"I'm not fully confident that Copilot will suggest the best solution. By reading Stack Overflow, the helpful thing is that there will always be someone who would just post a better solution, and people will discuss and compare. I feel like that is missing from Copilot."* Since Copilot only generates one solution at a time and does not provide any explanations, programmers cannot compare multiple alternative solutions and assess their quality as they often do in an online search.

Furthermore, we observed eight instances of over-reliance on Copilot. For example, P8 simply accepted the generated code and said, *"I guess I will take its word."* This over-reliance also makes participants defer code validation. P20 said *"Not exactly sure what this does. I'll figure it out later"*. Some participants later spot errors in the accepted code. They had to go back and spend a lot of time debugging the previous code.

4.2.3 Coping Strategies. There are two main ways participants cope with incorrect code generated by Copilot. The first way is to accept the incorrect suggestion and attempt to repair it. Twelve participants attempted to repair the code when there was an error. However, the participants always found it difficult to repair the code since the code was not written by themselves. Of these twelve participants open to repair the code, five participants were only willing to repair the code if the code generated by Copilot is easy to read and understand. P7 said, *"it made debugging the code more difficult as I hadn't written the code directly and didn't have an initial intuition about where the bugs might be. Especially with a final bug in my program I really had no idea why it was happening and had to refactor the code."*

In cases where the participant is unable or unwilling to repair the code, they will simply get rid of the entire generated code and search for solutions online. Seven participants mentioned they will rewrite the whole code by themselves without any attempt to repair if there is an error in the code generated by Copilot. P13 said, *"I think getting rid of the whole code is easier than reading the code and making the changes."* P1 also said, *"because I did not understand several parts of the function generated by Copilot, I did not know how to debug the function. This caused me to get rid of the whole function generated by Copilot and start over."*

4.2.4 Obstacles and Limitations. During the user study, we observed three major obstacles to using Copilot in practice. First, participants often failed to understand and assess the correctness of the generated code. Since Copilot often generates a big chunk of code at a time, participants found it hard to understand and debug the code. This is already discussed in Section 4.2.1. The second obstacle is the underestimation of the effort required to fix a bug in the code generated by Copilot. Among the five task failures by participants using Copilot, three were due to incorrect suggestions by Copilot. While participants recognized these errors, they underestimated how much effort it took to fix the bug and got stuck in a debugging rabbit hole they could not get out of. For instance, for P20, Copilot generated a regular expression based code for extracting URLs from HTML. It is extremely hard to get the regular expression right for this task and a better solution is to parse the HTML and extract attributes instead. Since Copilot suggested the regular expression, P20 decided to stick with it and overlooked the better solution. Yet P20 failed to fix the regular expression after 20 minutes, leading to a task failure. The third obstacle is the brittleness and ambiguity of using comments (or prompts) as a specification for Copilot. As discussed in the previous sections, participants used comments to describe the desired code that should be generated by Copilot. However, Copilot is very sensitive to these comments. A little tweak in a comment can cause Copilot to generate a significantly different code snippet. P24 said, *"it is ambiguous to use comments to hint at Copilot what I want."*

5 DISCUSSION

The majority of participants (19 out of 24) expressed a strong preference to use Copilot for their day-to-day programming tasks for several reasons. In many cases, Copilot accurately generated the code from the prompts provided by the participants. In four instances, it even generated the correct code for almost the whole task in one shot. Generating a whole block of code improves developer productivity significantly. However, we did not see a big difference in the time saved by Copilot during the study. Our observations point to a plausible explanation for this non-significance—though it is faster to generate code through Copilot compared to acquiring code from the internet, the code generated by Copilot can be buggy, leading to more time spent in debugging. Whereas, code from the internet is generally bug-free, comes with explanations and discussion, and can be composed suitably for the current task by just doing some minor edits like changing the variable names. Moreover, Copilot also provides a useful starting point for the users to get started, even if the generated code was incorrect. This is especially useful for users who are stuck in a problem or who do not know how to approach the task. Several participants request to see multiple code suggestions so they can compare and compose code from different snippets to suit their needs. Furthermore, we found participants used Copilot as a replacement for internet search. However, they missed out on comparing multiple sources and community discussions. Hence, it is worthwhile integrating online search with code generation to help users compare AI-generated code with online code examples and identify the best possible solution for a task. This can also prevent users from getting trapped in a debugging rabbit hole whenever Copilot suggests an incorrect or inefficient solution.

Another observation that is worth investigating is that participants had a hard time understanding the code generated by Copilot. One way to help users understand the generated code is to provide explanations using inline comments. We can highlight different parts of the code based on model confidence similar to the approach suggested by [44]. We can also help users debug code by automatically generating test cases and test data for users to validate generated code and identify corner cases. We would like to study this in-depth and come up with ways to make the code more understandable and help users to debug and repair generated code. Moreover, we observed that Copilot led to more task failures in medium and hard tasks since it was hard for Copilot to generate correct code in one shot. Three participants who finished the hard task approached the problem by decomposing the complex task into simpler sub-tasks and wrote prompts for each sub-task for Copilot to solve. Such a task decomposition strategy led to higher task-solving efficiency and a better user experience. Therefore, it is worth working on interaction mechanisms that facilitate task decomposition in the future.

6 CONCLUSION

This paper presents a user study with 24 participants on the usability of GitHub Copilot, a groundbreaking code generation tool empowered by an ultra-large language model. In particular, we investigated users' perception of Copilot, their interaction patterns, and their coping strategies when the generated code is not correct.

We found that, despite all the promising results on benchmarks [8], Copilot did not necessarily reduce the task completion time or increase the success rate of solving programming tasks in a real-world setting. On the other hand, participants overwhelmingly preferred using Copilot in their programming workflow since Copilot often provided a good starting point to approach the programming task. Furthermore, our study shed light on several promising future directions for improving the design of Copilot. For example, instead of simply using Copilot as a one-shot code generation tool, there should be more support for understanding and validating the generated code, exploring multiple solutions, and task decomposition.

ACKNOWLEDGMENTS

This material is based upon work supported by the NSF under Grant No. IIS-2107391 and Grant No. CCF-2123965.

REFERENCES

[1] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. 2020. Structural Language Models of Code. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*, Hal Daumé III and Aarti Singh (Eds.). PMLR, 245–256. https://proceedings.mlr.press/v119/alon20a.html

[2] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. *Syntax-guided synthesis*. IEEE.

[3] Stavros Antifakos, Nicky Kern, Bernt Schiele, and Adrian Schwaninger. 2005. Towards Improving Trust in Context-Aware Systems by Displaying System Confidence *(MobileHCI '05)*. Association for Computing Machinery, New York, NY, USA, 9–14. https://doi.org/10.1145/1085777.1085780

[4] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. *ArXiv* abs/2108.07732 (2021).

[5] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. *ArXiv* abs/1611.01989 (2017).

[6] Sid Black, Leo Gao, Phil Wang, Connor Leahy, and Stella Biderman. 2021. *GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow.* https://doi.org/10.5281/zenodo.5297715 If you use this software, please cite it using these metadata..

[7] Sarah E Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping distributed hierarchical web data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. 963–975.

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[9] Matteo Ciniselli, Nathan Cooper, Luca Pascarella, Denys Poshyvanyk, Massimiliano Di Penta, and Gabriele Bavota. 2021. An Empirical Study on the Usage of BERT Models for Code Completion. *arXiv preprint arXiv:2103.07115* (2021).

[10] Allen Cypher. 1995. Eager: Programming repetitive tasks by example. In *Readings in human–computer interaction*. Elsevier, 804–810.

[11] Mary T Dzindolet, Scott A Peterson, Regina A Pomranky, Linda G Pierce, and Hall P Beck. 2003. The role of trust in automation reliance. *International journal of human-computer studies* 58, 6 (2003), 697–718.

[12] John K Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices* 50, 6 (2015), 229–239.

[13] Github Copilot [n. d.]. Your AI pair programmer.

[14] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330.

[15] Sumit Gulwani, José Hernández-Orallo, Emanuel Kitzelmann, Stephen H Muggleton, Ute Schmid, and Benjamin Zorn. 2015. Inductive programming meets the real world. *Commun. ACM* 58, 11 (2015), 90–99.

[16] Tong Guo and Huilin Gao. 2019. Content enhanced bert-based text-to-sql generation. *arXiv preprint arXiv:1910.07179* (2019).

[17] Shirley Anugrah Hayati, Raphael Olivier, Pravalika Avvaru, Pengcheng Yin, Anthony Tomasic, and Graham Neubig. 2018. Retrieval-based neural code generation. *arXiv preprint arXiv:1808.10025* (2018).

[18] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[19] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. IEEE, 215–224.

[20] Rafael-Michael Karampatsis and Charles Sutton. 2019. Maybe deep neural networks are the best choice for modeling source code. *arXiv preprint arXiv:1903.05734* (2019).

[21] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 150–162.

[22] Kite - Free AI Coding Assistant and Code Auto-Complete Plugin 2020. Kite - Free AI Coding Assistant and Code Auto-Complete Plugin. https://www.kite.com/. Accessed: 2022-1-8.

[23] Rafal Kocielnik, Saleema Amershi, and Paul N Bennett. 2019. Will you accept an imperfect ai? exploring designs for adjusting end-user expectations of ai systems. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–14.

[24] Tessa Lau, Steven A Wolfman, Pedro Domingos, and Daniel S Weld. 2003. Programming by demonstration using version space algebra. *Machine Learning* 53, 1 (2003), 111–156.

[25] Vu Le and Sumit Gulwani. 2014. Flashextract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 542–553.

[26] Brian Y Lim and Anind K Dey. 2009. Assessing demand for intelligibility in context-aware applications. In *Proceedings of the 11th international conference on Ubiquitous computing*. 195–204.

[27] Brian Y Lim and Anind K Dey. 2010. Toolkit to support intelligibility in context-aware applications. In *Proceedings of the 12th ACM international conference on Ubiquitous computing*. 13–22.

[28] Brian Y Lim, Anind K Dey, and Daniel Avrahami. 2009. Why and why not explanations improve the intelligibility of context-aware intelligent systems. In *Proceedings of the SIGCHI conference on human factors in computing systems*. 2119–2128.

[29] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 336–347.

[30] Brad A Myers. 1990. Creating user interfaces using programming by example, visual programming, and constraints. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 2 (1990), 143–177.

[31] Brad A Myers. 1991. Graphical techniques in a spreadsheet for specifying user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 243–249.

[32] OpenAI and Ashley Pilipiszyn. 2021. GPT-3 Powers the Next Generation of Apps. https://openai.com/blog/gpt-3-apps/. Accessed: 2022-1-8.

[33] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2021. Can OpenAI Codex and Other Large Language Models Help Us Fix Security Bugs? *arXiv preprint arXiv:2112.02125* (2021).

[34] Hila Peleg, Sharon Shoham, and Eran Yahav. 2018. Programming not only by example. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 1114–1124.

[35] Paul Robinette, Wenchen Li, Robert Allen, Ayanna M Howard, and Alan R Wagner. 2016. Overtrust of robots in emergency evacuation scenarios. In *2016 11th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*. IEEE, 101–108.

[36] Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. 2005. Programming by sketching for bit-streaming programs. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 281–294.

[37] Simone Stumpf, Vidya Rajaram, Lida Li, Weng-Keen Wong, Margaret Burnett, Thomas Dietterich, Erin Sullivan, and Jonathan Herlocker. 2009. Interacting meaningfully with machine learning systems: Three experiments. *International journal of human-computer studies* 67, 8 (2009), 639–662.

[38] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. Treegen: A tree-based transformer architecture for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 8984–8991.

[39] Tabnine [n. d.]. Code Faster with AI Code Completions. https://www.tabnine.com/. Accessed: 2022-1-8.

[40] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 4 (2019), 1–29.

[41] Richard J Waldinger and Richard CT Lee. 1969. PROW: A step toward automatic program writing. In *Proceedings of the 1st international joint conference on Artificial intelligence*. 241–252.

[42] Ben Wang. 2021. Mesh-Transformer-JAX: Model-Parallel Implementation of Transformer Language Model with JAX. https://github.com/kingoflolz/mesh-

transformer-jax.

[43] Chenglong Wang, Yu Feng, Rastislav Bodik, Alvin Cheung, and Isil Dillig. 2019. Visualization by example. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–28.

[44] Justin D Weisz, Michael Muller, Stephanie Houde, John Richards, Steven I Ross, Fernando Martinez, Mayank Agarwal, and Kartik Talamadupula. 2021. Perfection Not Required? Human-AI Partnerships in Code Translation. In *26th International Conference on Intelligent User Interfaces*. 402–412.

[45] Daniel S Weld and Gagan Bansal. 2018. Intelligible artificial intelligence. *ArXiv e-prints, March 2018* (2018).

[46] Frank F Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig. 2020. Incorporating external knowledge through pre-training for natural language to code generation. *arXiv preprint arXiv:2004.09015* (2020).

[47] Frank F Xu, Bogdan Vasilescu, and Graham Neubig. 2021. In-IDE Code Generation from Natural Language: Promise and Challenges. *arXiv preprint arXiv:2101.11149* (2021).

[48] Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696* (2017).

[49] Pengcheng Yin and Graham Neubig. 2018. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. *arXiv preprint arXiv:1810.02720* (2018).

[50] Wojciech Zaremba, Greg Brockman, and OpenAI. 2021. OpenAI Codex. https://openai.com/blog/openai-codex/. Accessed: 2022-1-8.

[51] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L Glassman. 2020. Interactive Program Synthesis by Augmented Examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 627–648.

[52] Yunfeng Zhang, Q Vera Liao, and Rachel KE Bellamy. 2020. Effect of confidence and explanation on accuracy and trust calibration in AI-assisted decision making. In *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*. 295–305.