"If it's what I wanted that's great, but if it's not, I just wasted time": Examining the perceived costs/benefits of ML-enhanced developer tooling.

Ambar Murillo, Sarah D'Angelo, Quinn Madison, Iris Chu, and Andrew Macvean
Google
{ambarm; sdangelo; qmadison; irischu; amacvean}@google.com

Abstract

ML-enhanced software development tooling is changing the way engineers write code. Even though development of such tools has accelerated, studies have primarily focused on the accuracy and performance of underlying models, rather than user experience. Recently however, researchers have begun to explore how developers interact with ML-enhanced tooling to better understand impact on workflows. We investigate what factors influence developers' decision to interact with ML-based suggestions by analyzing the perceived benefits and costs of ML-based assistance and then, compare developers' experiences with different prototypes. Our findings suggest that developers engage in a form of ad-hoc cost-benefit analysis mediated by their familiarity with the task and the complexity of the suggestion. We close by providing design guidance aimed at increasing the perceived benefits and lowering costs. This research is intended to spark a growing conversation about what makes successful interactions with ML-based software development tooling.

1 Introduction

Machine Learning (ML) enhanced software development tooling is changing the way software developers write code (Mozannar et al., 2022; Bader et al., 2021). ML enhancement aims to improve the efficiency and productivity of software developers by 1) reducing developer workloads; 2) reducing task time; and 3) mitigating human errors. ML enhancements appear throughout the development life-cycle and use various ML technologies and approaches (e.g., from smart code auto-complete, predicting future bugs, to personalized programming education) (Sorte et al., 2015). Development of such tools has accelerated, yet accompanying studies have primarily focused on the accuracy and performance of underlying models, rather than user experience.

Recently, researchers have started to explore how developers interact with ML enhanced developer tools and how it impacts developer workflows (Ziegler et al., 2022; Barke et al., 2022; Vaithilingam et al., 2022). These studies highlight the need to further understand the experience of developers who work with ML-based assistance, to create tooling that is useful and integrated into workflows. Research shows that developers feel more productive when using ML-based assistance, even if they aren't always faster at producing code (Vaithilingam et al., 2022). Understanding this aspect of the user experience provides insight into the potential impact of ML-based tooling on developer productivity. There is still more to understand however. For example, what do developers take into consideration before they accept ML-based suggestions?

In this research, we investigate what factors influence developers' decision to interact with ML-based suggestions by first examining the perceived benefits and costs of having ML-based assistance in their developer tooling and second, comparing their experiences with different prototypes used for hypothetical coding tasks. Using different prototypes and prompts for various tasks allowed us to gather a wide range of feedback. Our findings suggest that developers engage in a form of ad-hoc cost-benefit analysis mediated by their familiarity with the task and the complexity of the suggestion, which informs how they interact with suggestions. We then provide design guidance aimed at increasing the perceived benefits and lowering the perceived costs.

Our research goals:

- Identify perceived benefits and costs of ML-based assistance in developer tooling
- Describe factors which mediate developers' decision to engage with ML-based assistance
- Define developers' decision making process for engaging with ML-based assistance
- Leverage the insights gained to highlight design guidance

We outline the different types of evaluations in the literature: ML model performance (focused on the ML model), behavioral online metrics (focused on usage), and user interaction (focused on user interactions with the ML-based assistance). We highlight a gap in the literature, and where our contribution lies: understanding how developers decide to engage with ML-based assistance. Then, we describe the methods, results, and implications of our work. This paper is intended to continue a growing conversation about the future of ML-based assistance in software development and provide further guidance on the design of this tooling.

2 RELATED WORK

The development of ML-enhanced software development tooling is a growing field of research and development (Bader et al., 2021). Many companies have recently developed code generation tools including Copilot from Github (Friedman, 2021), AlphaCode from Alphabet's DeepMind (Li et al., 2022), and CodeWhisperer from Amazon (Amazon, 2022). Other companies have also started developing ML-enhanced software development tooling for internal use, such as Meta (Bader et al., 2021) and Google (Tabachnyk & Nikolov, 2022). While these tools have the potential to integrate into many stages of the software development life cycle (Sorte et al., 2015; Bader et al., 2021), there has been a focus on the implementation stage (including coding and building) (Bader et al., 2021; Sorte et al., 2015).

As the availability of ML-enhanced developer tooling has grown, so has the interest and need to evaluate its success factors. One of these factors is ML model performance: measuring the accuracy of the model's predictions (eg., if the prediction exactly matches a ground truth dataset) (Garg et al., 2022), or their functional correctness (eg., an output is deemed correct if it passes unit tests) (Chen et al., 2021). Other types of model performance evaluations include quality assessments. For example, if the ML-based suggestions are perceived as useful (Beller et al., 2022); or if they are semantically similar to a ground truth dataset, even if not a verbatim match (Garg et al., 2022). These evaluations typically include expert human evaluators to judge these factors.

Another type of evaluation of ML-based assistance looks at its usage and impacts with behavioral online metrics. For example, in the case of code completion, these metrics may include daily completions per user (DCPU), the raw number of accepted suggestions per developer per day (Zhou et al., 2022). Some research has gone further, examining the impact to developer productivity of accepting code completion suggestions, by looking at coding iteration time (Tabachnyk & Nikolov, 2022).

While evaluations focusing on model performance and online behavioral metrics are important; they don't account for user experience: how developers feel about ML-based assistance, how it fits into their workflows and task context, and how they decide whether or not to interact with it. If we leave these topics unexamined, we are by default assuming that if a prediction is accurate, developers will want to engage with ML-based assistance. However, this may not be the case. User experience is an understudied field of ML-enhanced tooling research, and where our contribution lies: understanding how developers decide to engage with ML-based assistance.

Ziegler et al. (2022) have explored developers' perceptions of productivity, and how they relate to online behavioral metrics such as acceptance and persistence. The authors note that ML-based assistance's "...central value lies not in being the way the user enters the highest possible number of lines of code. Instead, it lies in helping the user to make the best progress towards their goals. A suggestion that serves as a useful template to tinker with may be as good or better than a perfectly correct (but obvious) line of code that only saves the user a few keystrokes. This suggests that a narrow focus on the correctness of suggestions would not tell the whole story for these kinds of tooling.” This quote highlights the importance of looking at developer interactions with ML-enhanced tooling, to better evaluate the tool’s success and its impact on developer workflows. The importance of looking at developer interactions and sentiment has also been echoed in other research (Barke et al., 2022; Vaithilingam et al., 2022; Mozannar et al., 2022).

Barke et al. (2022) have looked at how developers interact with Copilot, and identified two modes of working with ML-based assistance: acceleration (where the developer used Copilot to execute planned actions, staying in flow, with a focus on accelerating their development) and exploration (where the developer used Copilot to help plan next steps and explore possible paths forward). Vaithilingam et al. (2022) have also conducted research on how developers perceive and use Copilot, finding that while Copilot did not necessarily improve metrics like task-completion time, and at times introduced difficulties (eg., some participants had difficulties understanding or debugging the code introduced by Copilot); study participants still preferred to use it because it provided “a good starting point.”

These prior studies start to shed light on developer perceptions towards ML-enhanced developer tooling. However, they overlook questions such as: What happens before the developer starts the interaction (eg., before they accept a suggestion)? What is the decision making process that leads developers to engage actively with assistance? Mozannar et al. (2022) have highlighted similar questions, and conducted a study to explore how participants interacted with Copilot. The authors produced a taxonomy of “common programmer activities” (eg., accepting or viewing suggestions), combined with telemetry, to gain a better understanding of these interactions. For example, authors ask what developers do before accepting a suggestion, finding that if the developer verified a suggestion, the probability of accepting it increased. However, if the developer was thinking about new code to write, the probability of accepting a suggestion dropped. The findings also highlight that the cost of the interaction with this ML enhanced developer tooling was higher than previously expected, because developers tended to double-check or edit suggestions. Mozannar et al. (2022) present a way to operationalize the developer’s decision making process, and understand the interactions with ML-based assistance. This research complements ours: while Mozannar et al examine behaviors (eg., when code suggestions were accepted, what happened around those events), our paper defines developers’ decision making process for engaging with ML-based assistance. For example, when developers didn’t interact with ML-based assistance, why was that the case?

To understand which factors mediate developers’ decision to engage with ML-based assistance, we conducted an interview study with developers as they interacted with two prototypes across two hypothetical task scenarios. We first describe our methods, then share our findings, follow with a discussion of developers’ decision making process, and leverage these insights to highlight design guidance.

3 METHODOLOGY

3.1 STUDY DESIGN

We conducted sixty-minute remote-moderated semi-structured interviews. The interviews contained two parts. First, participants were asked about their perceptions of ML-enhanced software development tooling in general. Second, participants were asked to complete two hypothetical tasks using two different prototypes (1) multi-line code completion and (2) code examples snippets. These prototypes were chosen to illustrate different levels of complexity (e.g. code examples provide more code that need to be altered) for the same task (writing code). After each scenario, participants were asked questions about their experience with the tooling. Themes explored in the interviews included: familiarity with ML-enhanced developer tooling, expectations, trust, and desired level of control. The interviews were transcribed and two researchers conducted a thematic analysis using inductive coding Braun & Clarke (2012). Themes were identified and clustered together, to shed light on how participants’ experience with the two prototypes differed.

3.2 PARTICIPANTS

Ten full time professional software developers at a large technology company. Participants had an average tenure of 4 years at the company and reported (every participant had more than 6 months tenure). All participants reported their primary coding language as Go. Before the interview, participants gave written consent.

3.3 Prototypes

Multi-line Code Completion: The first prototype was a multi-line code completion feature in the IDE, of a relatively high fidelity, and similar to the multiple line completion interaction described in this blog post (Tabachnyk & Nikolov, 2022). The multi-line code completion prototype provided code suggestions for the next few lines of code. Therefore, to complete the hypothetical coding tasks, participants had to read the suggestions, evaluate their fit, and then make decisions about whether to accept or reject, or if to accept the suggestion and then edit it further. Each of these decisions would impact several lines of code. Participants had prior experience with this type of prototype in the context of their daily work.

Code Examples Snippets: The second prototype provided participants with examples of code snippets from their code base that could be in the UI within the IDE. These snippets were relevant to the code they were writing. This prototype used a "Wizard of Oz" method (Green & Wei-Haas, 1985) to share code examples snippets with developers that could have been suggested by an ML-based assistance. The code examples snippets prototype provided suggestions of entire snippets of code. Although the steps of reading the suggestions and evaluating their fit is similar to the multi-line prototype, the code suggestions being evaluated are longer (e.g., blocks of code) and typically from files not familiar to the user, making this ML-based interaction relatively more complex than the interaction with the code completion prototype. Participants had no prior experience with this prototype.

3.4 Tasks

Participants were asked to complete two hypothetical coding tasks that used existing code from the participant's code base. For each task, the prototypes displayed two "successful" outcomes (displaying useful suggestions), and two "failure" outcomes (displaying incorrect or not useful suggestions). The coding tasks were all a similar level of difficulty, while the prototypes varied in complexity.

Code generation tasks were written in the Go programming language, which all participants reported as their primary language. For the code examples snippets prototype, one task was written in Java, a language participants were less familiar with, to provide a comparison based on familiarity with the language.

4 Findings

4.1 Research Goal 1: Identify the primary perceived benefits and costs of ML-based assistance in developer tooling

These findings reflect participants' general perceptions of the benefits and costs of working with ML-enhanced developer tooling. Importantly, these reactions were captured before participants interacted with specific prototypes.

4.1.1 Perceived Benefits

In general, participants were excited about having ML-based assistance in their development workflows, but did not have a lot of experience with a wide range of ML-enhanced developer tooling. Code completion was the most common type of ML enhanced developer tooling that participants were aware of or had interacted with, but participants were asked to reflect on ML based assistance broadly.

Overall, participants felt the benefits of ML-enhanced development tooling to be: saving time and effort, for example, by getting more repetitive or "boilerplate" type tasks out of the way, allowing the developer to focus more on the logic behind the coding task. As described by several participants: *"boilerplate just flows from the machine rather than having to type it all"*.

Other benefits included learning new languages, error prevention, discoverability, and improving code quality (through proliferation of best practices, for example). One participant mentioned that increased speed was not the most important factor and said *"When the suggestions and the assistance can get ahead of where I'm thinking and allow me to think faster, that to me is interesting. It's not the typing speed that is really the bottleneck for most software developers, it's the ability to really think things through"*.

4.1.2 PERCEIVED RISKS

Participants perceived the costs to be: over-reliance on tools, ML models imposing standards on code that prevent evolution or encourage complexity. One participant said: *"the biggest risk is blindly trusting the machine, and ending up with things that are way more complicated than they have to be"*.

Participants also mentioned the inability to gracefully recover from errors suggested by ML-based assistance. For example, a case where users accept an ML-based suggestion, then make edits , then accept more ML based suggestions (in different places in a file), and then undo some edits. A participant describes a similar situation: *"I think it's sort of this double-edged sword, right? Where the better your suggestions get, the more likely you are to accept them quickly, and then when they are wrong, you've entered a bug and you haven't fully read it. And you've got to go back and scrutinize your code, as if somebody else had written it, because in a way someone else did write it."*

4.2 RESEARCH GOAL 2: DESCRIBE FACTORS WHICH MEDIATE DEVELOPERS' DECISION TO ENGAGE WITH ML-BASED ASSISTANCE

As participants completed each task, they were asked to think aloud and comment on their interaction with the prototypes and their decision to accept, accept and edit, or reject the suggestions; as well as how they felt about the ML based assistance for that type of task. Two factors clearly impacted developers' interactions with the prototypes: (1) complexity of both the task and the suggestion and (2) familiarity with the code (e.g. language or task).

4.2.1 COMPLEXITY

Overall, participants felt more comfortable with the code completion prototype, especially for straightforward tasks (e.g. writing tests), because they perceived the benefits (saving time) to clearly outweigh the costs (introducing an error). For example, one participant commented: *"I felt more trust because it was doing a simple more structural task"*. This demonstrates that the complexity of the task influences how developers feel about the ML-based suggestions, with simpler tasks being perceived as lower cost.

Participants were less comfortable with the code examples snippets prototype because it was perceived as higher cost. One participant said: *"my first thought is 'oh wow that's too much', I don't want to read all of that... if it's what I wanted that's great, but if it's not, I just wasted time"*. The increased amount of text was perceived as a high cost and participants were unsure if the time spent would be beneficial.. Since participants were less familiar with this type of assistance, their reactions could be related to limited experience. However, their behavior suggests that the complexity of the suggestion, is an important factor that influences developers' trust in ML-based assistance.

4.2.2 FAMILIARITY

While participants were shown most scenarios in their primary language, one scenario for similar code examples used a language (Java) unfamiliar to participants . This scenario revealed that familiarity with a language plays a critical role in how participants interact with suggestions. One participant said: *"I'm not sure if any of these are helpful, I'm too unfamiliar with the language."* This response highlights that while ML-based assistance can be helpful, developers need baseline knowledge of the language to determine if the suggestion is correct or helpful.

In addition to familiarity with the language, participants also expressed that familiarity with the ML based assistance influences how likely they are to engage with it. For example: *"you would get to know how accurate it is, and if every time it is always the right chunk of code, you would naturally*

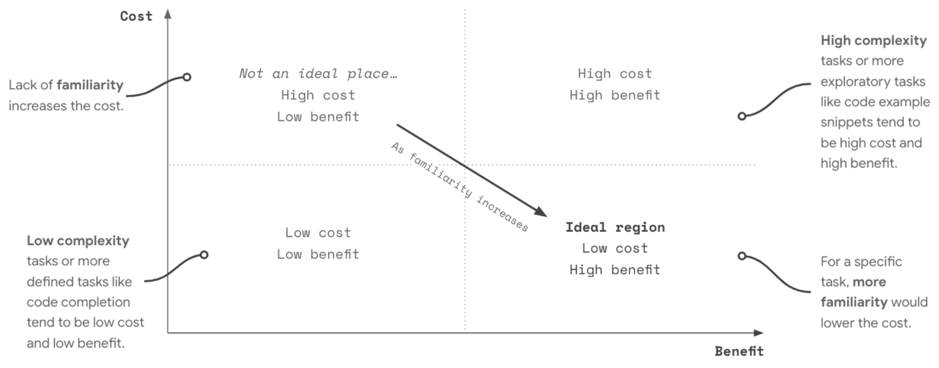
Cost
Lack of **familiarity** increases the cost.
Not an ideal place…
High cost
Low benefit
High cost
High benefit
High complexity tasks or more exploratory tasks like code example snippets tend to be high cost and high benefit.
As familiarity increases
Low complexity tasks or more defined tasks like code completion tend to be low cost and low benefit.
Low cost
Low benefit
Ideal region
Low cost
High benefit
For a specific task, **more familiarity** would lower the cost.
Benefit

Figure 1: Cost Benefit Analysis

build up trust". Based on these findings, we discuss how developers consider the costs and benefits of engaging with ML-based tools in the context of the task at hand.

5 Discussion

5.1 Research Goal 3: Define developers' decision making process for engaging with ML-based assistance

Participants engaged in cost-benefit analyses to determine when to accept ML generated suggestions, which was mediated by the complexity of the task and suggestion, as well as their familiarity with the task and ML-based assistance. Figure 1 illustrates this cost-benefit analysis by displaying increase in benefit on the x-axis (e.g. the cost associated with accepting an incorrect suggestion) and the increase in cost on the y-axis (e.g. time saved by accepting a correct suggestion or improvements to the code quality). The quadrants illustrate how familiarity and the complexity of the interaction influences whether a suggestion is perceived as high/low cost and high/low benefit.

Familiarity: A lack of familiarity increases the cost and decreases the benefit (upper left quadrant) of a suggestion because without sufficient knowledge of the language or the code base it is difficult and time intensive for developers to discern good or bad suggestions. Whereas, increasing familiarity decreases the cost and increases the benefit (lower right quadrant) because good suggestions are quick and easy to identify.

Complexity: Low task complexity decreases the cost because simple and more defined tasks are easier to identify correct suggestions for. However low task complexity also decreases the benefit because these tasks are not as time consuming and ML based assistance doesn't always save time (lower left quadrant). Whereas, high complexity increases the cost and increases the benefit because correct solutions become harder to identify, complex tasks take longer so the opportunity for time savings increases (upper right quadrant).

For example, when a developer is provided with code completion for a boilerplate test, they are likely to accept it because the cost is low. However, the benefit in that particular example is also low, so we would only expect to see small gains in performance or improved code quality due to saved time and lack of human error. It is important to note that we are using developers' perceived benefits which do not include their satisfaction with the experience of having ML-based assistance. Prior work has shown this to be an important factor in perceived productivity Vaithilingam et al. (2022) and we discuss the implications of this in the Future Work section.

As we think about the future of ML-enhanced developer tooling, it is important that we aim to make suggestions as low cost and high benefit as possible. Therefore, taking into consideration complexity and familiarity gives researchers and designers the ability to create a user experience that optimizes for low cost and high benefit interactions. Based on cost-benefit analysis described above, we define design guidance for ML enhanced developer tooling.

5.2 Research Goal 4: Leverage insights to highlight design guidance

When designing ML-based assistance for developer tooling, we recommend first: follow a principle-based approach, one that enables user control and builds user trust over time. Then: know your users and consider the context (of their task) so that the system can surface ML-assistance that's on-task in any given moment. Consider questions such as: how complex is the task and the ML-proposed suggestions? Is the task exploratory or well-defined? How knowledgeable are the users? These questions will help determine where the ML-based assistance lies on the cost/benefit matrix.

For example, designing a tool that helps novice developers write complex functions by providing in-IDE documentation would be high cost, high benefit because familiarity is low and complexity is high. To decrease the cost and increase the benefit, we recommend leveraging UX design patterns. UX design patterns provide familiar and consistent ways for users to interact with ML-enhanced tooling. Based on our research and the PAIR framework (Google, 2021), we outline a subset of UX design considerations aimed at highlighting benefits and decreasing perceived cost.

5.2.1 Highlight Benefits

While the costs of interacting with ML-enhanced developer tooling (e.g. time it takes to read a suggestion) may be apparent, this is not always the case for understanding the benefits, particularly for tooling unfamiliar to developers. Even if an ML-based suggestion meets certain model performance thresholds, the relevance may not be immediately obvious. Therefore, use design patterns to highlight benefits. For example, design the content hierarchy so that benefits are easily understood, and provide contextual information from trusted sources to help developers judge suggestions (eg., ratings from other developers). Additionally, provide cues—such as the source, or where in the code the suggestion would apply—so developers can understand why a suggestion is relevant.

When benefits are well understood and perceived as high, users will tolerate features that use a large portion of their screen (which would otherwise be perceived as a cost, as it blocks visibility of code or other important information).

5.2.2 Lowering Cost

As ML-based developer tooling becomes more prevalent, we can leverage developers' familiarity to lower the cost of interacting with new forms of ML-based assistance. For example, build on existing mental models, and use familiar UX design patterns to guide the design of new ML-enhanced features (e.g, graying out predictive text). Such patterns help developers quickly identify the benefits and make decisions on how to engage with the tooling. Providing UX design patterns that allow developers to accept several high quality suggestions at a time, preview the impact of suggested changes, and enable developers to easily undo actions and revert to previous states, will lower perceived cost.

When the cost is perceived as low, developers will likely tolerate disruptive patterns such as a code completion pattern that appears unprompted onscreen. The cost of an unprompted code completion window appearing on screen is low, particularly since code completion is a familiar pattern; and the benefit of accepting a suggestion is high.

5.3 Limitations

The number of prototypes used represent only a small range of potential ML-based assistance suggestions and tooling. The difference in fidelity of the prototypes is another factor that could have influenced participants' perceptions. The code examples snippets prototype used a "Wizard of Oz" technique which provided less flexibility for developers to explore scenarios with ML-based assistance and does not reflect real world scenarios as closely. However, our goal was to elicit reactions from participants rather than measure task completion or code quality, or provide an extensive overview of perceptions based on different types of tooling. Additionally, recent research has leveraged "scenario-based design" Rosson & Carroll (2009), successfully employing a similar method in the context of understanding the role of ML in code translation Weisz et al. (2021). Furthermore, having different levels of fidelity helped us gather a wide range of feedback.

6 Conclusion and Future Work

ML-enhanced software development tooling will continue to shape the future of engineering workflows. Therefore, it is critical we continue to invest in learning how and why developers interact with these tools and better understand how to build them. In this work, we build upon prior literature, by investigating developers' perceptions towards ML-based tooling and their decision making process before they accept an ML-based suggestion. Based on these findings we outline a cost-benefit analysis framework that is mediated by familiarity and complexity. Additionally, we discuss how to leverage these findings into concrete design recommendations for future tools. This research is intended to continue a growing conversation about the future of assistance in software development.

An important area for future research and conversations involves determining what constitutes successful interactions with ML based assistance. Historically, research has largely focused on the performance of the models as success. Recent research however, examines developers' perceptions and sentiment towards assistance to better understand and shape interactions. Our research builds on that work by looking at success from the developers' perspective and defining interactions in terms of costs and benefits. Our work provides another factor of success to consider as ML based tooling continues to evolve.

This work reveals how developers' perceived costs influence the way they interact with ML based assistance. One question that would benefit from further research is how assistance impacts creativity in development. How can models evolve with code (e.g. style changes, novel solutions)? How do developers navigate the tension between working efficiently with ML and continuing to produce novel code? We hope our work sparks interest in investigating many aspects of the user experience with ML enhanced developer tooling.

Acknowledgments

We would like to thank Daniel Berlin, Maxim Tabachnyk, Collin Green, and Eunice Chang for their review of this work, Alberto Elizondo for his work on the prototypes, and our participants for sharing their insights and experiences.

References

Amazon. Ml-powered coding companion for developers - amazon codewhisperer features, 2022. URL `https://aws.amazon.com/codewhisperer/features/`.

Johannes Bader, Sonia Seohyun Kim, Frank Sifei Luan, Satish Chandra, and Erik Meijer. Ai in software engineering at facebook. *IEEE Software*, 38(4):52–61, 2021. doi: 10.1109/MS.2021.3061664.

Shraddha Barke, Michael B James, and Nadia Polikarpova. Grounded copilot: How programmers interact with code-generating models. *arXiv preprint arXiv:2206.15000*, 2022.

Moritz Beller, Hongyu Li, Vivek Nair, Vijayaraghavan Murali, Imad Ahmad, Jürgen Cito, Drew Carlson, Ari Aye, and Wes Dyer. Learning to learn to predict performance regressions in production at meta. *arXiv preprint arXiv:2208.04351*, 2022.

Virginia Braun and Victoria Clarke. *Thematic analysis.* American Psychological Association, 2012.

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. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Nat Friedman. Introducing github copilot: Your ai pair programmer, Jun 2021. URL `https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer/`.

Spandan Garg, Roshanak Zilouchian Moghaddam, Colin B Clement, Neel Sundaresan, and Chen Wu. Deepdev-perf: a deep learning-based approach for improving software performance. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 948–958, 2022.

Google. People + ai guidebook, 2021. URL `https://pair.withgoogle.com/guidebook`.

Paul Green and Lisa Wei-Haas. The rapid development of user interfaces: Experience with the wizard of oz method. In *Proceedings of the Human Factors Society Annual Meeting*, volume 29, pp. 470–474. SAGE Publications Sage CA: Los Angeles, CA, 1985.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. Reading between the lines: Modeling user behavior and costs in ai-assisted programming. *arXiv preprint arXiv:2210.14306*, 2022.

Mary Beth Rosson and John M Carroll. Scenario-based design. In *Human-computer interaction*, pp. 161–180. CRC Press, 2009.

Bhagyashree W Sorte, Pooja P Joshi, and Vandana Jagtap. Use of artificial intelligence in software development life cycle—a state of the art review. *International Journal of Advanced Engineering and Global Technology*, 3(3):398–403, 2015.

Maxim Tabachnyk and Stoyan Nikolov. Ml-enhanced code completion improves developer productivity, 2022. URL `https://ai.googleblog.com/2022/07/ml-enhanced-code-completion-improves.html`.

Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *CHI Conference on Human Factors in Computing Systems Extended Abstracts*, pp. 1–7, 2022.

Justin D Weisz, Michael Muller, Stephanie Houde, John Richards, Steven I Ross, Fernando Martinez, Mayank Agarwal, and Kartik Talamadupula. Perfection not required? human-ai partnerships in code translation. In *26th International Conference on Intelligent User Interfaces*, pp. 402–412, 2021.

Wen Zhou, Seohyun Kim, Vijayaraghavan Murali, and Gareth Ari Aye. Improving code autocompletion with transfer learning. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '22, pp. 161–162, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392266. doi: 10.1145/3510457.3513061. URL `https://doi.org/10.1145/3510457.3513061`.

Albert Ziegler, Eirini Kalliamvakou, X Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pp. 21–29, 2022.