Grounded Copilot: How Programmers Interact with Code-Generating Models

SHRADDHA BARKE*, UC San Diego, USA
MICHAEL B. JAMES*, UC San Diego, USA
NADIA POLIKARPOVA, UC San Diego, USA

Powered by recent advances in code-generating models, AI assistants like Github Copilot promise to change the face of programming forever. But what *is* this new face of programming? We present the first grounded theory analysis of how programmers interact with Copilot, based on observing 20 participants—with a range of prior experience using the assistant—as they solve diverse programming tasks across four languages. Our main finding is that interactions with programming assistants are *bimodal*: in *acceleration mode*, the programmer knows what to do next and uses Copilot to get there faster; in *exploration mode*, the programmer is unsure how to proceed and uses Copilot to explore their options. Based on our theory, we provide recommendations for improving the usability of future AI programming assistants.

CCS Concepts: • **Human-centered computing** → **HCI theory, concepts and models**; • **Software and its engineering** → *Software creation and management*.

Additional Key Words and Phrases: Program Synthesis, AI Assistants, Grounded Theory

ACM Reference Format:
Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 78 (April 2023), 27 pages. https://doi.org/10.1145/3586030

1 INTRODUCTION

The dream of an "AI assistant" working alongside the programmer has captured our imagination for several decades now, giving rise to a rich body of work from both the programming languages [Ferdowsifard et al. 2020; Miltner et al. 2019; Ni et al. 2021; Raychev et al. 2014] and the machine learning [Guo et al. 2021; Kalyan et al. 2018; Xu et al. 2020] communities. Thanks to recent breakthroughs in large language models (LLMs) [Li et al. 2022; Vaswani et al. 2017] this dream finally seems within reach. OpenAI's Codex model [Chen et al. 2021], which contains 12 billion model parameters and is trained on 54 million software repositories on GitHub, is able to correctly solve 30–70% of novel Python problems, while DeepMind's AlphaCode [Li et al. 2022] ranked in the top 54.3% among 5000 human programmers on the competitive programming platform Codeforces. With this impressive performance, large code-generating models are quickly escaping research labs to power industrial programming assistant tools, such as Github Copilot [Friedman 2021].

The growing adoption of these tools gives rise to questions about the nature of AI-assisted programming: *What kinds of tasks do programmers need assistance with? How do programmers prefer to communicate their intent to the tool? How do they validate the generated code to determine its*

*Equal contribution

Authors' addresses: Shraddha Barke, UC San Diego, USA, sbarke@ucsd.edu; Michael B. James, UC San Diego, USA, m3james@ucsd.edu; Nadia Polikarpova, UC San Diego, USA, npolikarpova@ucsd.edu.

2475-1421/2023/4-ART78
https://doi.org/10.1145/3586030

correctness and how do they cope with errors? It is clear that the design of programming assistants should be informed by the answers to these questions, yet research on these topics is currently scarce. Specifically, we are aware of only one usability study of Copilot, by Vaithilingam et al. [2022]; although their work makes several interesting observations about human behavior (which we discuss in more detail in Sec. 7), ultimately it has a narrow goal of measuring whether Copilot helps programmers in solving stand-alone Python programming tasks. To complement this study and to obtain more generalizable insights that can inform the design of future tools, our work sets out to explore how programmers interact with Copilot in a broader setting.

Our contribution: grounded theory of Copilot-assisted programming. We approach this goal using the toolbox of *grounded theory* (GT) [Glaser and Strauss 1967], a qualitative research technique that has a long history in social sciences, and has recently been adopted to study phenomena in software engineering [Stol et al. 2016] and programming languages [Lubin and Chasins 2021]. GT is designed to build an understanding of a phenomenon from the ground up in a data-driven way. To this end, researchers start from raw data (such as interview transcripts or videos capturing some behavior) and tag this data with categories, which classify and explain the data; in GT parlance, this tagging process is called *qualitative coding*. Coding and data collection must interleave: as the researcher gains a better understanding of the phenomenon, they might design further experiments to collect more data; and as more data is observed, the set of categories used for coding is refined.

In this paper, we present the first grounded theory of how users interact with an AI programming assistant—specifically Github Copilot. To build this theory, we observed 20 participants as they used Copilot to complete several programming tasks we designed. Some of the tasks required contributing to an existing codebase, which we believe more faithfully mimics a realistic software development setting; the tasks also spanned multiple programming languages—Python, Rust, Haskell, and Java—in order to avoid language bias. We then iterated between coding the participants' interactions with Copilot, consolidating our observations into a theory, and adjusting the programming tasks to answer specific questions that came up. The study method is described in detail in Sec. 3.

Summary of findings. The main thesis of our theory (Sec. 4) is that user interactions with Copilot can be classified into two modes—*acceleration* and *exploration*—akin to the two systems of thought in dual-process theories of cognition [Carlston 2013; Milli et al. 2021], popularized by Daniel Kahneman's "Thinking, Fast and Slow" [Kahneman 2011]. In acceleration mode, the programmer already knows what they want to do next, and Copilot helps them get there quicker; interactions in this mode are fast and do not break programmer's flow. In exploration mode, the programmer is not sure how to proceed and uses Copilot to explore their options or get a starting point for the solution; interactions in this mode are slow and deliberate, and include explicit prompting and more extensive validation.

Sec. 5 describes two kinds of further analysis of our theory. First, we performed a quantitative analysis of the data collected during the study, comparing prompting and validation behaviors across modes, and quantifying the factors that influence the relative prevalence of each mode. Second, to reinforce our findings, we gathered additional data from five livestream videos we found on YouTube and Twitch, and confirmed that the streamers' behavior was consistent with our theory.

Based on our theory, we provide design recommendations for future programming assistants (Sec. 6). For example, if the tool is aware that the programmer is currently in acceleration mode, it could avoid breaking their flow by sticking with only short and high-confidence code suggestions. On the other hand, to aid exploration, the IDE could provide better affordances to compare and contrast alternative code suggestions, or simplify validation of generated code via automated testing or live programming.

2 COPILOT-ASSISTED PROGRAMMING, BY EXAMPLE

Copilot is a programming assistant released by Github in June 2021 [Friedman 2021], and since integrated into several development environments, including Visual Studio Code, JetBrains and Neovim. Copilot is powered by the OpenAI Codex family of models [Chen et al. 2021], which are derived by fine-tuning GPT-3 [Brown et al. 2020] on publicly available Github repositories.

In the rest of this section, we present two concrete scenarios of users interacting with Copilot, which are inspired by real interactions we observed in our study. The purpose of these scenarios is twofold: first, to introduce Copilot's UI and capabilities, and second, to illustrate the two main interaction modes we discovered in the study.

2.1 Copilot as Intelligent Auto-Completion

Axel, a confident Python programmer, is solving an Advent of Code [Wastl 2021] task, which takes as input a set of rules of the form `AB => C`, and computes the result of applying these rules to a given input string. He begins by mentally breaking down the task into small, well-defined subtasks, the first of which is to parse the rules from the input file into a dictionary. To accomplish the first subtask, he starts writing a function `parse_input` (Fig. 1). Although Axel has a good idea of what the code of this function should look like, he thinks Copilot can help him finish it faster and save him some keystrokes and mental effort of recalling API function names. To provide some context for the tool, he adds a comment before the function definition, explaining the format of the rules.

As Axel starts writing the function body, any time he pauses for a second, Copilot's grayed-out *suggestion* appears at the cursor. Fig. 1 shows an example of an *end-of-line suggestion*, which only completes the current line of code. In this case, Copilot suggests the correct API function invocation to split the rule into its left- and right-hand sides. To come up with this suggestion, Copilot relies on the *context*, *i.e.* some amount of source file content preceding the cursor, which can include both code and natural language comments, as is the case in our example.

```
# rules are formatted like:
# AB => C
def parse_input(filename):
    with open(filename) as f:
        template, rules = f.read().split("\n\n")
        for rule in rules:
            rule_parts = rule.split(" => ")
```

Fig. 1. Copilot's end-of-line suggestion appears at the cursor without explicit invocation. The programmer can press <tab> to accept it.

Because the suggestion in Fig. 1 is short and closely matches his expectations, Axel only takes a fraction of a second to examine and accept it, without ever leaving his state of flow. Throughout the implementation of `parse_input`, Axel might see a dozen of suggestions, which he quickly accepts (by pressing <tab>) or rejects (by simply typing on). Some of them are larger, *multi-line suggestions*, but Axel still seems to be able to dispatch them quickly by looking for patterns, such as expected control flow and familiar function names. We liken this kind of interaction with Copilot to the fast *System 1* in dual-process theories of cognition [Carlston 2013], which is characterized by quick, automatic, and heuristic decisions.

2.2 Copilot as an Exploration Tool

Emily is new to data science, and wants to visualize a dataset as a histogram. While she is familiar with Python, she is not familiar with the plotting library `matplotlib`. As a result, she does not know how to approach this task: not only which API functions to call, but also how to decompose the problem and the right set of abstractions to use. Emily decides to use Copilot to explore solutions.

Emily explicitly *prompts* Copilot with a natural-language comment, as shown in lines 12–13 of Fig. 2. Moreover, since she wants to explore multiple options, she presses <ctrl> + <enter> to bring up the *multi-suggestion pane*, which displays up to 10 unique suggestions in a separate pane

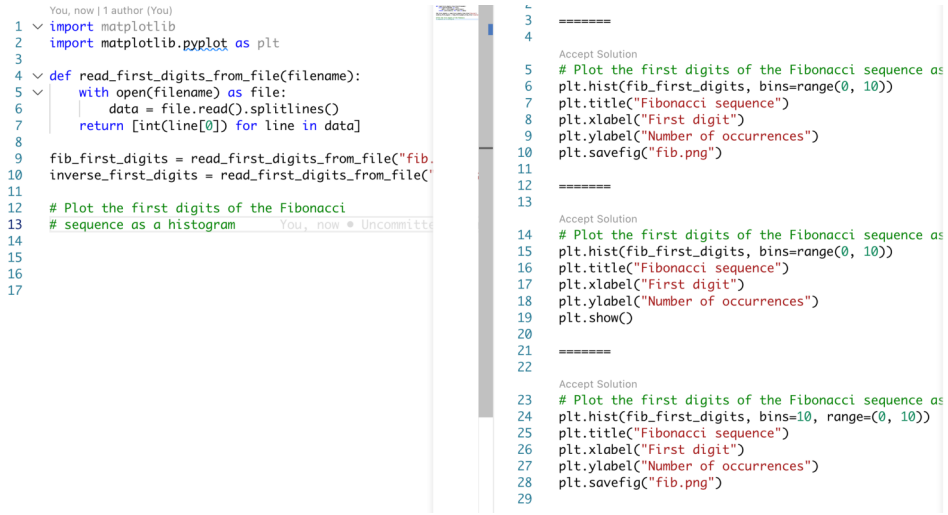


Fig. 2. The user writes an explicit comment prompt (lines 12–13 on the left) and invokes Copilot's multi-suggestion pane by pressing <ctrl> + <enter>. The pane, shown on the right, displays up to 10 unique suggestions, which reflect slightly different ways to make a histogram with `matplotlib`.

(shown on the right of Fig. 2). Emily carefully inspects the first three suggestions; since all of them have similar structure and use common API calls, such as `plt.hist`, she feels confident that Copilot understands her task well, and hence the suggestions can be trusted. She copy-pastes the part of the first suggestion she likes best into her code; as a side-effect, she gains some understanding of this part of the `matplotlib` API, including alternative ways to call `plt.hist`. To double-check that the code does what she expects, Emily runs it and inspects the generated histogram. This is an example of *validation*, a term we use broadly, to encompass any behavior meant to increase user's confidence that the generated code matches their intent.

When faced with an unfamiliar task, Emily was prepared to put deliberate effort into writing the prompt, invoking the multi-suggestion pane, exploring multiple suggestions to select a suitable snippet, and finally validating the generated code by running it. We liken this, second kind of interaction with Copilot to the slow *System 2*, which is responsible for conscious thought and careful, deliberate decision-making.

3 METHOD

Participants. We developed our theory through a user study with 20 participants (15 from academia and 5 from industry). We recruited these participants through personal contacts, Twitter, and Reddit. Nine of the participants had used Copilot to varying degrees prior to the study. Participants were not paid, but those without access to Copilot were provided access to the technical preview for continued use after the study concluded. Tab. 1 lists relevant information about each participant. We asked each participant to select a statement best describing their level of experience with possible target languages, with options ranging from "I have never used Python", to "I use Python professionally" (from least-to-most, used in Tab. 1: Never, Occasional, Regular, and Professional).

[1]Participant did not have time to attempt Python section

Table 1. Participants overview. PCU: Prior Copilot Usage. We show the language(s) used on their task, their usage experience with their task language (Never, Occasional, Regular, Professional), whether they had used Copilot prior to the study, their occupation, and what task they worked on.

ID	Language(s)	Language Experience	PCU	Occupation	Task
P1	Python	Professional	Yes	Professor	Chat Server
P2	Rust	Professional	No	PhD Student	Chat Client
P3	Rust	Occasional	No	Professor	Chat Client
P4	Python	Occasional	Yes	Postdoc	Chat Server
P5	Python	Regular	No	Software Engineer	Chat Client
P6	Rust	Professional	Yes	PhD Student	Chat Server
P7	Rust	Professional	No	Software Engineer	Chat Server
P8	Rust	Professional	No	PhD Student	Chat Server
P9	Rust[1]	Occasional	No	Undergraduate Student	Benford's law
P10	Python	Occasional	No	Undergraduate Student	Chat Client
P11	Rust+Python	Professional + Professional	Yes	Cybersecurity Developer	Benford's law
P12	Rust+Python	Professional + Occasional	Yes	Software Engineer	Benford's law
P13	Rust+Python	Regular + Occasional	Yes	PhD Student	Benford's law
P14	Python	Professional	No	PhD Student	Advent of Code
P15	Python	Professional	Yes	PhD Student	Advent of Code
P16	Haskell	Professional	No	PhD Student	Advent of Code
P17	Rust	Professional	Yes	Founder	Advent of Code
P18	Java	Occasional	No	PhD Student	Advent of Code
P19	Python	Occasional	No	PhD Student	Advent of Code
P20	Haskell	Occasional	Yes	PhD Student	Advent of Code

We screened out participants who had never used the target language. We choose a qualitative self-assignment of experience as other common metrics, such as years-of-experience, can be misleading. For example, a professor having used Rust occasionally over eight years is arguably less experienced than as a software engineer using Rust all day for a year.

User protocol. To study participants using Copilot, we gave them a programming task to attempt with Copilot's help. Over the course of an hour a participant was given a small training task to familiarize them with Copilot's various usage models (*i.e.* code completion, natural language prompt, and the multi-suggestion pane). During the core task—about 20-40 minutes—a participant was asked to talk through their interactions with Copilot. They were encouraged to work Copilot into their usual workflow, but they were not required to use Copilot. After the task, the interviewer asked them questions through a semi-structured interview; these questions as well as the tasks are available in our supplementary package. The entire session was recorded and transcribed to use as data in our grounded theory.

Grounded Theory Process. Grounded Theory (GT) takes qualitative data and produces a theory in an iterative process, first pioneered by [Glaser and Strauss 1967]. As opposed to evaluating fixed, a priori hypotheses, a study using the GT methodology seeks to generate new hypotheses in an overarching theory developed without prior theoretical knowledge on the topic. A researcher produces this theory by constantly interleaving data collection and data analysis. GT has diversified into three primary styles over the past half-century. We follow the framework laid out by Strauss and Corbin [Strauss and Corbin 1990], commonly called *Straussian Grounded Theory* [Stol et al. 2016]. We describe our process below.

We began our study with the blank slate question: "How do programmers interact with Copilot?" Our bimodal theory of acceleration and exploration was not yet formed. During each session, we took notes to guide our semi-structured interview. After each session, we tagged any portion of

the recording relevant to Copilot with a note. We took into account what the participant said, what they did, and their body language. For example, we initially tagged an instance where P2 was carefully examining and highlighting part of a large Copilot suggestion as "validating sub-expression". Tagging the data in this way is called *(qualitative) coding*; and doing so without a set of predefined codes is called *open coding* in Straussian GT. The first two authors coded the first two videos together, to agree on a coding style, but later data were coded by one and discussed by both.

By the end of the eighth session, we began to see patterns emerging in our data. We noticed two distinct patterns in our codes which eventually crystallized into our acceleration and exploration modes. During this phase of analysis, we aggregated our codes to understand the *conditions* when a participant would enter acceleration or exploration, and the *strategies* a participant deployed in that mode. For example, we realized that if a programer can decompose a problem, then they often ended up in acceleration (details in Sec. 4.1.1). Once in this acceleration mode, programmers would validate a suggestion by a kind of visual "pattern matching" (details in Sec. 4.1.4). This process of aggregating and analyzing our codes form the *axial coding* phase of GT.

After the eighth session, we created a new task to specifically test our emerging theory. This process of testing aspects of a theory-in-progress is known in GT as *theoretical sampling*. After gathering sufficient data on that third task, we created a fourth task to investigate one final aspect of our theory (validation of Copilot's suggestions). In the second half of the study, we linked together our codes and notes into the final bimodal theory we present, in what Straussian GT calls *selective coding*. At the 20th participant, we could fit all existing data into our theory and no new data surprised us. Having reached this point of *theoretical saturation*, we concluded our GT study.

Tasks. The list of all four tasks and their descriptions can be found in Tab. 2. The full task templates we provided to participants are available in our replication package [Barke et al. 2023].

Our tasks evolved over the course of the study. We started with the "Chat Server" and "Chat Client" pair of tasks, meant to emulate working on a complex project, with a shared library and specialized APIs. These two initial tasks required contributing to an existing codebase we created, which implements a secure chat application. The first task, Chat Server, asked participants to implement the server backend, focusing on its "business logic". We provided most of the networking code, and the participant's task was to implement the log-in, chat, and chat-command functionality (*e.g.* `/quit` to quit). The complementary task Chat Client focused on the client side of the chat application. Here, we provided no networking code so the participant had to figure out how to use the often unfamiliar socket API. We also required using a custom cryptographic API we implemented, in order to ensure that some part of the API was unfamiliar both to the participant and to Copilot.

To investigate the acceleration and exploration modes further, we created the "Benford's Law"[2] task. This task had two parts, to separately investigate the acceleration and exploration modes we found. In the first half, the participant implements an efficient Fibonacci sequence generator. We believed that all participants would be familiar with the algorithm, and hence would accelerate through this half of the task, allowing us to more deeply characterize the acceleration mode. In the second half, they plotted this sequence and another ($\frac{1}{2}$, $\frac{1}{3}$, ..., $\frac{1}{180}$ as `floats`) using `matplotlib`; this sub-task is used as the example in Sec. 2.2. Our participants were not confident users of the plotting library's API, so they needed to turn to some external resource to complete the task. This half stressed the code exploration part of our theory. In addition, our Benford's Law task asked participants to complete the first half in Rust and the second half in Python. This division gave us within-participant information on how different languages impact Copilot usage.

[2]Benford's Law says that in natural-looking datasets, the leading digit of any datum is likely to be small. It is useful as a signal for finding fraudulent data.

Table 2. The four programming tasks used in our study and their descriptions. Task LOC is the lines of code in the provided code and Solution LOC are the number of lines in our canonical solutions.

Task	Language(s)	Description	Task LOC	Solution LOC
Chat Server	Python/Rust	Implement core "business logic" of a chat application, involving a small state machine.	253/369	61/83
Chat Client	Python/Rust	Implement networking code for a chat application, using a custom cryptographic API and standard but often unfamiliar socket API.	262/368	52/84
Benford's Law	Rust & Python	Use Rust to generate two sequences—the Fibonacci sequence and reciprocals of sequential natural numbers; then plot these sequences using Python's `matplotlib`.	9	35
Advent of Code	Python/Rust/ Haskell/Java	Implement a string manipulation task from a programming competition.	2-18	29-41

Our fourth task was a string manipulation problem inspired by the 2021 edition of Advent of Code (this task is used as the example in Sec. 2.1). We wanted to collect more data about how programmers validate suggestions from Copilot, and this task was a good fit because it comes with a test case and a very precise description, and also has two independent sub-tasks, so it provided several options for checking solutions at different levels of granularity. The data we collected rounded out our hypotheses about validation (Sec. 4.1.4, Sec. 4.2.5).

4 THEORY

Through our grounded theory analysis, we identified two main modes of developer interactions with Copilot: *acceleration* and *exploration*. In acceleration mode, a programmer uses Copilot to *execute* their planned code actions, by completing a logical unit of code or a comment. Acceleration works within user's sense of flow. For example, recall how in Sec. 2.1 Axel accepted Copilot's suggestion of `rule.split(" => ")`, knowing it was what he wanted to type anyways. This is a characteristic example of acceleration, where Copilot was helping him program faster.

In exploration mode, a programmer relies on Copilot to help them *plan* their code actions. A programmer may use Copilot to assist with unfamiliar syntax, to look up the appropriate API, or to discover the right algorithm. In Sec. 2.2, when Emily was searching for the right set of `matplotlib` calls, she was considering alternatives, gaining confidence in the API, and simply trying to learn how to finish her task. All of these intentions are part of the exploration mode when using Copilot. We found that programmers alternate between these two modes as they complete their task, fluidly switching from one mode to the other.

In this section, we systematize our observation of each mode: *acceleration* (Sec. 4.1) and *exploration* (Sec. 4.2). For each mode, we start with identifying the *conditions* that lead the participant to end up in that mode, and then proceed to describe common *strategies* (*i.e.* behavioral patterns) we observed in that mode. Each numbered subsection (*e.g.* Sec. 4.1.3) is a hypothesis deriving from our top-level bimodal theory. Each named paragraph heading is an aspect of that hypothesis.

4.1 Acceleration

Acceleration is characterized by the programmer being "in the zone" and actively "driving" the development, while occasionally relying on Copilot to complete their thought process. A programmer will often accept a Copilot suggestion without much comment and keep on going without losing focus. In this interaction mode, programmers tend to think of Copilot as an intelligent autocomplete that just needs to complete their line of thought. This idea was well put by P13 who said:

> "I think of Copilot as an intelligent autocomplete... I already have the line of code in mind and I just want to see if it can do it, type it out faster than I can."

P15 added to this, calling Copilot "more or less an advanced autocomplete".

4.1.1 ***Programmers Use Acceleration after Decomposing the Task.*** We found that the main causal condition for a participant to end up in acceleration mode is being able to decompose the programming task into *microtasks*. We define a microtask to be a participant-defined task with a well-understood and well-defined job. For example, when P16 was working on the Advent of Code task, they created two separate microtasks to parse the input and to compute the output. Because they understood these microtasks well, they wrote a type signature and used descriptive names for each of them; as a result, Copilot was adept at completing these microtasks for them. Another illustrative example is our Benford's Law task, which was explicitly designed to have a familiar and an unfamiliar subtask. In the first subtask, participants were asked to implement a fast Fibonacci function. All four participants were familiar with the Fibonacci sequence and knew how to make it efficient. As a result, all of them were able to use Copilot to accelerate through this familiar microtask. P14 explicitly noted:

> "I think Copilot would be more helpful in cases where there are a lot of tedious subtasks which requires less of thinking and more of just coding."

We observed that language expertise or familiarity with Copilot seem to play less of a role in determining whether a participant would engage in acceleration, compared to their understanding of the *algorithm* for solving the task. For example, P4 was not very comfortable with Python, but they knew what needed to be done in their task algorithmically, and so were able to break it down into microtasks, leading to acceleration. That said, we do observe that language experts and prior Copilot users spend a larger proportion of their total interaction time in acceleration mode; we present quantitative data supporting this observation in Sec. 5.

4.1.2 ***Programmers Focus on Small Logical Units.*** Participants who interacted with Copilot in acceleration mode would frequently accept end-of-line suggestions. These were often function calls or argument completions. For example, when P1 wanted to send a message to a client connection object in the Chat Client task, they typed `client_conn.se` and immediately accepted Copilot's suggestion `client_conn.send_message()`. This behavior was seen across all the four tasks when participants were in acceleration mode. For a microtask of parsing file input, P15 wanted to spilt the data based on spaces so they typed `data = x.` to which Copilot correctly suggested `data = x.split("") for x in data`. Participants would happily accept these end-of-line completions with reactions like "Yes that's what I wanted!" and "Thank you Copilot!"

When a programmer is focused on a logical unit of code, they want suggestions *only for that unit*. When they are writing a print statement, they prefer to get a suggestion to the end of the statement. When writing a snippet to message all connected clients, they might instead prefer an entire `for` loop, *but not more*. For example, at one point P8 was focused on a single call to the `startswith` function, but Copilot suggested a large piece of code; P8 reacted with "that's way more than what I needed!" and went on to delete everything but the first line `if msg.startswith('/')`.

The size of a logical unit differs based on the language and context. In an imperative language, this is most often a line of code. However, in a functional language like Haskell, logical units appear to be smaller. P16 said that "in Haskell it just needs to suggest less. [It should] give me the next function I'm going to compose and not the whole composition chain."

4.1.3 ***Long Suggestions Break Flow.*** In acceleration mode, long, multi-line suggestions are at best dismissed out of hand and at worst distract the programmer away from their flow.

Upon getting a 16-line suggestion and after just four seconds of review P6 uttered: "Oh God, no. Absolutely not". When P6 got other large suggestions, they would exclaim, "Stop it!", and continue to program as before. This participant also made use of the `<esc>` key binding to actively dismiss a suggestion they did not care for.

On the other hand, many programmers felt "compelled to read the [suggested] code" (P16) and noted that reading long suggestions would often break their flow. As P1 puts it:

> "When I'm writing, I already have in mind the full line and it's just a matter of transmitting to my fingertips, and to the keyboard. But when I have those mid-line suggestions and those suggestions are not just until the end of line, but actually a few more lines, that breaks my flow of typing. So instead of writing the full line, I have to stop, look at the code, think whether I want this or not."

This sentiment was echoed by multiple participants: P11 was "distracted by everything Copilot was throwing at [them]"; P7 was "lost in the sauce" after analyzing a long suggestion; P17 felt "discombobulated", and others (P8, P11) made similar comments. P16 put it eloquently:

> "I was about to write the code and I knew what I wanted to write. But now I'm sitting here, seeing if somehow Copilot came up with something better than the person who's been writing Haskell for five years, I don't know why am I giving it the time of day."

Such distractions cause some programmers to give up on the tool entirely: P1, P6, and P15 all had Copilot disabled prior to the study—having had access for several months—and they all cited distractions from the always-on suggestions as a factor.

4.1.4 ***Programmers Validate Suggestions by "Pattern Matching".*** In order to quickly recognize whether a suggestion is worthwhile, participants looked for the presence of certain keywords or control structures. The keywords included function calls or variable names that they expected should be part of the solution. P1 explicitly stated that the presence or absence of certain keywords would determine whether the suggestion was worth considering.

Most other programmers who commented on how they validated suggestions in acceleration mode mentioned control structures (P4, P17, P19). P4, for instance, immediately rejected an iterative suggestion because they strongly preferred a recursive implementation. On one occasion, Copilot suggested code to P6 when they already had an idea of what shape that code should take; they described their validation process in this instance as follows:

> "I have a picture in mind and that picture ranges from syntactic or textual features—like a literal shape in words—to semantic about the kind of methods that are being invoked, the order in which they should be invoked, and so on. When I see a suggestion, the closer that suggestion is to the mental image I hold in my head, the more likely I am to trust it."

These participants appear to first see and understand control-flow features before understanding data or logic flow features. This is consistent with previous findings dating back to FORTRAN and COBOL programming [Pennington 1987], where programmers briefly shown small code snippets could best answer questions about control flow compared to data- or logic-flow.

4.1.5 ***Programmers Are Reluctant to Accept or Repair Suggestions.*** Participants in acceleration mode end up quickly rejecting suggestions that don't have the right patterns. Suggestions that are almost-correct were accepted if a small repair was obvious to the participant. P1 accepted

a small inline suggestion which had a call to `handshake()` function, checked if it existed, and since it did not, they made a minor modification, changing the function name to `do_dh_handshake()`. The entire accept-validate-repair sequence seemed to occur without interrupting their state of flow. P1, P4 would often accept similar-looking function names but double check if they actually existed:

> "Each time it uses something else from the context, I usually double check, like in this case it was very similar so I could have been fooled, and each time this happens it reinforces the need to check everything just to see if it has the proper names."

Although programmers tend to dismiss code that does not match their expectations, sometimes Copilot's suggestion makes them aware of a corner case they have not yet considered. P4 saw Copilot write an inequality check while working on the Chat Server task, and they said that they "probably wouldn't have remembered on their first run through to check that [clients] are distinct". Both P6 and P8, working in Rust on the Chat Server, noticed that Copilot used a partial function `.unwrap()`. When asked about this, P8 said:

> "Copilot suggested code to handle it in one case and now I'm going to change it around to handle the other case as well."

4.2 Exploration

In the previous section we focused on the use of Copilot when the programmer has a good idea for how to approach the task. But what if they do not? In that case they might use Copilot to help them get started, suggest potentially useful structure and API calls, or explore alternative solutions. All of these behaviors fit under what we call exploration mode. Exploration is characterized by the programmer letting Copilot "drive", as opposed to acceleration, where the programmer is the driver. In the rest of this section, we first describe the conditions that lead programmers to enter exploration mode, and then we characterize the common behaviors in that mode.

4.2.1 ***Programmers Explore when Faced with Novel Tasks or Unexpected Behavior.*** Recall that most often the programmer ended up in acceleration mode once they had successfully decomposed the programming task into a sequence of steps (Sec. 4.1.1); dually, when the programmer was uncertain how to break down the task, they would often use Copilot for code exploration. P4 said:

> "Copilot feels useful for doing novel tasks that I don't necessarily know how to do. It is easier to jump in and get started with the task".

Not knowing where to start was one of two primary ways we observed participants begin an exploration phase of their study. The other way participants (P11, P13, P14) began exploration was when they hit some code that does not work as expected, regardless of the code's provenance. They would try a variety of prompting and validation strategies to attempt to fix their bug.

4.2.2 ***Programmers Explore when They Trust the Model.*** A participant's level of confidence and excitement about code-generating models was highly correlated with whether and to which extent they would engage in exploration. During the training task, Copilot produced a large, correct suggestion for P18; they exclaimed, "I'm not gonna be a developer, I'm gonna be a guy who comments!" This level of excitement was shared among many of our participants early in the task, like P7 saying, "it's so exciting to see it write [code] for you!". Those participants who were excited about Copilot would often let the tool drive before even attempting to solve the task themselves.

Sometimes, such excessive enthusiasm would get in the way of actually completing a task. For example, P10 made the least progress compared to others on the same task; in our post-study interview, they admitted that they were, "a little too reliant on Copilot":

> "I was trying to get Copilot to do it for me, maybe I should have given smaller tasks to Copilot and done the rest myself instead of depending entirely on Copilot."

This overoptimism is characteristic of the misunderstanding users often have with program synthesizers. P9 and P10 were both hitting the *user-synthesizer gap*, which separates what the user expects a program synthesizer to be capable of, and what the synthesizer can actually do [Ferdowsifard et al. 2020].

4.2.3 ***Programmers Explicitly Prompt Copilot with Comments.*** Nearly every participant (P2, P3, P4, P5, P7, P8, P10, P11, P12, P13, P14, P17, P18, P19, P20) wrote at least one natural language comment as a prompt to Copilot, specifically for an exploratory task.

Programmers prefer comment prompts in exploration. Programmers felt that natural language prompts in the form of comments offered a greater level of control than code prompts (P17). P2 told us that, "writing a couple of lines [of comments] is a lot easier than writing code." This feeling of being more in control was echoed by P5 who said:

> "I think that the natural language prompt is more cohesive because it's interruptive to be typing out something and then for Copilot to guess what you're thinking with that small pseudocode. It's nice to have a comment that you've written about your mental model and then going to the next line and seeing what Copilot thinks of that."

Programmers write more and different comments when using Copilot. Participants seem to distinguish between comments made for themselves and Copilot. In the words of P6, "The kind of comments I would write to Copilot are not the kind of comments I would use to document my code." P2, P3, P5, P12, and P19 all told us that the majority of their comments were explicitly meant for Copilot. P7 was the sole exception: they wrote comments to jot down their design ideas saying, "I'm writing this not so much to inform Copilot but just to organize my own thoughts"; they added that being able to prompt Copilot using those comments was a nice side effect.

Participants were willing to invest more time interacting with Copilot via comment prompts in exploration mode. They would add detailed information in the comments in the hope that Copilot would have enough context to generate good suggestions (P2, P3). They would rewrite comments with more relevant information if the suggestions did not match their expectations, engaging in a conversation with Copilot. P2 and P6 wished they had a "community guide" (P2) on how to write comments so that Copilot could better understand their intent.

Further, in our interviews, multiple people described their usual commenting workflow as post-hoc: they add comments after completing code. Hence, the participants were willing to change their commenting workflow to get the benefits of Copilot.

Programmers frequently remove comments after completing an interaction with Copilot. Many participants (P3, P4, P7, and P8) would repeatedly delete comments that were meant for Copilot. P19 said that cleaning up comments written for Copilot is essential:

> "I wrote this comment to convert String to array just for Copilot, I would never leave this here because it's just obvious what it's doing. [...] These comments aren't adding value to the code. I think you also have to do like a comment cleanup after using Copilot."

4.2.4 ***Programmers are Willing to Explore Multiple Suggestions.*** In exploration mode, we often saw participants spend significant time foraging through Copilot's suggestions in a way largely unseen during acceleration. This included using the *multi-suggestion pane*, both for its primary intended purpose—selecting a single suggestion out of many—and for more creative purposes, such as cherry-picking snippets from multiple suggestions, API search, and gauging Copilot's confidence in a code pattern.

Participants tend to use the multi-suggestion pane when faced with an exploratory task (P2, P4, P5, P7, P10, P12–20). They would either write a comment prompt or a code prompt before invoking the multi-suggestion pane. This enabled participants to explore alternate ways to complete their task while also providing an explicit way to invoke Copilot. P10, P15, P19 preferred the multi-suggestion pane over getting suggestions inline in all cases. P15 said:

> "I prefer multiple suggestions over inline because sometimes the first solution is not what I want so if I have something to choose from, it makes my life easier."

Some only occasionally got value from the multi-suggestion pane. P6 said that:

> "If I think there's a range of possible ways to do a task and I want Copilot to show me a bunch of them I can see how this could be useful."

Similar to P6, P14 and P17 preferred the multi-suggestion pane only while exploring code as it showed them more options. Yet others turned to the multi-suggestion pane when Copilot's always-on suggestions failed to meet their needs.

Programmers cherry-pick code from multiple suggestions. Participants took part of a solution from the multi-suggestion pane or combined code from different solutions in the pane. P2, P3, P4, P5, P18 often accepted only interesting sub-snippets from the multi-suggestion pane. For example, P18 forgot the syntax for declaring a new `Hashmap` in Java, and while Copilot suggested a bunch of formatting code around the suggestion, P18 only copied the line that performed the declaration. P2 went ahead to combine interesting parts from more than one suggestion stating:

> "I mostly just do a deep dive on the first one it shows me, and if that differs from my expectation, for example when it wasn't directly invoking the handshake function, I specifically look for other suggestions that are like the first one but do that other thing correctly."

Programmers use the multi-suggestion pane in lieu of StackOverflow. When programmers do not know the immediate next steps in their workflow, they often write a comment to Copilot and invoke the multi-suggestion pane. This workflow is similar to how programmers already use online forums like StackOverflow: they are unsure about the implementation details but they can describe their goal. In fact, P12 mentioned that they were mostly using the multi-suggestion pane as a search engine during exploration. P4 often used Copilot for purely syntactic searches, for example, to find the `x in xs` syntax in Python. P15 cemented this further:

> "what would have been a StackOverflow search, Copilot pretty much gave that to me."

Participants emphasized that the multi-suggestion pane helped them use unfamiliar APIs, even if they did not gain a deep understanding of these APIs. P5 explains:

> "It definitely helped me understand how best to use the API. I feel like my actual understanding of [the socket or crypto library] is not better but I was able to use them effectively."

Programmers use the multi-suggestion pane to gauge Copilot's confidence. Participants assigned a higher confidence to Copilot's suggestions if a particular pattern or API call appeared repeatedly in the multi-suggestion pane. Participants seemed to think that repetition implied Copilot was more confident about the suggestion. For example, P5 consulted Copilot's multi-suggestion pane when they were trying to use the unfamiliar socket library in Python. After looking through several suggestions, and seeing that they all called the same method, they accepted the first inline suggestion. When asked how confident they felt about it, P5 said:

> "I'm pretty confident. I haven't used this socket library, but it seems Copilot has seen this pattern enough that, this is what I want."

P4 had a similar experience but with Python syntax: they checked the multi-suggestion pane to reach a sense of consensus with Copilot on how to use the `del` keyword in Python.

Programmers suffer from cognitive overload due to multi-suggestion pane. P1, P4, P6, P7 and P13 did not like the multi-suggestion pane popping up in a separate window stating that it added to their cognitive load. P4 said that they would prefer a modeless (inline) interaction, and P6 stated:

> "Seeing the code in context of where it's going to be was way more valuable than seeing it in a separate pane where I have to draw all these additional connections."

P13 spent a considerable amount of time skimming and trying to differentiate the code suggestions in the multi-suggestion pane, prompting them to make the following feature request:

> "It might be nice if it could highlight what it's doing or which parts are different, just something that gives me clues as to why I should pick one over the other."

Programmers suffer from an anchoring bias when looking through multiple suggestions. The anchoring bias influences behavior based on the first piece of information received. We observed participants believe that suggestions were ranked and that the top suggestion *must* be closest to their intent (P18). This was also evident through P2's behavior who would inspect the first suggestion more deeply then skim through the rest.

4.2.5 ***Programmers Validate Suggestions Explicitly.*** Programmers would validate Copilot's suggestions more carefully in exploration mode as compared to acceleration. Their validation strategies included code *examination*, code *execution* (or testing), relying on IDE-integrated *static analysis* (*e.g.* a type checker), and looking up *documentation*. We look at these techniques in detail.

Examination. Unlike acceleration mode, where participants quickly triage code suggestions by "pattern matching", exploration mode is characterized by carefully examining the details of Copilot-generated code. For example, P19 said that they would "always check [the code] line by line", and P5 mentioned that their role seemed to have shifted from being a programmer to being a code reviewer: "It's nice to have code to review instead of write". Participants found it important to cross-check Copilot's suggestions just as they would do for code from an external resource. When asked how much they trusted Copilot's suggestions, P14 said:

> "I consider it as a result I would obtain from a web search. It's not official documentation, it's something that needs my examination...if it works it works"

Execution. Code execution was common—occurring in every task by at least one participant—although not as common as examination. In case of the server and client task, participants P3 and P7 would frequently run their code by connecting the client to server and checking if it has the

expected behavior. For the Benford's law task, P11 wrote test cases in Rust using `assert_eq` to check whether the Fibonacci function suggested by Copilot was correct. All participants in the Advent of Code task ran their code to check whether they parsed the input file correctly.

In addition to executing the entire program, some participants used a Read-Eval-Print-Loop (REPL) as a scratchpad to validate code suggestions (P14, P16, P19). P16 used the Haskell REPL throughout the study to validate the results of subtasks. Copilot suggested an `adjacents` function that takes a string and pairs adjacent characters together. P16 validated the correctness of this function by running it on toy input `adjacents "helloworld"`.

Static analysis. In typed languages like Rust, the type checker or another static analyzer frequently replaced validation by execution. For example, P17 did not run their code even once for the Advent of Code task, despite the task being designed to encourage testing. They reasoned that the `rust-analyzer`[3] tool—which compiles and reports type errors in the background—took away the need to explicitly compile and execute the code.

> "In Rust I don't [explicitly] compile often just because I feel like the there's a lot of the type system and being able to reason about state better because mutability is demarcated a lot. But if this were in Python, I would be checking a lot by running in a REPL."

P7 thought it was "cool how you can see the suggestion and then rely on the type checker to find the problems." In general, most participants using statically typed languages relied on IDE support to help them validate code. P6, P8 and P17 relied on Rust analyzer and P6 had this to say:

> "I rely on the Rust compiler to check that I'm not doing anything incorrect. The nice part about being a statically typed language is you can catch all that at compile time so I just rely on Rust analyzer to do most of the heavy lifting for me."

Documentation. Lastly, consulting documentation was another common strategy to explicitly validate code from Copilot. As an example, P11 was trying to plot a histogram in `matplotlib`, but was unsure of the correct arguments for the `plt.hist` function. They accepted a couple of Copilot's suggestions but explicitly went to validate the suggested arguments by reading the documentation within the IDE. Participant P17, who never executed their Rust code, would instead hover over the variables and function names to access API documentation within the IDE. Participants that did not have documentation built into their IDE would turn to a web resource. For example, P14 accepted Copilot's suggestion for parsing file input in the Advent of Code task, and then validated the functionality of `splitlines` by crosschecking with the official Python documentation. P11 also used Google for crosschecking whether the Fibonacci sequence suggested by Copilot was accurate.

4.2.6 ***Programmers Are Willing to Accept and Edit.*** Unlike acceleration mode, where participants were quick to dismiss a suggestion that didn't match their expectations, during exploration, they seemed to prefer deleting or editing code rather than writing code from scratch. When a participant saw a segment of code that they felt they were likely to need in the future, they would hang on to it (P2, P3, P4, P6, P8). P2 was exploring code for one stage of writing a chat server when they saw code needed for a later stage and said: "I'm keeping [that] for later". During their exploration, they accepted a 40 line block of code to add:

> "Eh, I'm just going to accept this. It's close enough to what I want that I can modify it."

[3]https://github.com/rust-lang/rust-analyzer

P3 said: "I wanna see what [Copilot] gives me, then I'll edit them away". Some participants were able to complete most of their task by accepting a large block of code and then slowly breaking it down. P7 accepted a large block of code early on and iteratively repaired it into the code they needed. P5 had a similar experience and said, "It's nice to have code to review instead of write".

Commonly, participants sought a suggestion from Copilot only to keep the control structure. As a representative example, P8 was writing a message-handling function in Rust, when Copilot produced a 15-line suggestion, containing a `match` statement and the logic of its branches. After examination, P8 accepted the suggestion but quickly deleted the content of the branches, retaining only the structure of the `match`. We saw this many times with P1, P2, P11, P17, P18 as well. P17 said:

> "If I'm in a mode where I want to rip apart a solution and use it as a template then I can look at the multi-suggestion pane and select whichever suits my needs."

Copilot-generated code is harder to debug. On the flip side, participants found it more difficult to spot an error in code generated by Copilot. For example, P13 had to rely on Copilot to interface with `matplotlib`; when they noticed undesired behavior in that code, they said:

> "I don't see the error immediately and unfortunately because this is generated, I don't understand it as well as I feel like I would've if I had written it. I find reading code that I didn't write to be a lot more difficult than reading code that I did write, so if there's any chance that Copilot is going to get it wrong, I'd rather just get it wrong myself because at least that way I understand what's going on much better."

We observed a similar effect with P9, who could not complete their task due to subtly incorrect code suggested by Copilot. Copilot's suggestion opened a file in read-only mode, causing the program to fail when attempting to write. P9 was not able to understand and localize the error, instead spending a long time trying to add more code to perform an unrelated file flush operation.

5 ADDITIONAL ANALYSIS

In this section, we first provide quantitative evidence to support the findings from our grounded theory analysis. We then present the results of a qualitative analysis on five livestream videos to provide additional evidence that further supports our theory.

5.1 Quantitative Analysis

At the end of our grounded theory analysis, we closed our codebook and re-coded all videos with a fixed set of codes that emerged to be most noteworthy. Fig. 3 represents this codeline of the different activities we observed in each of the two interaction modes. The activities include prompting strategies, validation strategies, and the outcomes of Copilot's suggestions *i.e.* whether the participant accepts, rejects, or edits the suggestion. We then performed a quantitative analysis on this codeline to investigate the following questions:

(Q1) What factors influence the time spent in each of the two interaction modes?
(Q2) What are the prompting strategies used to invoke Copilot in the two interaction modes?
(Q3) How do the validation strategies differ across the two interaction modes and by task?

5.1.1 ***Time Spent in Interaction Modes.*** The total amount of study time spent by all participants interacting with Copilot in exploration mode (248.6 minutes) is more than twice that in acceleration mode (104.7 minutes). This is not surprising, since exploration is the "slow *System 2*" mode, where each interaction takes longer. At the same time, the ratio of time spent in the two modes is not

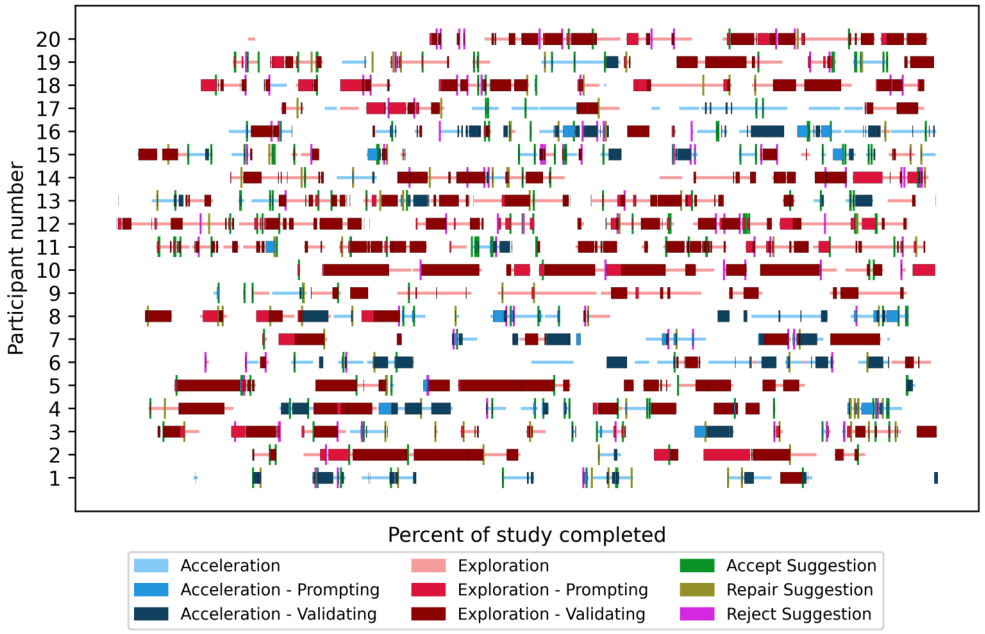
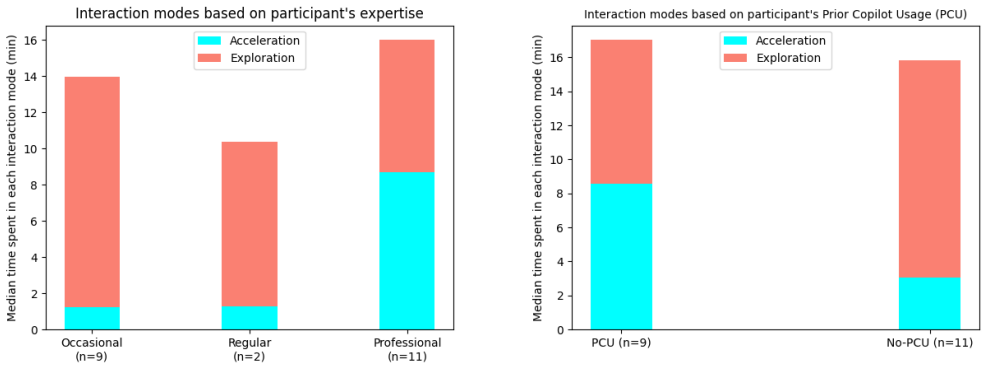


Fig. 3. Timeline of observed activities in each interaction mode for the 20 study participants. The qualitative codes include different prompting strategies, validation strategies and outcomes of Copilot's suggestions (accept, reject or repair)



(a) Grouped by language expertise.

(b) Grouped by prior Copilot usage.

Fig. 4. Median time spent in acceleration vs exploration mode for different participant groups.

constant across participants. Below, we investigate which factors influence this ratio, including language expertise, prior Copilot usage, the nature of the task, and the programming language.

Language expertise. Fig. 4a shows the median time spent in two modes split by the participant's language expertise. We can clearly see that professional participants with the most language

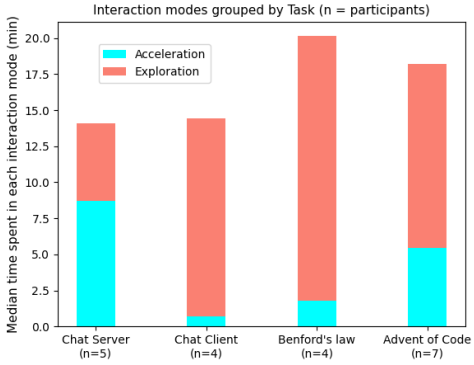


Fig. 5. Median time spent in acceleration vs exploration mode, grouped by task.

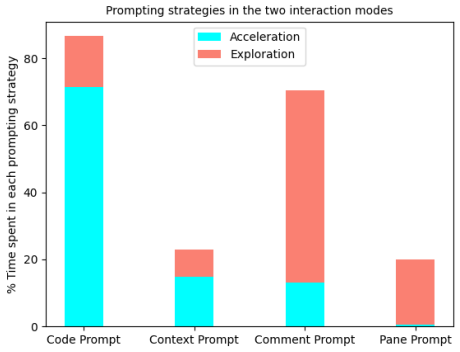


Fig. 6. Prevalence of prompting strategy as percentage of total prompting time.

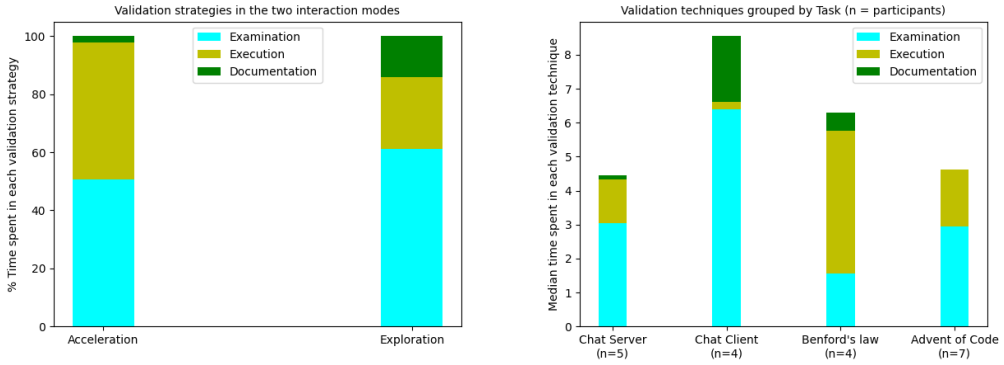
expertise spend more time accelerating than the other two groups. This is not surprising, since they are more likely to already know how to solve the task in the given language.

Prior Copilot usage. We can see in Fig. 4b that the total interaction time is roughly the same for participants with and without prior Copilot usage. Given roughly the same overall time, prior users spend less time exploring (and more time accelerating) than novice users. We attribute this difference to the effect we observed in Sec. 4.2.2, where novice users have higher expectations of Copilot's ability to solve high-level exploratory tasks.

Nature of Task. Fig. 5 shows the median time spent in each mode grouped by task. Both Chat Client and Benford's Law prominently feature interaction with unfamiliar APIs; as a result, all participants in these two tasks spent considerably more time in exploration, irrespective of other varying factors such as language expertise and prior Copilot usage. Advent of Code was more algorithmically challenging than the other tasks, and also involved the File I/O API, which was somewhat unfamiliar to participants. Both of these factors pushed participants to explore but there was more variance in the data than in Chat Client and Benford's Law: for example, P16, who figured out the algorithm early on, spent more time accelerating (15.8 minutes) than exploring (3.4 minutes). Chat Server, on the other hand, involved simple business logic, so participants leaned towards acceleration in this task.

Programming Language. We did not identify any noticeable differences in either total interaction time or ratio of acceleration to exploration between Python and Rust. For the other two languages (Haskell and Java), we have too few data points to make any conclusions.

5.1.2 ***Prompting Strategies across Interaction Modes.*** Our codebook identifies four strategies participants use to invoke Copilot: *code prompts*, *context prompts*, *comment prompts*, and the *multi-suggestions pane*. We can cluster the four prompting strategies into two categories: unintentional prompting (Sec. 2.1) and intentional prompting (Sec. 2.2). Unintentional prompting involves participants invoking Copilot without explicitly meaning to. For example, with *code prompts*, the participant is often simply writing code when Copilot pops up a suggestion to complete their partially written line of code. *Context prompts* are those where Copilot generates suggestions even when the participant is not actively writing code. From the language model perspective, these two kinds of prompts are indistinguishable but we consider them distinct from the user interaction perspective. Intentional prompting involves explicit intent from the participant. This can be in



(a) Aggregate time, split by interaction mode.

(b) Median time, grouped by task.

Fig. 7. Time spent in different validation strategies.

the form of writing a natural-language comment intended for Copilot (Sec. 4.2.3) or invoking the multi-suggestions pane by pressing `<ctrl> + <enter>` (Sec. 4.2.4).

Fig. 6 shows the aggregate percentage of times the 20 participants invoked Copilot using the four different prompting strategies. We notice that in acceleration mode, the most commonly used prompting strategy is code prompts (71.4%), with the other unintentional strategy, context prompts, coming in second (15.2%). The multi-suggestions pane is rarely used, which is consistent with our theory, since it would break the participant's flow. In exploration mode, participants intentionally prompt with comments a lot more than in acceleration mode (57.2% vs 13.1%). The percentage of multi-suggestion pane prompts also shoots up in exploration mode as it provides a rich body of suggestions for participants to explore from.

5.1.3 ***Validation Strategies across Interaction Modes and Tasks.*** Recall that Sec. 4.2.5 identified four different validations strategies: *examination*, *execution*, *static analysis*, and consulting the *documentation*. We measured the time participants spent in each of these strategies, with the exception of static analysis, which runs automatically in the background, so it was hard for us to determine precisely when a participant was "using" its results.

Fig. 7a shows the percentage of validation time spent in each strategy, split by interaction mode. Predictably, participants spent more time reading *documentation* in exploration mode than in acceleration mode, likely because exploration was commonly used when interfacing with unfamiliar APIs. A somewhat surprising result is that *execution* seems to be more prevalent during acceleration. One reason for this is that during exploration the code is often incomplete and cannot be executed. Another reason is simply that the remaining strategy, *examination*, takes up more time in absolute terms during exploration, as participants carefully examine the code line by line as opposed to making quick decisions via "pattern matching". We conclude that in exploration mode, programmers use validation strategies that *aid comprehension* (careful examination, reading documentation), while in acceleration more, they focus on strategies that provide *rapid feedback* on code correctness (execution).

The nature of the task also has impact on the validation strategies, as shown in Fig. 7b. The prevalence of complex and unfamiliar APIs in Chat Client both increases the overall validation time for this task and favors exploratory validation, such as examination and documentation. Interestingly, the task with most time spent in execution is Benford's Law, and not Advent of Code,

which was explicitly designed to be easy to test (it came with a test case). We conjecture that Benford's Law was executed so often because it has visual output, which is easy and exciting for programmers to inspect.

5.2 Qualitative Analysis of LiveStreams

We gathered additional evidence in the form of five livestream videos to support our theory. We present our findings from a qualitative analysis of these videos in this section.

5.2.1 ***Data Collection.*** The livestream videos were taken from Youtube (S1, S2, S4) and Twitch (S3, S5), and involved a developer using Copilot while constantly talking aloud to an audience. S1 and S2 had Copilot turned on to solve Advent of Code tasks in Haskell and C# respectively. S3, S4 and S5 all did web-based programming tasks using Copilot in Javascript, Typescript, HTML, SCSS, and other web languages. For example, S4's task was to build a Go game in Angular. While S1, S2 and S4 had well-defined tasks, S3 and S5 used Copilot for exploratory tasks, in fact, S3 even asked their viewers to suggest random programming tasks for Copilot.

5.2.2 ***Qualitative Data Analysis.*** One of the authors coded all the livestream videos with the same closed codebook used to re-code our participant videos in Sec. 5.1. We present the results from our qualitative analysis and draw parallels to our bimodal theory of acceleration and exploration.

5.2.3 ***Acceleration Mode.*** We observed that when the task was relatively well-defined (S1, S4, S5), acceleration mode was prevalent, consistent with our theory. All streamers used Copilot for end-of-line completions in acceleration mode at least once, accompanied with comments like, "Yeah Copilot knows what I'm trying to do!" In fact, S4 used Copilot only for end-of-line completions and said, "I need to let the AI help more, I'm doing too much stuff myself." Streamers would only focus on small logical units, for instance, S2 accepted a long suggestion only to retain the structure of a for loop and the condition within. S2 repeated this behavior when they just wanted to fill in the parameters of a function so they ended up deleting everything in a suggestion except the parameters. S1 often used end-of-line completions to complete type signatures in Haskell, which would correspond to a logical unit. As observed in our theory, the streamers would reject long suggestions that broke their flow (S1, S2, S4). S4 exclaimed, "Thank you, that's not what I want" when Copilot suggested an extremely long snippet while they were accelerating. In addition, both S1 and S4 made only minor edits to suggestions accepted in acceleration mode, whereas S1 made relatively major edits to suggestions in exploration.

5.2.4 ***Exploration Mode.*** S3 and S5, who worked on exploratory tasks, spent considerably more time in exploration mode than in acceleration. Streamers were willing to write a lot of comments while in exploration mode (S2, S3, S5). S5 tried to use Copilot to generate documentation and said, "as a person who usually writes comments after writing code, Copilot might change the way I code". S3 had an interesting way of prompting Copilot: by writing unusually descriptive function names instead of comments. S3 and S5 often used the multi-suggestion pane as a fallback option when the inline suggestions did not meet their expectations. S3 expected the multiple suggestions to be diverse and was sometimes disappointed when they were not. In addition to using the pane, S5 also explored multiple suggestions inline by pressing tab. We did not observe this behavior in our main study, because neither we nor our participants were aware of this feature.

5.2.5 ***Validation Strategies.*** We observed the same validation behavior as seen in our theory in all the livestream videos. After accepting a suggestion, S3 said, "Let's just check if this part works" and S1 echoed, "I think Copilot wrote that for me, let me just check". S1 and S2 constantly validated their code using the test inputs provided by the Advent of Code tasks and also used

specific test inputs for debugging code. All streamers spent considerable time in code examination as a validation strategy both inline (S1, S2, S4, S5) and in the multi-suggestion pane (S3, S5). S2 and S3 referred to API documentation using web search to validate Copilot's code while S4 resorted to reading in-IDE API documentation as a form of validation. S3 and S4 whose tasks involved building a website ran their webpage remotely as a validation strategy.

The blame game of who wrote the buggy code was also observed in the livestreams. While debugging their code, S5 expressed this by saying, "not sure if they are my bugs or Copilot's bugs." S3 had a bug that they were baffled by, turns out it was some residual code from Copilot's suggestion which they forgot to delete. S5 summed up Copilot's behavior as being a "mixed bag, when it understands what I want it feels like it's reading my mind. Otherwise it produces random code." Streamers were generally confident using Copilot for writing boilerplate, repetitive code (S3, S4).

6 RECOMMENDATIONS

This section outlines recommendations for how programming assistants could be improved in the future, We classify these suggestions into two categories: improving the way programmers could provide *input* to a future tool, and improving the kinds of *output* the tool could generate.

6.1 Better Input

Control over the context. There was general confusion among participants about how Copilot uses their code to provide suggestions. Some participants were unsure how much code Copilot can take into context, for example, P8 theorized a hard limit to the input length: "I think the README is too long and complicated for it to actually extract [helpful information]". Other participants (P8, P10, P15, P18) mentioned they were unsure about which pieces of information Copilot had extracted about their local codebase. Specifically, there appeared to be a broad misconception that commenting out code made it invisible to Copilot, despite those same participants using comment prompts. P20 "assumed it wouldn't be aware of code if [they] commented it out". We also observed participants (P2, P3, P4 P6) comment out code generated by Copilot in an attempt to get it to generate an alternative suggestion.

Participants that *were* aware of Copilot's sensitivity to context wanted to have more control over that context. Some participants wanted to give Copilot *specific context*: in describing their work outside of the study, P15 mentioned poor suggestions from Copilot and wished they could emphasize a subset of their code (*i.e.* niche libraries they imported), so they could feel more confident that the suggestions were relevant to their code. Others, P4 and P12, wished to query Copilot with a natural-language prompt *without* any code context, just as they would query StackOverflow.

In order to achieve this control, participants wanted Copilot to provide dedicated *syntax*. For example, P2 wanted Copilot to use a specific function, and tried to achieve this by "using the function name in backquotes". P18 asked: "Is there a way to prompt Copilot into suggesting a data structure?" Finally, P4, when looking for examples of using the `del` operation in Python, wanted to explicitly ask Copilot to show only "syntax examples".

Based on these observations, future tools could give programmers ways to customize the context. For example, a future tool could provide a scratchpad to isolate general, StackOverflow-style prompts from the rest of the codebase. It could also provide expert prompt syntax, similar to advanced operators in Google search; for example, including `:use plt.show()` in a comment prompt might restrict the assistant's suggestions to only those snippets using the expression `plt.show()`, like the work of [Peleg et al. 2018]. Finally, programmers would likely appreciate a separate type of comments that make code invisible to the tool.

Cross-language translation. P13 said that they were more familiar with Julia than the task language (Python), and at some point they wrote some Julia code which Copilot then translated to Python. This type of interaction opens up the possibility of users giving prompts in programming languages they are more familiar with. The task for Copilot then becomes a cross-language translation task. It would be interesting to fine-tune Copilot for this particular task, by training it on equivalence classes of syntactic constructs in different programming languages.

6.2 Better Output

Awareness of the interaction mode. Perhaps the most important outcome of our study is the bimodal nature of programmers' interactions with Copilot: they are either in an acceleration or exploration mode. We conjecture that the user experience could be improved if the tool were aware of the current interaction mode and adjusted its behavior accordingly. In acceleration mode, it should not break the programmer's flow (P6 mentioned that they intentionally turned Copilot off because it disrupted their workflow). To this end, the tool should avoid low-confidence suggestions—which are unlikely to be accepted—and long suggestions—which distract the programmer.

Going beyond simply avoiding multi-line suggestions, the tool could be made more aware of how the code is divided into logical units. As we mentioned in Sec. 4.1.2, programmers in acceleration mode focus on a single logical unit of code at a time, which is often one line, but can also be shorter (the next function call in Haskell) or longer (an entire loop). It would be interesting to explore if we can make the scope of Copilot's suggestions match the scope the programmer's current focus. Participants also mentioned that it would be helpful if Copilot gave suggestions more selectively as opposed to being always on. This could be achieved, *e.g.*, by reinforcement learning to obtain a policy for when Copilot should intervene, based on the local context and programmer's actions.

Exploring multiple suggestions. As we mentioned in Sec. 4.2.4, in exploratory searches, programmers commonly used the multi-suggestion pane, but also often got overwhelmed by the results they saw there. Several participants had trouble identifying meaningful differences between the suggestions (P1, P4, P6, P7, P13). This observation motivates the need for a tool that would help programmers explore a large space of suggestions, perhaps similarly to how Overcode [Glassman et al. 2015] supports exploring a space of student solutions to a programming assignment.

Suggestions with holes. Recall from Sec. 4.2.6, that when programmers modify suggestions, they often keep control-flow features and little else, as seen for P1, P2, P8, P11, P17, and P18. Based on this observation, programmers would likely benefit from *suggestions with holes*, where the tool only generates control structures, which users are likely to understand quickly, leaving their bodies for the programmer to fill out (either by hand, or by giving more targeted prompts to the tool). For example, P2 explicitly mentioned that "if [Copilot] gives me a mostly filled out skeleton, I can be the one who fills out holes". Recent work by [Guo et al. 2021] generated holes in their suggestions where the underlying model had low confidence.

Low-confidence suggestions are not the only motivation for a hole: participants reported feeling frustrated and distracted by large code snippets. When offered these large snippets, some participants felt Copilot was forcing them to jump in to write code before coming up with a high-level architectural design. P4 said:

> "I wrote code as one might read code, rather than the way I might write it which is generally top-down, where I will fill in the control structure and then I'll do the little bits and pieces after I build in the full control structure. It made me jump in to write code instead of the normal way."

P16 normally writes a high-level design first and then gets to function implementations—as the grounded theory from [Lubin and Chasins 2021] describes of functional programmers. Other participants (P2, P3, P4, P5, P7, P8) also felt Copilot forced significant change on their code authorship process. Based on our observations, future tools should mind how large code blocks can break the user's natural development flow, instead offering code holes for users to fill in when ready.

Always-on validation. Several participants (P2, P14, P16) wished to have better tool support for validating suggestions. For example, P16 wanted to set up property-based testing [Claessen and Hughes 2000] to run automatically on Copilot suggestions. P14 wished they had *projection boxes* [Lerner 2020], a live programming environment that constantly displays runtime values of relevant variables (usually on a single test input). In the future, IDEs could couple code-generating models with some kind of always-on validation, in order to make the process of evaluating code suggestions less taxing for the developer.

7 RELATED WORK

Usability of Copilot. The closest to our work is the study by Vaithilingam et al. [2022], which also evaluates Copilot. The main differences are: (1) their study is on stand-alone tasks, whereas ours includes tasks that require contributing to an existing codebase; (2) their study is comparative and focuses on the rate and time of task completion with and without Copilot's help; (3) their study only used Python, whereas we used several programming languages. In our study, we explicitly stepped away from the common comparative setting, where participants are given well-defined stand-alone tasks, and the goal is to collect quantitative data on how well and quickly they complete the tasks, with and without the tool under evaluation. Instead, we chose more open-ended tasks in the context of an existing codebase, which we believe is closer to the real-world use case. Further, instead of skewing quantitative answers to predefined research questions, we chose the grounded theory approach, with the general goal of finding patters in programmers' behavior when they interact with Copilot; we believe this approach is complementary to the quantitative studies. Finally, our usage of multiple languages enables inter-language comparisons and more generalizable conclusions.

On the other hand, Vaithilingam et al. [2022] also report several qualitative findings. Most of them agree with ours, such as: that Copilot often provides a good starting point for programmers who do not know how to approach the task, that programmers are generally willing to repair code suggestions, but Copilot-generated code is harder to debug. There are also some differences; for example, half of their participants (12/24) said they had trouble understanding and modifying Copilot-generated code, whereas our participants did not seem to share this difficulty; this might be because our study is with more experienced developers: only one participant in our study was an undergraduate student, whereas 10/24 in their study were undergraduates.

Usability of other LLM tools. Beyond Copilot, Jiang et al. [2022] conducted a user study to analyse the interaction of developers with a natural language to code tool called GenLine. GenLine is similar to Copilot but involves explicitly invoking a command within a text editor. Similar to our findings, developers in their study were willing to rewrite the natural language prompt to clarify their intent and expressed the need for a syntax to communicate with the model more clearly. However, their findings were mainly centered around prompting strategies whereas we did a more comprehensive analysis of developer interactions with Copilot. Moreover, the tool was not integrated in the participant's daily workflow like in our study. In a similar vein, Xu et al. [2021] investigated the usefulness of an NL-to-code plugin previously developed by the same authors [Xu et al. 2020]. They found no statistically significant difference in task completion times or correctness scores when using the plugin, and the participants' feedback about the plugin was neutral to slightly positive. We conjecture, however, that these findings are not as relevant anymore, thanks to recent

breakthroughs in large language models, which significantly increased the quality of generated code. In another related study, Weisz et al. [2021] interviewed IBM software engineers about their experience with a neural machine translation tool for translating code between programming languages. This study focuses on the engineers' code validation strategies and future UI features that might help with this task, such as confidence highlighting and alternative translations; in this sense, their study is complementary to our work, conducted in the context of a different task (language-to-language translation).

[Sarkar et al. 2022] compiled observations from the above user studies and additionally gathered experience reports of programming assistants usage from Hacker News. The compiled observations were similar to what we found—prompting is hard, validation is important, and programmers use assistants for boilerplate, reusable code. There are a few other industrial-grade programming assistants powered by statistical models, such as TabNine [TabNine 2018] and Kite [Kite 2020], but we are not aware of any research on their usability.

Usability of program synthesis tools. Another approach to code generation, is the more traditional, search-based program synthesis. As program synthesis technology matures, it becomes increasingly common to evaluate the usability of synthesizers on human subjects. Many of these usability studies are for domain-specific synthesizers targeting API navigation [James et al. 2020], regular expressions [Zhang et al. 2021, 2020], web scraping [Chasins et al. 2018], or data querying, wrangling, and visualization [Drosos et al. 2020; Wang et al. 2021; Zhou et al. 2022]. These studies usually focus on measuring the tool's effect on task completion rates and times, which is less relevant to our questions. The work on RESL [Peleg et al. 2020] and Snippy [Ferdowsifard et al. 2021, 2020] include user studies of general-purpose programming-by-example synthesizers for JavaScript and Python. Although both also focus mainly on task completion times, they do make some interesting qualitative observations. For example, Ferdowsifard et al. [2020] observe that one of the main barriers to the usefulness of the synthesizer is the so-called *user-synthesizer gap*, *i.e.* the programmer's overestimation of the synthesizer's capabilities; we observed a similar phenomenon in our study (see Sec. 4.2.2), although it appears to be less prominent in LLM-based tools, since their performance degrades more gradually with the complexity of the task.

[Jayagopal et al. 2022] study how undergraduate students learned to use six different synthesizers—Copilot among them—with different interaction modes. Not all of the themes they identify are applicable to Copilot, but those that are, are corroborated and explored in more depth in our study. For example, they identify that novice participants would often accept then modify code. We support and extend this (Sec. 4.2.6), adding that this is characteristic of exploration mode, which indeed occurs more commonly in novices.

Grounded Theory for software development. Grounded Theory (GT) has a relatively long history in software-related fields, with its application to software engineering dating as far back as 2004 [Carver 2004]. [Stol et al. 2016] provide a survey and a critical evaluation of 93 GT studies in software engineering. Recently, GT has also drawn interest in the programming languages community: Lubin and Chasins [2021] study how statically-typed functional programmers write code, and deliver a set of guidelines meant for functional language tool-builders.

8 LIMITATIONS AND THREATS TO VALIDITY

Our participants worked on tasks of our design, as opposed to their own projects. If they were working in a more familiar codebase and without the time pressure of a study, their interactions could have been different. Moreover, our tasks focused only on code authorship, as opposed to refactoring, testing, debugging, or other common aspects of software engineering. We consider

these beyond the scope of this study, although our participants did occasionally get a chance to test or debug their code.

We recruited 20 participants, with a skew towards those in academia, hardly a representative sample of all programmers. Similarly, although we tried to diversify the type of tasks our participants were solving and the programming languages they were using, other kinds of tasks and languages could have lead to different interactions.

11 of our participants had not used Copilot before the study, and hence might not be representative of regular users of the tool. We gave all participants a 5-minute training task so they could familiarize themselves with Copilot, and yet we observed that first-time users were sometimes over-reliant on Copilot, in a way that prior users were not. We chose to include new users in our study since the majority of programmers in the wild have never used a code-generating model. Meanwhile, our participants who already had access to the tool may have formed a usage pattern (or dis-uage pattern in the case of P6) based on poor experience early in the technical preview, where its behavior may have been rapidly changing. Ideally, we would have liked to observe programmer over a longer period of time, in order to study how their usage patterns changed over time, but this was not feasible given the time constraints of the study.

Finally, the research on code-generating models is progressing very rapidly, and it is possible that new technological breakthroughs will soon render our findings obsolete. That would be a nice problem to have indeed!

ACKNOWLEDGMENTS

The authors would like to thank Devon Rifkin from GitHub for his assistance with getting Copilot access for our study participants. This work was supported by NSF grants 2107397 and 1955457.

REFERENCES

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989* (2016).

Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. *Grounded Copilot: How Programmers Interact with Code-Generating Models.* https://doi.org/10.5281/zenodo.7713789

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. Efficient Training of Language Models to Fill in the Middle. arXiv:2207.14255 (Jul 2022). https://doi.org/10.48550/arXiv.2207.14255 arXiv:2207.14255 [cs].

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. *Advances in neural information processing systems* 33 (2020), 1877–1901.

Antony Bryant and Kathy Charmaz. 2007. *The SAGE Handbook of Grounded Theory.* SAGE Publications Ltd. https://doi.org/10.4135/9781848607941

Donal E Carlston. 2013. *Dual-Process Theories.* http://public.ebookcentral.proquest.com/choice/publicfullrecord.aspx?p=1336453

Jeff Carver. 2004. The Impact of Background and Experience on Software Inspections. *Empirical Softw. Engg.* 9, 3 (sep 2004), 259–262. https://doi.org/10.1023/B:EMSE.0000027786.04555.97

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* (Berlin, Germany) *(UIST '18).* Association for Computing Machinery, New York, NY, USA, 963–975. https://doi.org/10.1145/3242587.3242661

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).

Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming (ICFP'00).* Association for Computing Machinery, 268–279. https://doi.org/10.1145/351240.351266

Michael Coblenz, Gauri Kambhatla, Paulette Koronkevich, Jenna L. Wise, Celeste Barnaby, Joshua Sunshine, Jonathan Aldrich, and Brad A. Myers. 2021. PLIERS: A Process that Integrates User-Centered Methods into Programming Language

Design. *ACM Transactions on Computer-Human Interaction* 28, 4 (Jul 2021), 28:1–28:53. https://doi.org/10.1145/3452379

Mihaly Csikszentmihalyi. 2014. *Flow and the Foundations of Positive Psychology.* Springer Netherlands. https://doi.org/10.1007/978-94-017-9088-8

Françoise Détienne and Frank Bott. 2001. *Software design—cognitive aspects.* Springer-Verlag.

Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. 2020. *Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists.* Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3313831.3376442

Kasra Ferdowsifard, Shraddha Barke, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2021. LooPy: Interactive Program Synthesis with Control Structures. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 153 (oct 2021), 29 pages. https://doi.org/10.1145/3485530

Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2020. Small-Step Live Programming by Example. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology.* Association for Computing Machinery, New York, NY, USA, 614–626. https://doi.org/10.1145/3379337.3415869

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. InCoder: A Generative Model for Code Infilling and Synthesis. arXiv:2204.05999 (Apr 2022). https://doi.org/10.48550/arXiv.2204.05999 arXiv:2204.05999 [cs].

Nat Friedman. 2021. https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer/

Barney G. Glaser and Anselm L. Strauss. 1967. *The discovery of grounded theory: strategies for qualitative research* (5. paperback print ed.). Aldine Transaction.

Elena L. Glassman, Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller. 2015. OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale. *ACM Trans. Comput.-Hum. Interact.* 22, 2, Article 7 (mar 2015), 35 pages. https://doi.org/10.1145/2699751

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

Daya Guo, Alexey Svyatkovskiy, Jian Yin, Nan Duan, Marc Brockschmidt, and Miltiadis Allamanis. 2021. Learning to Complete Code with Sketches. In *International Conference on Learning Representations.*

Michael B. James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Ranjit Jhala, and Nadia Polikarpova. 2020. Digging for Fold: Synthesis-Aided API Discovery for Haskell. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 205 (nov 2020), 27 pages. https://doi.org/10.1145/3428273

Dhanya Jayagopal, Justin Lubin, and Sarah E Chasins. 2022. Exploring the Learnability of Program Synthesizers by Novice Programmers. (2022), 15.

Ellen Jiang, Edwin Toh, Alejandra Molina, Kristen Olson, Claire Kayacik, Aaron Donsbach, Carrie J Cai, and Michael Terry. 2022. Discovering the Syntax and Strategies of Natural Language Programming with Generative Language Models. In *CHI Conference on Human Factors in Computing Systems.* 1–19.

Daniel Kahneman. 2011. *Thinking, fast and slow.* Penguin Books.

Ashwin Kalyan, Abhishek Mohta, Oleksandr Polozov, Dhruv Batra, Prateek Jain, and Sumit Gulwani. 2018. Neural-guided deductive search for real-time program synthesis from examples. *arXiv preprint arXiv:1804.01186* (2018).

Kite. 2020. Kite: AI-Powered Completions for JupyterLab. https://www.kite.com/integrations/jupyter/.

Matthew Lee. 2020. Detecting Affective Flow States of Knowledge Workers Using Physiological Sensors. *arXiv:2006.10635 [cs]* (Jun 2020). http://arxiv.org/abs/2006.10635 arXiv: 2006.10635.

Sorin Lerner. 2020. Projection Boxes: On-the-fly Reconfigurable Visualization for Live Programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems.* ACM, 1–7. https://doi.org/10.1145/3313831.3376494

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-Level Code Generation with AlphaCode. https://doi.org/10.48550/ARXIV.2203.07814

Justin Lubin and Sarah E. Chasins. 2021. How statically-typed functional programmers write code. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (Oct 2021), 1–30. https://doi.org/10.1145/3485532

Smitha Milli, Falk Lieder, and Thomas L. Griffiths. 2021. A rational reinterpretation of dual-process theories. *Cognition* 217 (2021), 104881. https://doi.org/10.1016/j.cognition.2021.104881

Anders Miltner, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. 2019. On the fly synthesis of edit suggestions. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.

Michael Muller. 2014. *Curiosity, Creativity, and Surprise as Analytic Tools: Grounded Theory Method.* Springer, 25–48. https://doi.org/10.1007/978-1-4939-0378-8_2

Brad A. Myers, Amy J. Ko, Thomas D. LaToza, and YoungSeok Yoon. 2016. Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools. *Computer* 49, 7 (Jul 2016), 44–52. https://doi.org/10.1109/MC.2016.200

Wode Ni, Joshua Sunshine, Vu Le, Sumit Gulwani, and Titus Barik. 2021. reCode: A Lightweight Find-and-Replace Interaction in the IDE for Transforming Code by Example. In *The 34th Annual ACM Symposium on User Interface Software and Technology*. 258–269.

Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. *Proc. ACM Program. Lang.* 3, POPL (Jan 2019), 14:1–14:32. https://doi.org/10.1145/3290327

Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2021. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. *arXiv preprint arXiv:2108.09293* (2021).

Hila Peleg, Roi Gabay, Shachar Itzhaky, and Eran Yahav. 2020. Programming with a Read-Eval-Synth Loop. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 159 (nov 2020), 30 pages. https://doi.org/10.1145/3428227

Hila Peleg, Sharon Shoham, and Eran Yahav. 2018. Programming Not Only by Example. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, 1114–1124. https://doi.org/10.1145/3180155.3180189 tex.ids: pelegProgrammingNotOnly2018a event-place: Gothenburg, Sweden.

Nancy Pennington. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology* 19, 3 (Jul 1987), 295–341. https://doi.org/10.1016/0010-0285(87)90007-7

Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 419–428.

Advait Sarkar, Andrew D Gordon, Carina Negreanu, Christian Poelitz, Sruti Srinivasa Ragavan, and Ben Zorn. 2022. What is it like to program with artificial intelligence? *arXiv preprint arXiv:2208.06213* (2022).

Armando Solar-Lezama. 2013. Program sketching. *International Journal on Software Tools for Technology Transfer* 15, 5–6 (Oct 2013), 475–495. https://doi.org/10.1007/s10009-012-0249-7

Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. 2016. Grounded theory in software engineering research: a critical review and guidelines. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. Association for Computing Machinery, 120–131. https://doi.org/10.1145/2884781.2884833

Anselm L. Strauss and Juliet Corbin. 1990. *Basics of Qualitative Reseach: Grounded Theory Procedures and Techniques*. SAGE Publications, Inc.

TabNine. 2018. TabNine: AI Assistant for Development Teams. https://www.tabnine.com/.

Steven J. Taylor and Robert Bogdan. 1998. *Introduction to qualitative research methods: A guidebook and resource, 3rd ed.* John Wiley and Sons Inc.

Priyan Vaithilingam, Tianyi Zhang, and Elena Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *CHI Late-Breaking Work*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. arXiv:1706.03762 (Dec 2017). https://doi.org/10.48550/arXiv.1706.03762 arXiv:1706.03762 [cs].

Regina Vollmeyer and Falko Rheinberg. 2006. Motivational Effects on Self-Regulated Learning with Different Tasks. *Educational Psychology Review* 18, 3 (Nov 2006), 239–253. https://doi.org/10.1007/s10648-006-9017-0

Chenglong Wang, Yu Feng, Rastislav Bodik, Isil Dillig, Alvin Cheung, and Amy J Ko. 2021. Falx: Synthesis-Powered Visualization Authoring. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 106, 15 pages. https://doi.org/10.1145/3411764.3445249

Eric Wastl. 2021. Advent of Code. https://adventofcode.com/2021.

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*. Association for Computing Machinery, New York, NY, USA, 402–412. https://doi.org/10.1145/3397481.3450656

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. https://doi.org/10.48550/ARXIV.2004.09015

Frank F. Xu, Bogdan Vasilescu, and Graham Neubig. 2021. In-IDE Code Generation from Natural Language: Promise and Challenges. https://doi.org/10.48550/ARXIV.2101.11149

Wojciech Zaremba, Greg Brockman, and OpenAI. 2021. Codex. https://openai.com/blog/openai-codex/.

Tianyi Zhang, Zhiyang Chen, Yuanli Zhu, Priyan Vaithilingam, Xinyu Wang, and Elena L. Glassman. 2021. Interpretable Program Synthesis. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 105, 16 pages. https://doi.org/10.1145/3411764.3445646

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*.

Association for Computing Machinery, New York, NY, USA, 627–648. https://doi.org/10.1145/3379337.3415900
Xiangyu Zhou, Ras Bodik, Alvin Cheung, and Chenglong Wang. 2022. Synthesizing Analytical SQL Queries from Computation Demonstration. In *PLDI*.

Received 2022-10-28; accepted 2023-02-25