Validating AI-Generated Code with Live Programming

Kasra Ferdowsi[*]
kferdows@ucsd.edu
UC San Diego
San Diego, CA, USA

Ruanqianqian (Lisa) Huang[*]
r6huang@ucsd.edu
UC San Diego
San Diego, CA, USA

Michael B. James
m3james@ucsd.edu
UC San Diego
San Diego, CA, USA

Nadia Polikarpova
npolikarpova@ucsd.edu
UC San Diego
San Diego, CA, USA

Sorin Lerner
lerner@cs.ucsd.edu
UC San Diego
San Diego, CA, USA

ABSTRACT

AI-powered programming assistants are increasingly gaining popularity, with GitHub Copilot alone used by over a million developers worldwide. These tools are far from perfect, however, producing code suggestions that may be incorrect in subtle ways. As a result, developers face a new challenge: *validating* AI's suggestions. This paper explores whether Live Programming (LP), a continuous display of a program's runtime values, can help address this challenge. To answer this question, we built a Python editor that combines an AI-powered programming assistant with an existing LP environment. Using this environment in a between-subjects study ($N = 17$), we found that by lowering the cost of validation by execution, LP can mitigate over- and under-reliance on AI-generated programs and reduce the cognitive load of validation for certain types of tasks.

CCS CONCEPTS

• **Human-centered computing → Empirical studies in HCI**; **Graphical user interfaces**; • **Software and its engineering → Automatic programming**.

KEYWORDS

Live Programming, AI Assistants

ACM Reference Format:
Kasra Ferdowsi, Ruanqianqian (Lisa) Huang, Michael B. James, Nadia Polikarpova, and Sorin Lerner. 2024. Validating AI-Generated Code with Live Programming. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '24), May 11–16, 2024, Honolulu, HI, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3613904.3642495

1 INTRODUCTION

Recent advances in large language models have given rise to AI-powered code suggestion tools like GitHub Copilot [12], Amazon CodeWhisperer [1], and ChatGPT [23]. These *AI programming assistants* are changing the face of software development, automating many of the traditional programming tasks, but at the same time introducing *new tasks* into the developer's workflow—such as prompting the assistant and reviewing its suggestions [2, 22]. Development environments have some catching up to do in order to provide adequate tool support for these new tasks.

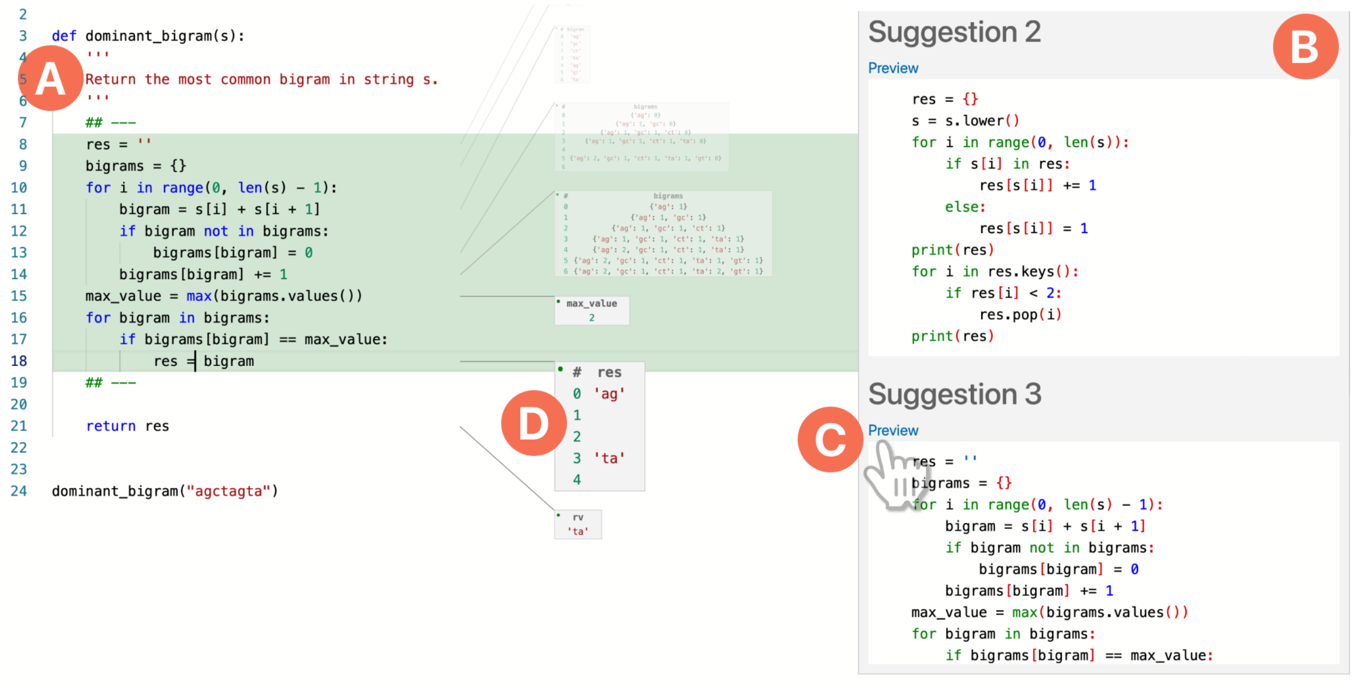
In this paper, we focus on the task of *validating* AI-generated code, *i.e.*, deciding whether it matches the programmer's intent. Recent studies show that validation is a bottleneck for AI-assisted programming: according to Mozannar et al. [22], it is the *single most prevalent activity* when using AI code assistants, and other studies [3, 21, 32, 36] report programmers having trouble evaluating the correctness of AI-generated code. Faced with difficulties in validation, programmers tend to either *under-rely* on the assistant—*i.e.*, lose trust in it—or to *over-rely*—*i.e.*, blindly accept its suggestions [27, 30, 34, 37]; the former can cause them to abandon the assistant altogether [2], while the latter can introduce bugs and security vulnerabilities [26]. These findings motivate the need for better validation support in AI-assisted programming environments.

This paper investigates the use of *Live Programming* (LP) [13, 31, 35] as a way to support the validation of AI-generated code. LP environments, such as Projection Boxes [20], visualize runtime values of a program in real-time without any extra effort on the part of the programmer. We hypothesize that these environments are a good fit for validation, since LP has been shown to encourage more frequent testing [4] and facilitate bug finding [41] and program comprehension [5, 7, 8]. On the other hand, validation of AI-generated code is a new and unexplored domain in program comprehension that comes with its unique challenges, such as multiple AI suggestions for the programmer to choose from, and frequent context switches between prompting, validation, and code authoring [22], which cause additional cognitive load [36]. Hence, the application of LP to the validation setting warrants a separate investigation.

To this end, we constructed a Python environment that combines an existing LP environment [20] with an AI assistant similar to Copilot's multi-suggestion pane. Using this environment, we conducted a between-subjects experiment ($N = 17$) to evaluate how the availability of LP affects users' effectiveness and cognitive load in validating AI suggestions. Our study shows that Live Programming facilitates validation through *lowering the cost of inspecting runtime values*; as a result, participants were more successful in evaluating the correctness of AI suggestions and experienced lower cognitive load in certain types of tasks.

[*]The first two authors, listed alphabetically, contributed equally.

CHI '24, May 11–16, 2024, Honolulu, HI, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0330-0/24/05
https://doi.org/10.1145/3613904.3642495



```
def dominant_bigram(s):
    '''
    Return the most common bigram in string s.
    '''
    ## ---
    res = ''
    bigrams = {}
    for i in range(0, len(s) - 1):
        bigram = s[i] + s[i + 1]
        if bigram not in bigrams:
            bigrams[bigram] = 0
        bigrams[bigram] += 1
    max_value = max(bigrams.values())
    for bigram in bigrams:
        if bigrams[bigram] == max_value:
            res = bigram
    ## ---

    return res


dominant_bigram("agctagta")
```

```
# bigram
0 'ag'
1 'gc'
2 'ct'
3 'ta'
4 'ag'
5 'gt'
6 'ta'
```

```
#                  bigrams
0                 {'ag': 0}
1            {'ag': 1, 'gc': 0}
2       {'ag': 1, 'gc': 1, 'ct': 0}
3  {'ag': 1, 'gc': 1, 'ct': 1, 'ta': 0}
4
5 {'ag': 2, 'gc': 1, 'ct': 1, 'ta': 1, 'gt': 0}
6
```

```
#                  bigrams
0                 {'ag': 1}
1            {'ag': 1, 'gc': 1}
2       {'ag': 1, 'gc': 1, 'ct': 1}
3  {'ag': 1, 'gc': 1, 'ct': 1, 'ta': 1}
4  {'ag': 2, 'gc': 1, 'ct': 1, 'ta': 1}
5 {'ag': 2, 'gc': 1, 'ct': 1, 'ta': 1, 'gt': 1}
6 {'ag': 2, 'gc': 1, 'ct': 1, 'ta': 2, 'gt': 1}
```

```
max_value
    2
```

```
# res
0 'ag'
1
2
3 'ta'
4
```

```
rv
'ta'
```

```
res = {}
s = s.lower()
for i in range(0, len(s)):
    if s[i] in res:
        res[s[i]] += 1
    else:
        res[s[i]] = 1
print(res)
for i in res.keys():
    if res[i] < 2:
        res.pop(i)
print(res)
```

```
res = ''
bigrams = {}
for i in range(0, len(s) - 1):
    bigram = s[i] + s[i + 1]
    if bigram not in bigrams:
        bigrams[bigram] = 0
    bigrams[bigram] += 1
max_value = max(bigrams.values())
for bigram in bigrams:
    if bigrams[bigram] == max_value:
```

Figure 1: LEAP is a Python environment that enables validating AI-generated code suggestions via Live Programming. (A) Users prompt the AI assistant via comments and/or code context. (B) The Suggestion Panel shows the AI-generated suggestions. (C) Pressing a `Preview` button inserts the suggestion into the editor. (D) Users can inspect the runtime behavior of the suggestion in Projection Boxes [20], which are updated continuously as the user edits the code.

2 RELATED WORK

Validation of AI-Generated Code. A rapidly growing body of work analyzes how users interact with AI programming assistants. Studies show that programmers spend a significant proportion of their time validating AI suggestions [2, 3, 22]. Moreover, a large-scale survey [21] indicates that 23% of their respondents *have trouble evaluating correctness of generated code*, which echoes the findings of lab studies [2, 32] and a need-finding study [36], where participants report difficulties understanding AI suggestions and express a desire for better validation support. Barke et al. [2] and Liang et al. [21] find that programmers use an array of validation strategies, and the prevalence of each strategy is *closely related to its time cost*. Specifically, despite the help of execution techniques built into the IDE for validating AI suggestions [30], execution is used less often than quick manual inspection or type checking because it is more time-consuming [2, 21] and interrupts programmers' workflows [36]. The lack of validation support designed for AI-assisted programming, as Wang et al. [36] identify, leads to a higher cognitive load in reviewing suggestions. The high cost of validating AI suggestions, according to some studies [27, 34, 37], can lead to both *under-reliance*—lack of trust—and *over-reliance*—uncritically accepting wrong code—on the part of the programmer.

Comparatively fewer existing papers explore interface designs to support validation of AI-generated code: Ross et al. [27] investigates a conversational assistant that allows programmers to ask questions about the code, while Vasconcelos et al. [33] targets over-reliance by highlighting parts of generated code that might need human intervention; our work is complementary to these efforts in that it focuses on facilitating validation by execution.

Validation in Program Synthesis. Another line of related work concerns the validation of code generated by *search-based* (non-AI-powered) program synthesizers. Several synthesizers help users validate generated code by proactively displaying its outputs [9, 16, 40] and intermediate trace values [25], although none of them use a LP environment. The only system we are aware of that combines LP and program synthesis is SNIPPY [11], but it uses LP to help the user specify their intent rather than validate synthesized code.

Live Programming. Live Programming (LP) provides immediate feedback on code edits, often in the form of visualizations of the runtime state [13, 31, 35]. Some quantitative studies find that programmers with LP find more bugs [41], fix bugs faster [18], and test a program more often [4]. Others find no effect in knowledge gain [15] or efficiency in code understanding [5]. Still, qualitative evidence points to the helpfulness of LP for program comprehension [5, 7, 8] and debugging [15, 17]. In contrast to these studies, which evaluate the effectiveness of LP for comprehending and debugging *human-written* code, our work investigates its effectiveness for validating *AI-generated* code, a setting that comes with a number of previously unexplored challenges [22, 36].

3 LEAP: THE TOOL USED IN THE STUDY

To study how Live Programming affects the validation of AI-generated code, we implemented LEAP (**L**ive **E**xploration of **A**I-Generated **P**rograms), a Python environment that combines an

AI assistant with LP. This section demonstrates Leap via a usage example and discusses its implementation.

Example Usage. Naomi, a biologist, is analyzing some genome sequencing data using Python. As part of her analysis, she needs to find the most common bigram (*i.e.*, two-letter sequence) in a DNA strand.[1] To this end, she creates a function `dominant_bigram` (line 3 in Fig. 1); she has a general idea of what this function might look like, but she decides to use Leap to help translate her idea into code.

- A Naomi adds a docstring (line 5), which conveys her intent in natural language, and a test case (line 24), which will help her validate the code. With the cursor positioned at line 7, she presses Ctrl and Enter to ask for suggestions.
- B Within seconds, a panel opens on the right containing five AI-generated code suggestions; Naomi quickly skims through all of them. The overall shape of Suggestion 3 looks most similar to what she has in mind: it first collects the counts of all bigrams into a dictionary, and then iterates through the dictionary to pick a bigram with the maximum count.
- C Naomi tries this suggestion, pressing its `Preview` button; Leap inserts the code into the editor and highlights it (lines 8-18).
- D As soon as the suggestion is inserted, Projection Boxes [20] appear, showing runtime information at each line in the code. Inspecting intermediate values helps Naomi understand what the code is doing step by step. When she gets to line 18, she realizes that the dictionary actually has *two* dominant bigrams with the same count, and the code returns *the last one*. She realizes this is not what she wants: instead, she wants to select the dominant bigram that comes first alphabetically (`ag` in this case).

One option Naomi has is to try other suggestions. She clicks on the `Preview` button for Suggestion 2; Leap then inserts Suggestion 2 into the editor, in place of the prior suggestion, and the Projection Boxes update instantly to show its behavior. Naomi immediately notices that Suggestion 2 throws an exception inside the second loop, so she abandons it and goes back to Suggestion 3, which got her closer to her goal.

To fix Suggestion 3, Naomi realizes that she can accumulate all dominant bigrams in a list, sort the list, and return the first element. She does not remember the exact Python syntax for sorting a list, so she tries different variations—including `l = l.sort`, `l = l.sort()`, `l = sort(l)`, `l = l.sorted()`, and so on. Fortunately, Leap's support for LP allows her to get instant feedback on the behavior of each edit, so she iterates quickly to find one correct option: `l = sorted(l)`. Note that Naomi's workflow for using Suggestion 3—validation, finding bugs, and fixing bugs—relies on full LP support, and would not work in traditional environments like *computational notebooks*, which provide easy access to the final output of a snippet but not the intermediate values or immediate feedback on edits.

Implementation. To generate code suggestions, Leap uses the `text-davinci-003` model [24], the largest publicly available code-generating model at the time of our study. To support live display of runtime values (Fig. 1 D), we built Leap on top of Projection Boxes, a state-of-the-art LP environment for Python [20] capable of running in the browser. The code for Leap can be found at https://bit.ly/leap-code. As the control condition for our study, we also created a version of Leap, where Projection Boxes are disabled, and instead the user can run the code explicitly by clicking a `Run` button and see the output in a terminal-like Output Panel.

[1]This is one of the programming tasks from our user study, and each of Naomi's interactions with Leap has been observed in some of our participants.

4 USER STUDY

We conducted a between-subjects study to answer the following research questions:

RQ1) How does Live Programming affect over- and under-reliance in validating AI-generated code?
RQ2) How does Live Programming affect validation strategies?
RQ3) How does Live Programming affect the cognitive load of validating AI-generated code?

Tasks. Our study incorporates two categories of programming tasks, *Fixed-Prompt* and *Open-Prompt* tasks.

In *Fixed-Prompt tasks*, we provide participants with a *fixed set* of five AI suggestions that are intended to solve the entire problem. We curated the suggestions by querying Copilot [12] and Leap with slight variations of the prompt. Fixed-Prompt tasks isolate the effects of Live Programming on validation behavior by controlling for the quality of suggestions. We created two Fixed-Prompt tasks, each with five suggestions: (T1) *Bigram*: Find the most frequent bigram in a given string, resolving ties alphabetically (same task in Sec. 3); (T2) *Pandas*: Given a `pandas` data frame with data on dogs of three size categories (small, medium, and large), compute various statistics, imputing missing values with the mean of the appropriate category. These tasks represent two distinct styles: Bigram is a purely algorithmic task, while Pandas focuses on using a complex API. Pandas has two correct AI suggestions (out of five) while Bigram has none, a realistic scenario that programmers encounter with imperfect models.

In *Open-Prompt tasks*, participants are free to invoke the AI assistant however they want. This task design is less controlled than Fixed-Prompt, but more realistic, thus increasing ecological validity. We used two Open-Prompt tasks: (T3) *String Rewriting*: parse a set of string transformation rules and apply them five times to a string; (T4) *Box Plot*: given a `pandas` data frame containing 10 experiment data records, create a `matplotlib` box plot of time values for each group, combined with a color-coded scatter plot. Both tasks are more complex than the Fixed-Prompt tasks, and could not be solved with a single interaction with the AI assistant.

Participants and Groups. We recruited 17 participants; 5 self-identified as women, 10 as men, and 2 chose not to disclose. 6 were undergraduate students, 9 graduate students, and 2 professional engineers. Participants self-reported experience levels with Python and AI assistants: 2 participants used Python 'occasionally', 8 'regularly', and 7 'almost every day'; 7 participants declared they had 'never' used AI assistants, and 8 used such tools 'occasionally'.

There were two experimental groups: "LP" participants used Leap with Projection Boxes, as described in Fig. 1; "No-LP" participants used Leap *without* Projection Boxes, instead executing programs in a terminal-like Output Panel. Participants completed both Fixed-Prompt tasks and one Open-Prompt task. We used block randomization [10] to assign participants to groups while evenly distributing across task order and selection and balancing experience with Python and AI assistants across groups. The LP group had 8 participants, and No-LP had 9.

Procedure and Data. We conducted the study over Zoom as each participant used Leap in their web browser. Each session was recorded and included two Fixed-Prompt tasks (10 minutes each), two post-task surveys, one Open-Prompt task (untimed), one post-study survey, and a semi-structured interview. A replication package[2] shows the details of our procedure, tasks, and data collection.

For *quantitative* analysis, we performed closed-coding on video recordings of study sessions to determine each participant's *subjective* assessment of their success on the task; we matched this data against the *objective* correctness of their final code to establish whether they succeeded in accurately validating AI suggestions. We also measured task duration—proportion of time Suggestion Panel (Fig. 1 B) was in focus—and participants' cognitive load (via five NASA Task Load Index (TLX) questions [14]). We used Mann-Whitney U tests to assess all differences except for validation success, which we analyzed via Fisher's exact tests.

In addition, we collected *qualitative* data from both Fixed-Prompt and Open-Prompt tasks. We noted validation-related behavior and quotes, which we discussed in memoing meetings [6] after the study. Through reflexive interpretation, we used category analysis [39] to assemble the qualitative data into groups. We then revisited notes and recordings to iteratively construct high-level categories.

5 RESULTS

5.1 RQ1: Over- And Under-Reliance on AI

To investigate if Live Programming affects over- and under-reliance, we measured whether participants successfully validated the AI suggestions in the Fixed-Prompt tasks, as described below. We also compared task completion times and participants' confidence in their solutions (collected through post-task surveys). However, neither result was significantly different between the two groups, so we do not discuss them below.[3]

We found six instances of unsuccessful validation, all from the No-LP group. As described in Sec. 4, we compared subjective and objective assessments of code correctness on the two Fixed-Prompt tasks, which resulted in four outcomes: (1) *Complete and Accurate*, where the participant submitted a correct solution within the task time limit, (2) *Complete and Inaccurate*, where the participant submitted an incorrect solution without recognizing the error, (3) *Timeout after Validation*, where the participant formed an accurate understanding of the correctness of the suggestions but reached the time limit before fixing the error in their chosen suggestion, and (4) *Timeout during Validation*, where the participant reached the time limit before they had finished validating the suggestions. We consider (1) and (3) to be instances of *successful validation*, (2) to be an instance of *over-reliance* on the AI suggestions, and (3) to be an instance of *under-reliance*, as the participant did not

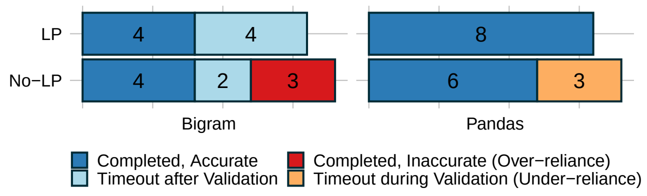


Figure 2: Success in validating AI suggestions across groups for Fixed-Prompt tasks. "Completed" means the participant submitted a solution they were satisfied with by the time limit, and "Timeout" means they did not. We deem the validation *successful* if a participant submitted a correct solution (dark blue) or timed out when attempting to fix the correctly identified bugs in their chosen suggestion (light blue).

successfully validate the suggestions in the given time. As Fig. 2 shows, we found three instances of over-reliance in the Bigram task and three instances of under-reliance in the Pandas task, *all from the No-LP group*, though the overall between-group difference was not significant ($p = .206$ for both tasks).

Participants with over-reliance did not inspect enough runtime behavior. The three No-LP participants with over-reliance in Bigram (P5, P12, P15) made a similar mistake: they accepted one of the mostly-correct suggestions (similar to Suggestion 3 in Sec. 3) and failed to notice that ties were not resolved alphabetically. Among the three participants, P5 did not run their code at all. P12 and P15 both tested *only one* suggestion on the given input and failed to notice the presence of two bigrams of the same count (and the fact that other suggestions returned different results). In addition, P15 cited *"reading the comments on what it was doing"* as a key factor for choosing the suggestion they did. That suggestion began with a comment stating that it resolved ties alphabetically, but the following code did not do so.

Participants with under-reliance lacked affordances for inspecting runtime behavior. The three No-LP participants who under-relied on AI suggestions (P7, P9, P15) tried to use runtime values for validation but struggled with doing so. P9 previewed and ran multiple suggestions but did not add any `print` statements to the code, and so they could only see the output of one of the suggestions, which ended in a `print` statement. P15 ran all suggestions and did add a `print` statement to each to inspect the final return value, but the need to change the `print` statement and re-run each time made this process difficult, and they lost track of which suggestions they considered the most promising, saying *"I forgot which ones looked decent."* Finally, P7's strategy was to print the output of subexpressions from various suggestions in order to understand their behavior and combine them into a single solution, but this was time-consuming, so they did not finish.

5.2 RQ2: Validation Strategies

Our participants had access to two validation strategies: *examination* (reading the code) and *execution* (inspecting runtime values). The general pattern we observed was that participants first did some amount of examination inside the Suggestion Panel—ranging from a quick glance to thorough reading—and then proceeded to

[2]https://bit.ly/leap-study-materials
[3]In median times, the LP group completed the Pandas task faster by 35 seconds ($p = .664, U = 31$). For Bigram, LP participants were slower by 3 minutes and 51 seconds ($p = .583, U = 42$), though this difference changes to *faster* by 10 seconds if we exclude those who solved the task incorrectly. For Pandas, both groups had the median ratings of confidence in correctness as "Agree" on seen inputs ($p = .784, U = 30$) and "Neutral" on unseen inputs ($p = .795, U = 33$). For Bigram, the LP group had the median rating of confidence in correctness on seen inputs as "Agree", while the No-LP group had "Strongly Agree" ($p = .097, U = 19.5$). As for confidence in correctness on unseen inputs, the median for the LP group was "Neutral", and that for the No-LP group was "Agree" ($p = .201, U = 22.5$).

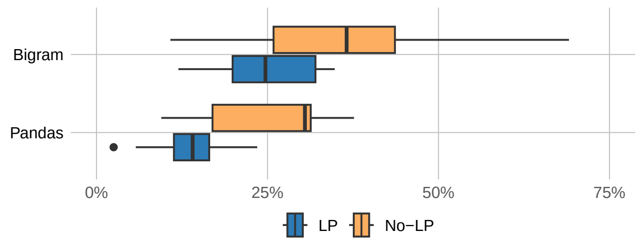


Figure 3: Percentage of time spent in the Suggestion Panel across the two groups for Fixed-Prompt tasks.

preview zero or more suggestions, performing further validation by execution inside the editor. To this end, No-LP participants in most tasks ran the code and added `print` statements for both final and intermediate values; LP participants in all tasks inspected both final and intermediate runtime values in Projection Boxes (by moving the cursor from line to line to bring different boxes into focus), and occasionally added `print` statements to see variables not shown by default. Below we discuss notable examples of validation behavior, as well as differences between the two groups and across tasks.

LP participants spent less time reading the code. We use the time the Suggestion Panel was in focus as a proxy for examination time; Fig. 3 shows this time as a percentage of the total task duration. The No-LP group spent more time in the Suggestion Panel compared to LP for both Fixed-Prompt tasks. The difference is significant in the Pandas task ($p = .02, U = 11$, $median_{LP} = 14.05\%$, $median_{No-LP} = 30.47\%$) but not in Bigram ($p = .14, U = 20$, $median_{LP} = 24.70\%$, $median_{No-LP} = 36.57\%$). We also collected this data for the Open-Prompt tasks, although it should be interpreted with caution due to the unstructured nature of the tasks (*e.g.*, participant engagement with the assistant and suggestion quality varied). The results are consistent with the Fixed-Prompt tasks—*i.e.*, No-LP participants spent more time in the Suggestion Panel—but the difference is not significant, and the effect in Box Plot is very small ($p = .14, U = 3.5$, $median_{LP} = 6.25\%$, $median_{No-LP} = 15.49\%$ for String Rewriting; $p = .67, U = 6$, $median_{LP} = 8.10\%$, $median_{No-LP} = 8.70\%$ for Box Plot).

Participants relied on runtime values more in API-heavy, one-off tasks. According to Fig. 3, both groups spent more time examining the code in Bigram, while in Pandas they jumped to execution more immediately ($median_{\text{Pandas}} = 16.96\%$, $median_{\text{Bigram}} = 31.67\%, p = .04, U = 206$). This difference in validation strategies between the two tasks was also reflected in the interviews. For example, P1 described their strategy for Pandas as follows: *"I didn't look too closely in the actual code, I was just looking at the runtime values on the side."* Instead, in Bigram, participants cared more about the code itself, preferring suggestions based on their expected algorithm, data structure, or style (*e.g.* P15 *"was really looking for the dictionary aspect"*), with the most popular attribute being "short"/"readable", cited by 10 out of 17 participants. One explanation participants gave for the difference in behavior is that Pandas is an API-heavy task, and when dealing with unfamiliar APIs, reading the code is just not very helpful: *"When it's using more jargony stuff that doesn't translate directly into words in your brain, then seeing the preview makes it clearer"* (P3). Another explanation they gave is that Pandas was perceived by the participants as a *one-off* task, *i.e.*, it only needed to work on the one specified input, whereas Bigram was perceived as *general*, *i.e.*, it needed to work on *"any sort of string [. . .] not only [. . .] the specific string that was tested"* (P3); this was not explicit in the instructions, but in retrospect it is a reasonable assumption, given the problem domains and structure of the starter code. On the other hand, some LP participants conjectured that with more familiarity with Live Programming, they would rely on runtime values more, even in tasks like Bigram: *"If I were to use this tool again I would preview more immediately, just because I think I was very focused on whether it produced how I would solve the problem vs. whether it solved the problem correctly"* (P4).

LP participants benefitted from visualizing intermediate values. We looked into the validation strategies used in Bigram to identify the tie-resolution issue in AI suggestions (excluding P17 because they wrote the code from scratch). In the input we provided, it was hard to identify the most common bigram at a glance, which made it difficult to validate suggestions just by looking at the final result. *Five out of eight* LP participants found the issue by inspecting *intermediate values* and noticing that multiple bigrams in the input have the same count (the other three relied on custom test cases and code examination). In the No-LP group, three out of eight participants failed to identify the issue and of the remaining five who succeeded, *only one* (P6) relied on intermediate values to do so. In addition, multiple LP participants (P1, P3, P4) mentioned the usefulness of intermediate values in the interview, especially for long suggestions. P1 said: *"Because it's a block of text as a suggestion, having projection boxes is more important [. . .] my thought was 'let me go line by line to see what is going on'."* In contrast, a No-LP participant (P9) remarked that they *"had to really look through the code and try to visualize it in [their] mind."*

LP participants used liveness features for validation and debugging. For validation, LP participants made use of full liveness, *i.e.*, the ability to see the immediate effects of their edits. *Five* participants in Pandas added auxiliary calculations to double-check the correctness of the final output, *e.g.*, the mean of specific cells in the input table, comparing it to the output table. When it comes to debugging, LP participants made multiple rounds of trial and error guided by liveness. In fact, the example in Sec. 3 was inspired by P4's debugging process in the Bigram task. Also, in Box Plot, P1 made many repeated edits in an AI suggestion to tune the placement of a label, guided by error messages and incorrect outputs to figure out the precise usage of an unfamiliar API call. In the interview, they noted: *"I was definitely using the projections [...] as I was editing the suggestions to see if my intended changes actually were followed through."*

5.3 RQ3: Cognitive Load in Validation

LP participants experienced significantly lower cognitive load in the Pandas task but not the Bigram task. In Pandas, LP participants experienced significantly lower cognitive load in four out of five aspects of NASA-TLX [14]: mental demand ($p = .039, U = 14.5$), performance ($p = .048, U = 15.5$), effort ($p = .015, U = 11$), and frustration ($p = .0004, U = 0$). We find no significant differences in responses to Bigram, but LP participants reported slightly higher

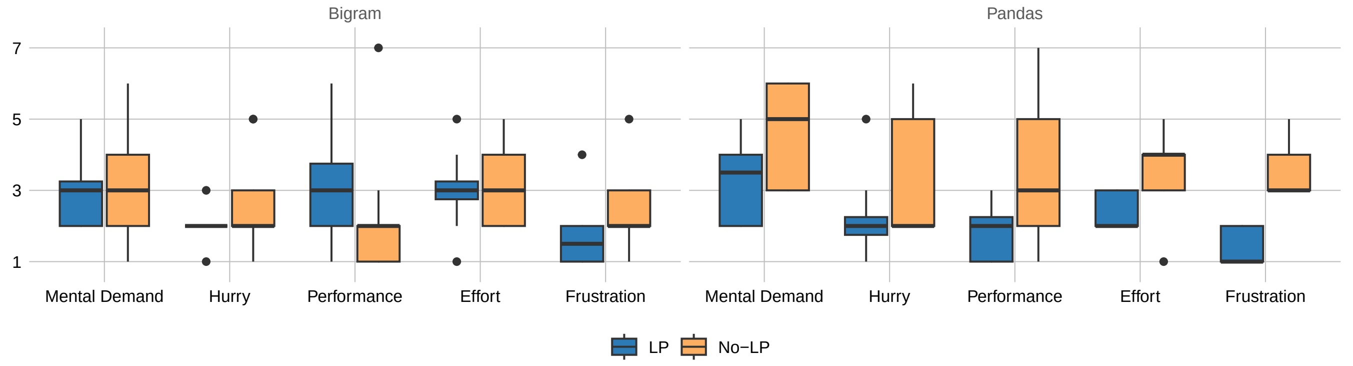


Figure 4: NASA Task Load Index (TLX) results for the Fixed-Prompt tasks: Bigram on the left, and Pandas on the right. Higher scores indicate higher cognitive load (in case of Performance this means higher failure rate).

performance measures ($median_{LP}$ = 3, $median_{No-LP}$ = 2), which stand for higher failure rates.

Participants found LP helpful in distinguishing between multiple suggestions. Participants from both the No-LP (P9, P14, P17) and LP (P3, P16) groups commented on the utility of seeing multiple suggestions at once: *"[Seeing multiple suggestions] gave me different ways to look at the code and gave me different ideas"* (P9) and *"multiple suggestions gave points of comparison that were useful"* (P14). However, some No-LP participants (P6, P7, P15, P17) said they found the suggestions hard to distinguish. They noted the difficulty of differentiating just by reading the code because *"the suggestions [were] all almost the same thing"* (P7), and observed that *"the tool did not really help with choosing between suggestions"* (P15). In comparison, some in the LP group (P1, P16) specifically commented that Live Programming was helpful in distinguishing and choosing between multiple code suggestions; P1 said: *"Being able to preview, edit, and look at the projection boxes before accepting a snippet was very helpful when choosing between multiple suggestions."* As far as we are aware, this is a new application of Live Programming, specific to AI programming assistants and not previously explored in Live Programming literature.

6 DISCUSSION

Live Programming lowers the cost of validation by execution. Although both LP and No-LP participants had access to runtime values as a validation mechanism, those without LP needed to examine the code to decide which values to print, add the `print` statements, run the code, and match each line in the output to the corresponding line in the code. If they wanted to inspect a different suggestion, they had to repeat this process from the start. Meanwhile, LP participants could simply click on the suggestion to preview it and get immediate access to all the relevant runtime information, easily switching between suggestions as necessary. In other words, LP lowers *the cost*—in terms of both time and mental effort—of access to runtime values. As a result, we saw LP participants relied on runtime values more for validation, as they spent less time examining the code in general—and significantly so for the Pandas task—and more often used intermediate values to find bugs in Bigram (Sec. 5.2). Our findings are consistent with prior work [2, 21], which demonstrated that programmers more often use validation strategies with lower time costs. Hence, *by lowering the cost of access to runtime values, Live Programming promotes validation by execution.*

The lower cost of validation by execution prevents over- and under-reliance. As discussed in Sec. 5.1, we found six instances of unsuccessful validation in our study, *all from the No-LP group*, over-relying on AI suggestions in the Bigram task, and under-relying in Pandas. We attribute these failures to the high cost of validation by execution: those who over-relied did not inspect the runtime behavior of the suggestions in enough detail, while those with under-reliance lacked the affordances to do so effectively, and so ran out of time before they could validate the suggestions. Our results echo prior findings [34] that relate the cost of a validation strategy to its effectiveness in reducing over-reliance on AI. Prior work has also shown [11, 36] that programmers often struggle to form an appropriate level of trust in code synthesizers, whether AI-based or not; our results suggest an important new role for Live Programming in addressing this challenge. We conclude that *the lower cost of validation by execution in Live Programming leads to more accurate judgments of the correctness of AI-generated code.*

Validation strategies depend on the task. Sec. 5.2 shows that participants overall spent significantly more time examining the code in Bigram than in Pandas and also paid more attention to code attributes in the former. Participants explained the difference in their validation strategies by two factors: (1) Pandas contained unfamiliar API calls, the meaning of which they could not infer from the code alone; and (2) they perceived Pandas as a one-off task, which only had to work on the given input. We conjecture that (1) is partly due to our participants being LP novices: as they get more used to the environment, they are likely to rely on previews more, even if they are not forced into it by an unfamiliar API (as P4 mentioned in Sec. 5.2). (2), though, is more fundamental: when dealing with a general task, correctness is not all that matters; code quality becomes important as well, and LP does not help with that.

In Open-Prompt tasks, code examination was less prevalent in the overall task duration, because in these tasks participants spent a significant amount of time on activities besides validation (*e.g.*, decomposing the problem and crafting prompts). It might seem surprising, however, that we did not see any difference in examination time between the two groups in Box Plot, which is an API-heavy, one-off task, similar to Pandas. This might be because, in Box Plot, the cost of validation by execution was already low for No-LP participants: this task did not require inspecting intermediate values, because the effects of each line of code were reflected on the final plot in a compositional manner (*i.e.*, it was easy to tell what each line of code was doing just by looking at the final plot).

In conclusion, *Live Programming does not completely eliminate the need for code examination but reduces it in tasks amenable to validation by execution.*

Live Programming lowers the cognitive load of validation by execution. In Pandas, LP participants experienced lower cognitive load in four out of five TLX categories (Sec. 5.3). This confirms our hypotheses that LP lowers the cost of validation by execution, and that Pandas is a task amenable to such validation. More specifically, we conjecture that, by automating away the process of writing `print` statements, LP reduces workflow interruptions, which were identified as one of the sources of increased cognitive load in reviewing AI-generated code [36].

In Bigram, however, we did not observe a similar reduction in cognitive load; in fact, LP participants reported *higher* cognitive load in the "performance" category (*i.e.*, they perceived themselves as less successful). Our interpretation is that the cognitive load in this task was dominated by debugging and not validation, and whereas all participants in the LP group engaged in debugging, only two-thirds of the No-LP group did so. Finally, the higher "performance" ratings from the LP group were from those who ran out of time trying to fix the code, and hence were aware that they had failed. These findings show that Live Programming by itself does not necessarily help with debugging a faulty suggestion. As we saw in Sec. 5.2, it can be helpful when the user has a set of potential fixes in mind, which they can quickly try out and get immediate feedback on. But when the user does not have potential fixes in mind, they need to rely on other tools, such as searching the web or using chat-based AI assistants.

From these findings, we conclude that *Live Programming lowers the cognitive load of validating AI suggestions when the task is amenable to validation by execution.*

7 CONCLUSION AND FUTURE WORK

We investigated an application of Live Programming in the domain of AI-assisted programming, finding that LP can reduce over- and under-reliance on AI-generated code by lowering the cost of validation by execution. Our work highlights new benefits of LP specific to AI-assisted programming, such as building appropriate trust in the assistant and helping to choose between multiple suggestions. Our study is necessarily limited in scope: we focused on self-contained tasks due to LP's limited support for complex programs [20, 31] and its need for small demonstrative inputs [28]. We hope that our findings inform future studies on code validation and motivate further research into AI-LP integration. To that end, we highlight key opportunities below.

To offer liveness, LP places several burdens on the user. The user must provide a complete executable program and a set of test cases, and then look through potentially large runtime traces for the relevant information. AI may alleviate these burdens by filling in missing runtime values [29] for incomplete programs, generating test cases [19, 38], and predicting the most relevant information to be displayed at each program point. Looking beyond the validation of newly generated code, there are also opportunities for AI-LP integration for debugging and code repair [38]. In combination, AI-LP would tighten the feedback loop of querying and repairing AI-generated code: users could validate code via LP, request repair using the runtime information from LP [11], and further validate the repair in LP.

ACKNOWLEDGMENTS

We would like to thank Elena Glassman for her invaluable feedback on a draft of this work, and the anonymous reviewers for their helpful comments and suggestions. This work was supported in part by NSF grant 2107397.

REFERENCES

[1] Amazon. 2023. CodeWhisperer. https://aws.amazon.com/codewhisperer/.

[2] 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 (apr 2023), 27 pages. https://doi.org/10.1145/3586030

[3] Christian Bird, Denae Ford, Thomas Zimmermann, Nicole Forsgren, Eirini Kalliamvakou, Travis Lowdermilk, and Idan Gazit. 2022. Taking Flight with Copilot: Early Insights and Opportunities of AI-Powered Pair-Programming Tools. *Queue* 20, 6 (Dec. 2022), 35–57. https://doi.org/10.1145/3582083

[4] Lautaro Cabrera, John H. Maloney, and David Weintrop. 2019. Programs in the Palm of your Hand: How Live Programming Shapes Children's Interactions with Physical Computing Devices. In *Proceedings of the 18th ACM International Conference on Interaction Design and Children.* ACM, Boise, ID, USA, 227–236. https://doi.org/10.1145/3311927.3323138

[5] Miguel Campusano, Alexandre Bergel, and Johan Fabry. 2016. Does Live Programming Help Program Comprehension? – A user study with Live Robot Programming. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools.* ACM, Amsterdam, Netherlands, 1–8. http://bergel.eu/MyPapers/Camp16-ComprehensionWithLRP.pdf

[6] Kathy Charmaz. 2014. *Constructing Grounded Theory.* sage.

[7] Robert DeLine and Danyel Fisher. 2015. Supporting Exploratory Data Analysis with Live Programming. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC).* IEEE, Atlanta, GA, 111–119. https://doi.org/10.1109/VLHCC.2015.7357205

[8] Robert A DeLine. 2021. Glinda: Supporting Data Science with Live Programming, GUIs and a Domain-specific Language. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems.* ACM, Yokohama, Japan, 1–11. https://doi.org/10.1145/3411764.3445267

[9] 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. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems.* ACM, Honolulu, HI, USA, 1–12. https://doi.org/10.1145/3313831.3376442

[10] Jimmy Efird. 2011. Blocked Randomization with Randomly Selected Block Sizes. *International Journal of Environmental Research and Public Health* 8, 1 (2011), 15–20. https://www.mdpi.com/1660-4601/8/1/15

[11] 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* (Virtual Event, USA). Association for Computing Machinery, New York, NY, USA, 614–626. https://doi.org/10.1145/3379337.3415869

[12] GitHub. 2023. GitHub Copilot - Your AI Pair Programmer. https://copilot.github.com/.

[13] Christopher Michael Hancock. 2003. *Real-Time Programming and the Big Ideas of Computational Literacy.* Ph. D. Dissertation. Massachusetts Institute of Technology. https://dspace.mit.edu/handle/1721.1/61549

[14] Sandra G Hart and Lowell E Staveland. 1988. Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research. In *Advances in*

Psychology. Vol. 52. Elsevier, 139–183. https://doi.org/10.1016/S0166-4115(08)62386-9

[15] Ruanqianqian (Lisa) Huang, Kasra Ferdowsi, Ana Selvaraj, Adalbert Gerald Soosai Raj, and Sorin Lerner. 2022. Investigating the Impact of Using a Live Programming Environment in a CS1 Course. In *Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2022).* Association for Computing Machinery, New York, NY, USA, 495–501. https://doi.org/10.1145/3478431.3499305

[16] 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 (November 2020), 27 pages. https://doi.org/10.1145/3428273

[17] Hyeonsu Kang and Philip J. Guo. 2017. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology.* ACM, Québec City, QC, Canada, 737–745. https://doi.org/10.1145/3126594.3126632

[18] Jan-Peter Kramer, Joachim Kurz, Thorsten Karrer, and Jan Borchers. 2014. How Live Coding Affects Developers' Coding Behavior. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC).* 5–8. https://doi.org/10.1109/VLHCC.2014.6883013 ISSN: 1943-6106.

[19] Shuvendu K. Lahiri, Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, Madanlal Musuvathi, Piali Choudhury, Curtis von Veh, Jeevana Priya Inala, Chenglong Wang, and Jianfeng Gao. 2023. Interactive Code Generation via Test-Driven User-Intent Formalization. https://doi.org/10.48550/arXiv.2208.05950 arXiv:2208.05950 [cs]

[20] 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* (Honolulu, HI, USA). Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3313831.3376494

[21] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. 2023. Understanding the Usability of AI Programming Assistants. arXiv:2303.17125 [cs.SE] https://doi.org/10.48550/arXiv.2303.17125

[22] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2022. Reading Between the Lines: Modeling User Behavior and Costs in AI-Assisted Programming. arXiv:2210.14306 [cs.SE] https://doi.org/10.48550/arXiv.2210.14306

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

[24] OpenAI. 2023. GPT-3.5. https://platform.openai.com/docs/models/gpt-3-5.

[25] 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 (November 2020), 30 pages. https://doi.org/10.1145/3428227

[26] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2022. Do Users Write More Insecure Code with AI Assistants? arXiv:2211.03622 [cs.CR] https://doi.org/10.48550/arXiv.2211.03622

[27] Steven I. Ross, Fernando Martinez, Stephanie Houde, Michael Muller, and Justin D. Weisz. 2023. The Programmer's Assistant: Conversational Interaction with a Large Language Model for Software Development. https://doi.org/10.1145/3581641.3584037 arXiv:2302.07080 [cs].

[28] Gustavo Soares, Emerson Murphy-Hill, and Rohit Gheyi. 2013. Live Feedback on Behavioral Changes. In *2013 1st International Workshop on Live Programming (LIVE).* 23–26. https://doi.org/10.1109/LIVE.2013.6617344

[29] Beatriz Souza and Michael Pradel. 2023. LExecutor: Learning-Guided Execution. https://doi.org/10.48550/arXiv.2302.02343 arXiv:2302.02343 [cs]

[30] Ningzhi Tang, Meng Chen, Zheng Ning, Aakash Bansal, Yu Huang, Collin McMillan, and Toby Jia-Jun Li. 2023. An Empirical Study of Developer Behaviors for Validating and Repairing AI-Generated Code. (March 2023). https://doi.org/10.1184/R1/22223533.v1

[31] Steven L Tanimoto. 2013. A Perspective on the Evolution of Live Programming. In *2013 1st International Workshop on Live Programming (LIVE).* IEEE, 31–34. https://doi.org/10.1109/LIVE.2013.6617346

[32] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems* (New Orleans, LA, USA). Association for Computing Machinery, New York, NY, USA, Article 332, 7 pages. https://doi.org/10.1145/3491101.3519665

[33] Helena Vasconcelos, Gagan Bansal, Adam Fourney, Q Vera Liao, and Jennifer Wortman Vaughan. 2022. Generation Probabilities Are Not Enough: Improving Error Highlighting for AI Code Suggestions. In *Virtual Workshop on Human-Centered AI Workshop at NeurIPS (HCAI NeurIPS'22). Virtual Event, USA.* 1–4. https://www.microsoft.com/en-us/research/uploads/prod/2022/10/Helena_s_Project.pdf

[34] Helena Vasconcelos, Matthew Jörke, Madeleine Grunde-McLaughlin, Tobias Gerstenberg, Michael Bernstein, and Ranjay Krishna. 2023. Explanations Can Reduce Overreliance on AI Systems During Decision-Making. http://arxiv.org/abs/2212.06823 arXiv:2212.06823 [cs].

[35] Bret Victor. 2012. Learnable Programming. http://worrydream.com/LearnableProgramming/

[36] Ruotong Wang, Ruijia Cheng, Denae Ford, and Thomas Zimmermann. 2023. Investigating and Designing for Trust in AI-powered Code Generation Tools. *arXiv preprint arXiv:2305.11248* (2023). https://doi.org/10.48550/arXiv.2305.11248

[37] 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

[38] Mark Wilson-Thomas. 2023. Simplified Code Refinement and Debugging with GitHub Copilot Chat. https://devblogs.microsoft.com/visualstudio/simplified-code-refinement-and-debugging-with-github-copilot-chat/

[39] Dvora Yanow. 2017. Qualitative-Interpretive Methods in Policy Research. In *Handbook of Public Policy Analysis.* Routledge, 431–442.

[40] 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

[41] Chunqi Zhao, I-Chao Shen, Tsukasa Fukusato, Jun Kato, and Takeo Igarashi. 2022. ODEN: Live Programming for Neural Network Architecture Editing. In *27th International Conference on Intelligent User Interfaces.* Association for Computing Machinery, New York, NY, USA, 392–404. https://doi.org/10.1145/3490099.3511120