Ivie: Lightweight Anchored Explanations of Just-Generated Code

Litao Yan
ltyan@seas.upenn.edu
University of Pennsylvania
Philadelphia, PA, USA

Alyssa Hwang
ahwang16@seas.upenn.edu
University of Pennsylvania
Philadelphia, PA, USA

Zhiyuan Wu
wuzed@seas.upenn.edu
University of Pennsylvania
Philadelphia, PA, USA

Andrew Head
head@seas.upenn.edu
University of Pennsylvania
Philadelphia, PA, USA

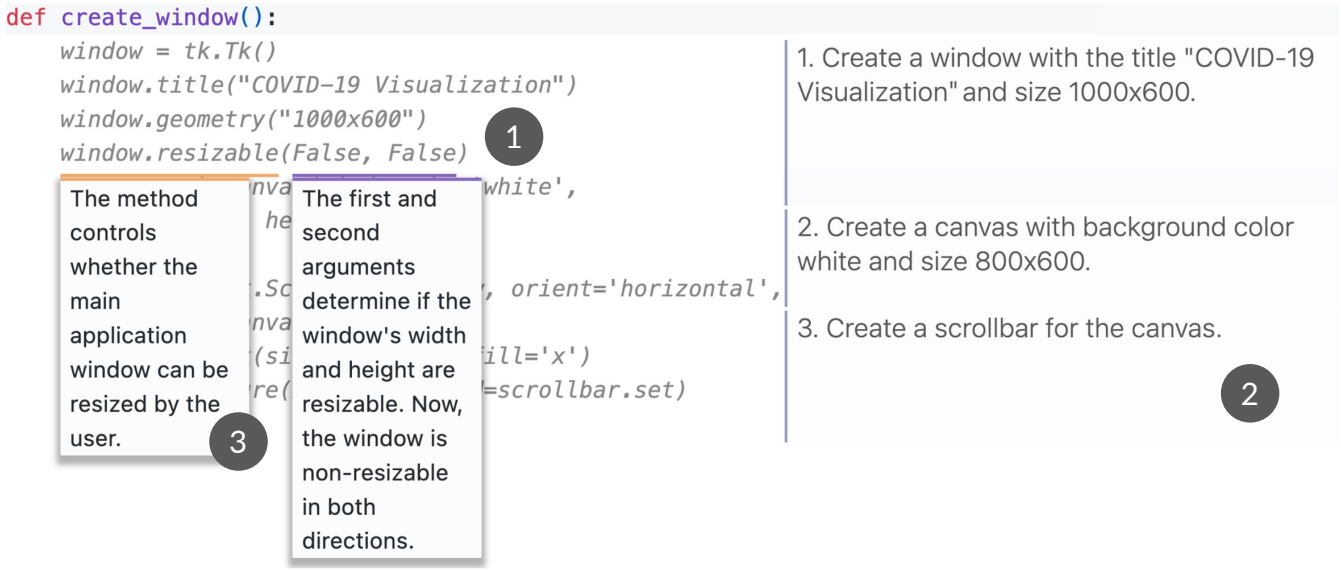

Figure 1: Ivie augments the interactive programming assistant with instant explanations that help programmers examine generated code. When a programming assistant suggests code (*italic* text above, ❶), Ivie annotates it with brief, informative explanations. Explanations appear at the level of blocks of code (in the right margin, ❷) and expressions (anchored beneath the line the programmer hovers over, ❸). For single-line suggestions, expression explanations appear automatically. Ivie's explanations help programmers break up complex or unfamiliar suggestions into pieces that can be more readily understood.

ABSTRACT

Programming assistants have reshaped the experience of programming into one where programmers spend less time writing and more time critically examining code. In this paper, we explore how programming assistants can be extended to accelerate the inspection of generated code. We introduce an extension to the programming assistant called Ivie, or *instantly visible in-situ explanations*. When using Ivie, a programmer's generated code is instantly accompanied by explanations positioned just adjacent to the code. Our design was optimized for low-cost invocation and dismissal. Explanations are compact and informative. They describe meaningful expressions, from individual variables to entire blocks of code. We present an implementation of Ivie that forks VS Code, applying a modern LLM for timely segmentation and explanation of generated code. In a lab study, we compared Ivie to a contemporary baseline tool for code understanding. Ivie improved understanding of generated code, and was received by programmers as a highly useful, low distraction complement to the programming assistant.

CCS CONCEPTS

• **Human-centered computing** → **Interactive systems and tools**.

KEYWORDS

Programming assistants, instructive copilots, anchored explanations, comprehension support, variable levels of detail, brevity, easy invocation, easy dismissal, label overlays

ACM Reference Format:
Litao Yan, Alyssa Hwang, Zhiyuan Wu, and Andrew Head. 2024. Ivie: Lightweight Anchored Explanations of Just-Generated Code. 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, 15 pages. https://doi.org/10.1145/3613904.3642239

CHI '24, May 11–16, 2024, Honolulu, HI, USA

ACM ISBN 979-8-4007-0330-0/24/05.
https://doi.org/10.1145/3613904.3642239

1 INTRODUCTION

Since their recent release, programming assistants have begun to reshape the process of writing code. Programming assistants are

a kind of interactive programming aid that help a programmer write code. The predominant interaction model is exemplified by GitHub Copilot [22]. Under this model, the programming assistant anticipates code that the programmer is going to write—either the remainder of a line, or even many lines at once—and proposes that code to the programmer. A programmer can then accept the code with a simple command, like pressing the "Tab" key. Production programming assistants have become very good at proposing code, even long chunks of code, should the programmer's code resemble something seen in the massive corpora of code that the programming assistant was trained on.

This interaction model has proven to be both desirable and useful. Programming assistants have seen massive adoption. Copilot estimates adoption by 1.2 million users [15]. They have seen integration into some of the most commonly-used code editors such as VS Code. And they are used to generate a lot of code: recent estimates suggest that as much as 30% [34] or 40% [15] of a programmer's code originates from programming assistants when in active use. The widespread and frequent usage of programming assistants suggests that they may be here to stay.

For the effort that they save, programming assistants introduce another kind of effort. Namely, programming assistants eliminate effort writing code and replace it with effort reviewing code [4]. Studies have reported programmers spending an undesired amount of time understanding and debugging generated code [3, 61]; they spend time examining generated code [3], often examining details of generated code's logic in depth [34]. Obstacles to understanding generated code include its use of previously-unknown APIs or methods, structural complexity [34], and the length of generated code [3, 34]. For particularly long generations, programmers have described themselves as "distracted by everything [the assistant] is throwing at [them]," "lost in the sauce," and "discombobulated" [3]. Perhaps troublingly, programmers have reported understanding less of how and why generated code works [4], with one lab study suggesting that programmers apportion less attention to generated code than that written by human authors [1].

This raises the question of whether our modern notion of the programming assistant is incomplete. A widely-used metaphor for programming assistants is that they are copilots. If a programming assistant is a copilot, it is one that does its work without explaining its actions. Its actions are visible, but the copilot makes no effort to make them understandable. This is desirable when the copilot's actions are familiar, like when it generates code with known idioms. However, it is less desirable when the copilot's actions are unfamiliar, like when it generates code with unfamiliar APIs and structures. If programmers are expending effort to understand generated code, perhaps the notion of the copilot needs to be expanded to one of an *instructive copilot* that prioritizes the programmer's understanding by explaining its work.

In this paper, we explore this notion of the instructive copilot. We develop a tool, Ivie, which provides lightweight, anchored, AI-generated explanations of just-generated code. When using an editor with Ivie, programmers see generated code instantly accompanied by brief labels that explain what the code does (Figure 1). These labels are meant to answer questions like "what does this parameter do?" or "what does this segment of generated code do?" That is, they are tailored to helping programmers understand unfamiliar APIs and idioms in the generated code. We call the labels *instantly visible in-situ explanations* (i.e., Ivie). The labels are designed for tight integration into the modern workflow of generating, reviewing, and accepting code; they are meant to eliminate extraneous time it takes to provide answers to basic questions about the code prior to the programmer accepting or rejecting the code. The labels are very easy to invoke and dismiss. They explain code at the level of both blocks (for long, multi-line generations) and expressions (for dense one-liners). By providing instant, brief, anchored explanations, Ivie seeks to remove information costs intrinsic to comparable forms of support for understanding generated code like chatbot-based programming helpers (e.g., [20, 21, 38]) or code explainers that require code selection (e.g., [17, 23]).

Ivie has a straightforward architecture, leveraging an off-the-shelf LLM to segment code into explainable units (at both the expression and block level), and create brief explanations of those units. Most of Ivie's code consists of presentation logic that overlays LLM outputs as anchored explanations in the editor. We demonstrate the viability of bringing Ivie to production editors by implementing it in the widely-used Visual Studio Code [62].

We present a detailed in-lab usability study of Ivie. The study explores Ivie's impact on the experience of understanding generated code containing unfamiliar APIs. The study compared Ivie to a contemporary AI-based help-seeking baseline, namely a GPT-based in-editor chatbot that can answer questions about selected code. Our study shows that there is a clear place in the ecosystem of programming tools for Ivie: it was preferred to the chatbot by nearly all participants. Ivie led to improved comprehension of generated code, while decreasing perceived task load. Participants described Ivie as being highly useful without being distracting. They characterized Ivie as a complementary addition to a modern programming workspace alongside documentation and chatbot help.

In summary, this paper contributes:

- The notion of an *instructive copilot* as a programming assistant that not only generates code, but also supports its understanding with timely, anchored, brief explanations.
- The implementation of this idea in Ivie, an extension to the programming assistant that augments generated code with overlay descriptions that are brief, informative, instantly visible, and easy to dismiss.
- A usability study that confirms Ivie's value as an desirable complement to a programming assistant that improves code comprehension with low task load and distraction.

2 BACKGROUND AND RELATED WORK

2.1 User experience of programming assistants

In recent years, a number of tools have incorporated large language models (LLMs) to provide production-level assistance for generating and describing code. These tools, often called "programming assistants," include GitHub Copilot [22], OpenAI GPT-4 [40], Amazon CodeWhisperer [2], IntelliCode Compose [56], and CodeT5+ [64]. As programming assistants have been adopted by a growing number of programmers, researchers have begun to examine their effect on programming activity. Some of these studies have characterized

the tools' influence on how programmers understand code, particularly generated code [1, 3, 34, 36, 61, 67]. We detailed takeaways from these studies in detail in Section 1; in brief, programmers report understanding less of the generated code than they do of the code they write, spending time examining the code, and sometimes finding the generated code difficult to read. These observations motivate our focus on developing aids to support the reading of generated code.

A recent study by Barke et al. [3] characterized interaction with programming assistants as consisting of two modes: acceleration, where the assistant helps a programmer write code of the kind they already have some idea how to write; and exploration, where the programming assistant assists a programmer in determining and carrying out goals. We posit that a tool like Ivie could be being useful in supporting both modes of interaction. As our results suggest in Section 6.1, explanations of the kind Ivie provides can be helpful in learning about the behavior of unfamiliar code that might be generated during exploration, as well as in providing a timely comprehension aid for longer and more complex code that might appear in the middle of acceleration.

2.2 Program comprehension

Program comprehension—or the process of understanding computer code—is an essential and frequent programming activity. In one study, program comprehension accounted for as much as 50% of programmers' time [66]. Program comprehension has been researched in depth by the HCI and software engineering research communities (see for instance reviews by Détienne [14] and Crichton [13]). In some theories of program comprehension (e.g., [14]), comprehension relies on the identification of schemas—or meaningful structures—in the code, and synthesis of those schemas into a mental model of the program. We see the role of Ivie as recognizing unfamiliar schemas in code on a programmer's behalf, and explaining them in approachable terms.

Program comprehension is sometimes considered as taking place either top-down or bottom-up. When following a top-down approach, programmers form hypotheses about a program's intent, and then verify those hypotheses by looking for recognizable features or "beacons" in the code [6, 55]. When following a bottom-up approach, programmers progressively group units of the code into larger and larger abstractions [43, 52, 63]. We consider Ivie as primarily supporting a bottom-up approach to comprehension with its explanations of expressions; at the same time, its block-level explanations provide some support for top-down comprehension, albeit at the level of dozens of lines of code.

Numerous prior studies have sought to characterize the kinds of questions that arise as programmers read and write code [16, 32, 47, 54]. Duala-Ekoko and Robillard [16], for instance, identified 20 difficult questions that programmers faced when working with unfamiliar APIs. Many of these are answerable using Ivie. For instance, we have seen Ivie answer questions like "what roles do the arguments of a given method play in its usage," "what is the valid range of values for a primitive argument or a given method," and "how do I determine the outcome of a method call?" from Duala-Ekoko and Robillard [16]'s study, or "what is the 'correct' way to use or access this data structure" from Sillito et al. [54]'s study. Ivie is tuned to answer questions about the high-level behavior of passages of code a couple dozen lines in length, as well as the role of individual expressions in the behavior of a statement.

There exist many methods for evaluating program comprehension, including think-aloud protocols [49, 63], memorization [43, 51, 55], comprehension tasks [55], fMRI readings [53], near-infrared spectroscopy [37], EEG [19], and eye tracking [1, 31, 50, 57]. Tang et al. [57] recently observed programmers' gaze when validating and fixing generated code; their study was able to reveal periods of careful attention to generated code. Our own study measures comprehension primarily through a set of questions about generated code, and coarse gaze-based measures of attention.

2.3 Interactive program comprehension aids

The HCI literature is replete with interactive systems research focused on helping people understand programs. Recently-developed research systems have supported the understanding of programs in many senses. For instance, they have supported sensemaking about code [30] and APIs [29] with novel annotation affordances, understanding of code examples with concept-annotated code snippets [69] and collated views of API usages [24, 68], and understanding how existing web pages are implemented with new methods to inspect their underlying source [7, 10, 26].

Some of this research has, like Ivie, focused on supporting in-situ understanding of code—that is, augmenting the editor in a way that helps a programmer understand the code that is within their focus. Hoffswell et al. [27] introduced a grammar of in-editor visualizations that can be used to understand the values of myriad data types with a small footprint. The projection boxes project [33] provides a framework for live programming with Python where values of expressions can be viewed adjacent to the lines on which they appear. LEAP [18] applies this idea to support the comprehension of AI-generated code. Tutorons [25] explores a similar notion to this paper, generating brief "microexplanations" for individual lines of code that appear in tutorials to provide on-demand comprehension assistance. A design space of related forms of on-demand help was explored by Potter et al. [44] in their design of ExplainThis. Ivie is inspired by this prior work, exploring how some of the above affordances could be brought into the editor to support expression- and block-level code understanding in an era where LLMs are powerful enough to segment and explain code on demand.

A growing number of tools have brought LLMs into the code editor to support code understanding. Notable contemporary examples include CodeHelp [35], EasyCode [17], Genie AI [20], GitHub Copilot X [21]. Nam et al. [38] recently designed and evaluated a system of this kind, which supports explanation of selected code, details of API calls used in the code, explanations of domain-specific terms, and provision of usage examples for an API, with a study showing advantages over web browser-based help. Ivie explores an interaction model where similar kinds of support are provided, in a way that is tightly integrated with the programming assistant to instantly provide clarifying information to programmers without diverting their attention away from the code.

3 SYSTEM

The purpose of Ivie is to help programmers acquire an understanding of whether generated code matches their intentions, and how

they should modify it if it does not. It has a particular focus on helping programmers understand unfamiliar APIs and idioms in generated code. When combined with a programming assistant like Copilot, Ivie embodies the idea of an *instructive copilot*—that is, a copilot that it explains its work in a way that empowers a user to further refine its output. There is considerable nuance in developing a usable instructive copilot. A fundamental tension is showing information to programmers in a way that is simultaneously instant and unobtrusive. Below, we describe a set of design goals for instructive copilots that we believe address this tension. The goals are motivated by best practices in interaction design, and by choices that arose during Ivie's iterative development. The goals serve to crystallize how our vision departs from contemporary, chat-based approaches to programming help (Section 2.3). Specifically, we posit that instructive copilots should provide explanations that are:

D1. Anchored. Programmers should not have to divert their attention from generated code to get assistance in understanding it. Doing so could induce split attention [9] and undesired cognitive load. Instead, explanations should appear in-situ, next to the code.

D2. Lightweight. Explanations should support a basic understanding of the code at a glance. They should be simple enough that they impose only a small load on programmers' working memory. This is consistent with the principle of minimalism, wherein documentation is kept concise and focused on users' tasks [8].

Furthermore, guided by standard usability recommendations in favor of speeding up frequent actions and allowing flexible ordering of tasks [39], we recommend that explanations are:

D3. Easy to invoke. Programmers should not need to expend any effort to see explanations.

D4. Easy to dismiss. Explanations should be dismissed automatically when they are no longer needed.

D5. Accessible anytime. While explanations should be hidden when not needed, it should be easy for programmers to bring explanations back for any generated code when they need them.

We also posit that explanations should appear at multiple multiple levels of abstraction. As discussed in Section 2.2, programmers need to understand the behavior of code not only at the level of individual expressions, but also higher-level structures. Finally, explanations of neighboring expressions should appear in parallel, because the task of understanding a programming statement often requires making sense of the interrelated behaviors of its component expressions. In the upcoming sections, we describe how these goals are addressed in the design and implementation of Ivie.

3.1 Interface Design

Ivie is designed to deliver on the design goals with the following affordances. We direct readers to Figure 1, demo video, and the scenario (Section 4) to see how these affordances appear to users.

3.1.1 Expression-level explanations. When the programming assistant suggests a single-line suggestion, Ivie shows explanations of major expressions that make up that line (Figure 4). Major expressions are automatically identified using an LLM (see Section 3.2.2). Then these expressions are assigned brief descriptive labels. These expression-level explanations are designed to adhere to the design goals, with the following choices:

Anchored (D1). To reduce split attention that arises when accessing conventional forms of documentation, explanations appear *anchored alongside the code*, as close to the expressions they explain as possible. Labels are associated with expressions using proximity and shared color on the borders of the expression and label. They appear beneath the suggestion, as we anticipate the context above a line of code will be more important than the context below it when the programmer is choosing whether to accept the suggestion. In the case where many labels are generated, some of the labels are moved further away from their expressions, in which case leader lines are used to associate expressions with labels. The programmer can hover over a label to highlight just that label and underline the corresponding expression. Labels track the code even as the programmer scrolls, zooms, and resizes the editor.

Ivie does not currently support floating explanations that stand apart from the code. Floating explanations could be useful in situations where there is more to explain that what appears in the code—like other arguments to an API that were not generated.

Lightweight (D2). Explanations are *very short*—typically only 1–2 sentences long. Because the explanations are short, programmers have the opportunity to get the gist of an expression very quickly. A consequence of the labels' brevity is that they take up little visual space, allowing programmers to continue to refer to the surrounding code as they consider whether to accept a suggestion.

Easy to invoke (D3). Explanations are *(near) instantaneous*—they appear within seconds of the appearance of a suggestion (i.e., as quickly as our implementation retrieves them). Appearing automatically, they require no effort to invoke. They also *appear for all expressions in a line at once*. The parallel appearance of explanations can help a programmer visually segment the line into its meaningful parts, and synthesize an understanding of the line by referring to the interrelated meanings of its expressions.

Easy to dismiss (D4). In-situ explanations run the risk of being unwelcome should they distract a programmer. To reduce this risk, we made it easy to dismiss explanations. Explanations are *automatically dismissed* as soon as a suggestion is accepted, or as soon as the programmer clicks away from the suggestion.

3.1.2 Block-level explanations. When the programming assistant suggests a long block of code (two or more lines), Ivie helps a programmer understand the high-level steps taken by the code by showing explanations of blocks of code (i.e., contiguous sets of lines) in the right margin (see example in Section 4). The interaction design around block-level explanations is the same as expression-level explanations, with the following differences:

Appearance in the margins. To avoid the risk of occluding code, block-level explanations appear in the right margin. They appear just to the right of the right-most character in the suggested code, or if the suggestion is very long, right after the 80th character. Fading is used on the left side of the explanations to suggest that code is being hidden underneath them. The explanations are associated with

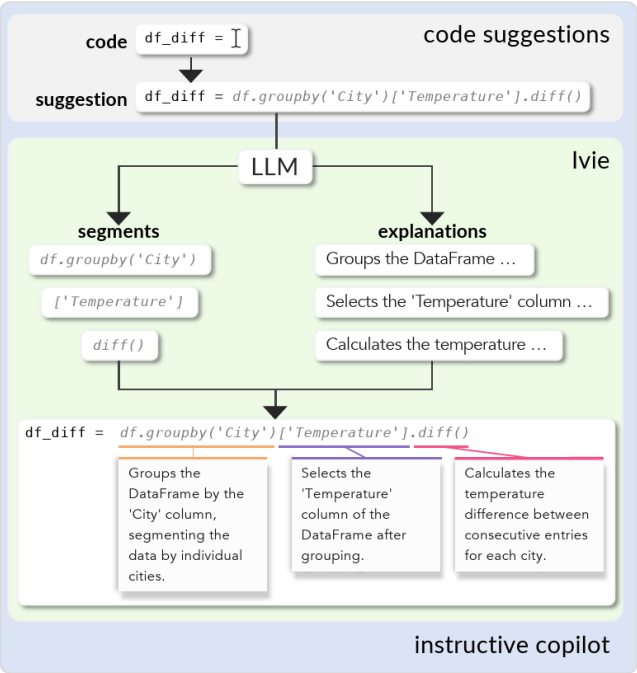

Figure 2: The implementation of an instructive copilot for programming. Ivie creates interactive overlays that explain suggestions made by a programming assistant. When the programming assistant (e.g., Copilot) displays the suggestion, Ivie submits that suggestion in a prompt to an LLM, requesting that the code be segmented and explained. Ivie then integrates the explanations into the editor as overlays beneath the expressions they explain.

lines of code with a single border on the left side of the explanation indicating the extent of code lines to which the explanation applies.

Expression-level drill-down. When using early prototypes of Ivie, we found that block-level explanations provided a useful entry point to understand suggestions, but sometimes they did not go into enough detail about what parts of the code did. Therefore, we extended block-level explanations to allow programmers to access details about the code on-demand, allowing them to hover over any line to see expression-level explanations for that line.

3.1.3 Making explanations accessible anywhere, anytime (D5). Ivie was designed to automatically dismiss explanations to reduce their imposition on programmers. A consequence of this choice is that programmers lose access to explanations when they accept code, when they may wish to continue seeing them. Ivie provides a button that allows programmers to see explanations after they have accepted suggestions. Once they press this button, explanations show not just for the generated code, but for the entire file: block-level explanations are shown in the margins, and the programmer can hover over any line in order to see expression-level explanations.

3.2 Implementation

A tool like Ivie can be implemented in a straightforward way, delegating much of the work to external tools and focusing mostly on

```
Please dissect the following line of code, and explain
the unfamiliar vocabulary and structures with less than
15 words each. Include ranges for parameter values and
describe how changes in these parameters will affect
the output.

Prompt:
fig, ax = plt.subplots(2, 1, figsize=(14, 10))
Output:
plt.subplots $#$ Create a figure and set of subplots.
2, 1 $#$ 2 rows, 1 column of subplots.
figsize=(14, 10) $#$ Width and height of entire figure.
fig $#$ The whole window/figure containing subplots.
ax $#$ Array of individual subplot axes.

Prompt:
df_all.merge(df_Apr, on='City', how='left',
suffixes=('_all', '_apr'))
Output:
```

Figure 3: A prompt for requesting expression-level explanations of generated code. This prompt requests explanations of suggested code. It provides a single example of how it would like code suggestions to be split into expressions with accompanying brief explanations of those expressions.

tailoring their output and tight editor integration. In this section we describe how Ivie works. A visual summary of Ivie's implementation appears in Figure 2. From suggestion to explanation, Ivie undertakes the following steps:

3.2.1 Activating Ivie. Ivie's explanations are triggered when a programming assistant—in our implementation, GitHub Copilot—proposes a suggestion. Ivie listens for suggestions by registering a listener with an internal VS Code API that is triggered whenever "ghost text" (VS Code's phrase for an in-editor suggestion) appears in the editor. Ivie collects the ghost text, and then composes a query to an LLM to retrieve explanations.

3.2.2 Requesting explanations. Ivie then generates explanations. Explanations are composed of two parts:

- A segmentation of the code into explainable parts (i.e., blocks or expressions)
- Natural language descriptions of those parts

Explanations are generated by querying a contemporary LLM. We use the GPT-3.5 `text-davinci-003` model, and access it through OpenAI's API. The LLM is prompted to simultaneously split and explain the code (see an example in Figure 3). The prompt asks for explanations to be brief yet informative. The prompt includes a single example demonstrating the intended output.

One of two prompts is submitted to the LLM, depending on the size of the suggestion; different prompts are used to generate block-level explanations and expression-level explanations. We set the `temperature` to 0.5 and `max_token` to 1000; these parameters were chosen to achieve good explanations with as little latency as possible. When a suggestion consists of two or more lines, Ivie actually submits many requests: block-level explanations are requested for the full suggested text, and expression-level explanations are requested for all of the constituent lines, in parallel. Prompts and parameters appear in the supplemental material.

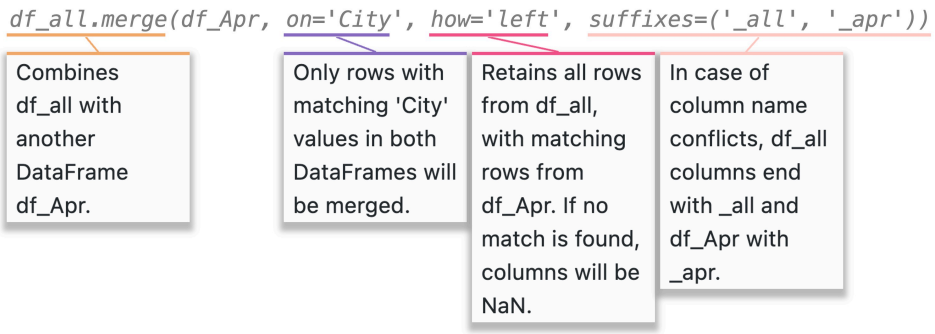

Figure 4: Explaining expressions. After a programming assistant suggests code, Ivie fetches explanations of that code. If the suggestion consists of a single line, Ivie reveals explanations of meaningful expressions within that line, such as function calls, and parameters for those calls. The purpose of these explanations is to make explicit the intent of code that may not be self-evident (as might be the case for the programmer seeking to understand the precise behavior of the suggested table merge operation above).

Our experiences with the LLM suggests that it achieved performance that was, while not perfect, quite good. In a test we ran of 100 generated explanations for two API calls used in our usability study (namely, "`cv.GaussianBlur(img, (5, 5), 0)`" and "`cv.Canny(img, 100, 200)`"), we assessed 97% as being correct. Our criteria for correctness were that explanations needed to be complete (i.e., the function name, return value, and all arguments were described), explanations needed to be accurate (i.e., explanations reflected the labeled expressions without any false information), and that the code needed to be properly segmented (i.e., the LLM correctly detected the bounds of each expression without bleeding over into adjacent expressions or delimiters). The positive usability results in Section 6 suggest that the current error rate around expression segmentation and explanations yields a positive first-use experience. Our discussion elaborates on tensions around using AIs for explaining code in comprehension tools.

3.2.3 Rendering explanations. Upon receipt of a response from the LLM, explanations are rendered as overlays on top of the editor widget. For expression-level explanations, explanations are rendered all at once. For block-level explanations, we request streamed responses from the LLM (i.e., setting the `stream` parameter to `true`). Because block-level explanation requests take longer to fulfill, this allowed us to render explanations for blocks as they are received, rather than waiting for all of them to become available.

Then, explanations are placed next to the expressions they explain. The default position of an expression label is left-aligned beneath an expression. Label placements are further adjusted to avoid overlap. If two labels overlap, the label to the right is moved rightward until there is no longer overlap. Labels are limited to a maximum width (approximately a few words long) to prevent any one label from taking up too much horizontal space. If a label is rendered far away from its expression (which we encode as less than 50% horizontal overlap with the expression), leader lines are added to visually link all explanations to their accompanying expressions.

Requesting explanations for the entire file. When a programmer clicks Ivie's button for showing explanations of the entire file's contents (Section 3.1.3), Ivie submits queries to the LLM as if the entire file's contents were one multi-line suggestion. This has the limitation that when the programmer requests explanations, the explanations might change each time a request is made. This is an artifact of the current implementation; we believe future implementations of Ivie should preserve explanations between requests.

4 SCENARIO

In this section, we describe the experience of interacting with Ivie in a brief narrative walkthrough. For a demonstration of interacting with Ivie, we encourage readers to view the accompanying video figure. Imagine Dorothy, a climate scientist who is about to perform exploratory data analysis of longitudinal climate data about the Amazon rainforest. Dorothy plans to use a common code-based data analysis toolset: namely, a contemporary code editor equipped with Copilot, and a Python environment with the `pandas` data manipulation library and `Matplotlib` visualization library preinstalled. In this scenario, Dorothy is using a code editor that has been extended with Ivie. Dorothy has passing knowledge of `pandas` and `Matplotlib`—enough to do work with them—though she often consults online resources to find out how to fine tune the API functions to manipulate and visualize data as she would like.

Viewing expression explanations. After Dorothy has spent some time loading, cleaning, and manipulating her data, she decides that she is ready to merge some fields from a derived data frame—`df_Apr`—back into the main data frame containing all of her data—`df_all`. When she queries Copilot for a suggestion of how to merge the two frames, she sees the following suggestion, augmented with explanations from Ivie (Figure 4).

Without Ivie, many parts of this line of code would have been difficult to understand. For instance, does a "left" join preserve all of the rows from the `df_all`, `df_Apr`, or both? What values appear in the new columns for rows from `df_all` that do not have a corresponding row in `df_Apr`? The overlay explanations answer these questions, and others. All rows in `df_all` will be maintained; `NaN` values will be inserted wherever the merged rows have new columns. The explanations remind Dorothy to add a guard to her code to check for `NaN` values. Dorothy is also reminded that the names of some of the columns will change, as the `suffixes` argument will rename columns that appear in both data frames. In this way, Dorothy acquires a detailed idea of the result of the merge that

```
def visualize_data(df, max_temp_city, max_rain_city):
    fig, ax = plt.subplots(2, 1, figsize=(14, 10))

    df[df['City'] == max_temp_city]['Temperature'].plot(ax=ax[0])
    ax[0].set_title(f'Yearly Average Temperature for {max_temp_city}')
    ax[0].set_xlabel('Year')
    ax[0].set_ylabel('Temperature (°C)')
    ax[0].yaxis.set_major_formatter(ticker.FormatStrFormatter('%0.1f'))

    df[df['City'] == max_rain_city]['Rainfall'].plot(ax=ax[1], color='green')
    ax[1].set_title(f'Yearly Average Rainfall for {max_rain_city}')
    ax[1].set_xlabel('Year')
    ax[1].set_ylabel('Rainfall (mm)')
    ax[1].yaxis.set_major_locator(ticker.MaxNLocator(nbins=6, integer=True))

    plt.tight_layout()
    plt.show()
```

1. Creates a figure with two vertical subplots.

2. Plot the temperature data for the city with the highest temperature in the top subplot.

3. Plot the rainfall data for the city with the highest rainfall in the bottom subplot.

4. Format the figure and display it.

Figure 5: Explaining multi-line suggestions. When a programming assistant suggests multiple lines of code, Ivie splits up and explains that code. Its explanations appear in the right margin of the editor. The explanations are meant to help a programmer get a high-level understanding of the code. In the pictured scenario, these explanations might help the programmer understand that the two longest sections of the code suggestion configure each of two subplots, each with a different slice of the data.

otherwise would need to have been pieced together from selective reading within the Pandas documentation.

Viewing high-level explanations of code suggestions. Later, Dorothy arrives at a stage of analysis where she would like to visualize a slice of her data. She would like to examine the temperature and rainfall for the cities that have experienced each most extremely. She outlines a Python function for visualization, and then Copilot makes a 16-line suggestion. Ivie augments the suggestion with descriptive labels (Figure 5).

The explanations help Dorothy orient to the code. Her initial skim of the code leads her to erroneously interpret that the function sets up one figure, and then configures the x-axis (in the sequence of lines beginning with "`ax[0]`)", and then the y-axis (in lines beginning with "`ax[1]`"). She pauses to examine the labels that Ivie provides in the right margin, and by the time she reads the third label, she realizes that the blocks of code she associated with the x- and y-axes in fact configure two distinct plots. Now she knows that she can edit the block of code for "`ax[0]`" to configure the plot for the temperature data for the city with maximum temperature. She further validates her understanding of what individual lines of code do by hovering over them—as she hovers over each line, expression-level explanations show for that line. These explanations allow Dorothy to understand how she can configure the formatting string "`%0.1f`" to include more significant digits, and that she can use the "`integer`" parameter to configure whether ticks in the y-axis of the second plot are constrained to integer values.

5 STUDY DESIGN

To evaluate Ivie, we conducted a usability study. The study focused on the impact of Ivie on working with generated code using unfamiliar APIs. We sought answers to the following questions:

Q1. Does Ivie improve understanding of generated code? Ivie's goal is to explain generated code to support high-level understanding, so we evaluated programmers' understanding of generated code.

Q2. Does Ivie influence how much attention programmers give to generated code? Our hypothesis was that programmers would more closely examine generated code when it was accompanied by lightweight explanations.

Q3. How distracting is Ivie? While Ivie was designed to only minimally distract programmers, we sought evidence of just how much distraction they really experienced.

Q4. How does Ivie compare to chat-based AI code comprehension aids? What are the benefits and downsides of Ivie compared to other contemporary alternatives for comprehension assistance?

5.1 Participants

32 programmers were recruited from academic mailing lists in the computer science department at the University of Pennsylvania. 31% were doctoral students, 63% were master's students, and 3% were bachelor's students.[1] 16% of participants reported their level of skill with Python to be advanced, and 50% reported proficient, 31% beginner, and 3% no experience.[2]

As intended, programmers were largely unfamiliar with the libraries involved in the programming tasks. The main library used in the tasks was the OpenCV computer vision library [41]. 38% of programmers reported no experience with this library, 53% were beginners, and 9% were proficient. When asked how familiar they were with computer vision generally, 31% were not at all familiar, 53% a little familiar, 13% somewhat familiar, and 3% very familiar. Participants also had very little experience with the library used in the open-ended programming task (see Appendix B).

31% of programmers had previously used Copilot. 90% of the programmers who had used Copilot reported using it for a few months or less, and only one participant reported using it for nearly a year

[1] Some percentages describing participant backgrounds do not add up to 100%. This reflects occasional non-response to questionnaire items.
[2] All proficiency questions allowed participants to report the level of "expert." No participants selected this level for any question.

(as reported in a multiple-choice question). Although these participants had previously used Copilot, we anticipated they would not experience a bias in the form of a novelty effect with Ivie, because our baseline was also novel in extending Copilot's functionality. Because our baseline involved interacting with an LLM-based chatbot, programmers were also asked about their experience using LLM-based chatbots. All (100%) participants had previously used ChatGPT before, and 59% had additionally used some other chat-based AI. 38% reported having a few months of experience using chat-based AI, 47% had about half a year, 13% had about one year, and 3% reported between one and two years of experience.

5.2 Baseline

Our study compared Ivie to a strong modern baseline. The baseline was the modern VS Code IDE with its built in documentation tooltips, access to a web browser, and an editor plugin providing an AI-based chatbot. We call our baseline condition the "chat condition" due the presence of the chatbot.

We chose a chatbot plugin for the baseline to represent a family of recently-developed AI-based code comprehension plugins (e.g., [12, 20, 21]). In these plugins, the main feature is a chat window to left side of the editor where a programmer can ask questions about their code, and follow-up questions after the chatbot responds. These plugins often support the ability to select arbitrary code and request an explanation of that code from the editor's menus. We chose the plugin "EasyCode" [17] which provides all these features and has lower latency among alternatives.

To promote parity between the baseline and Ivie, we configured both the chatbot and Ivie to use GPT-3.5 for their explanations. Aside from the presence of the chat functionality, the editor was configured identically for both the baseline and Ivie.

5.3 Procedure

Each participant came to our lab for a one-hour-long session. To reduce demand characteristics [42] that might have biased a participant in favor of Ivie, we told them our goal was to understand the influence of two explanation tools—both the chat baseline and Ivie—on understanding generated code. To avoid leaking our role in developing Ivie, we referred to both tools using pseudonyms "chat explanations" and "overlay explanations." Participants consented and completed a questionnaire about their programming background before completing the following stages:[3]

5.3.1 Tutorial. The programmer was instructed in the use of all tools used in the study—Copilot, the chat baseline, and Ivie. The tutorial consisted of a 5-minute slide presentation, and an activity the programmer was guided to create a data visualization with Copilot and invoke both the baseline and Ivie to access explanations.

5.3.2 Timed Programming Tasks. The programmer then undertook two timed programming tasks (Tasks A and B), one with Ivie, and one with the baseline, with order of tasks and interfaces counterbalanced. Each task required the programmer to write a short snippet of image manipulation code using OpenCV. As a prompt, participants were provided an input image, a target output image, a goal (e.g., "blur the image so it resembles the target image"), and starter code. Both tasks were designed to be similar in complexity and focus. Each task require understanding of an OpenCV API that would be unfamiliar to the programmer. The tasks were validated through extensive piloting until we were confident that programmers would almost always be recommended the expected APIs by Copilot, and that the explanations from Ivie would be coherent. Tasks lasted 5 minutes each. This amount of time was sufficient to ensure that all programmers would make some progress (e.g., achieve some blurring of an image), while introducing a cutoff that let us compare how closely each programmer approached to the target parameters across conditions. Programmers were allowed to use a web browser in either condition, though no programmers did so.

After each task, the programmer reported task difficulty using the NASA-TLX questionnaire [58] and answered Likert scale questions about how useful the available tools were in helping them understand generated code. To decrease the likelihood that programmers studied code to an unnatural degree during the programming tasks, we had them complete both tasks before telling them that their understanding of code would be assessed.

5.3.3 Timed Comprehension Assessments. After each programming task, comprehension of generated code was assessed with a timed assessment. The assessment focused solely on the OpenCV API call—i.e., what we believed would be unfamiliar generated code—that was used for image manipulation in the programming task. Each assessment consisted of 20 "yes" / "no" questions about the API call, including the function name and its arguments (see details in Appendix A). Our choice to time the assessment was inspired by priming tests that have been used in program comprehension (e.g., [43]), where response time in answering questions is used to measure the strength of learned associations. A potential threat to validity is that answers to comprehension questions resembled Ivie's in length (i.e., programmers indicated the meaning of an expression by selecting from a set of short text descriptions). We note, however, that answers resembled descriptions of expressions similarly well for the the baseline and Ivie. After running the study, a follow-up investigation of 12 randomly-sampled programmers' session videos showed that, of the correct answers, 83.3% exactly matched the description of the expression from Ivie, and 88.9% exactly matched the description of the expression shown by the baseline. The difference is that descriptions of expressions appeared as part of much longer texts in the chat baseline, which is just the problem that Ivie is meant to address.

5.3.4 Open-Ended Programming Task. The programmer was given the remainder of the time (typically 10 minutes) to explore Ivie's support for an exploratory programming task (see details in Appendix B). This task was designed to help participants ground qualitative feedback on the anticipated usefulness of the tool.

5.3.5 Questionnaire + Interview. The programmer filled out a questionnaire where they reflected on the usability of Ivie and the baseline. If there was time remaining in the session, we conducted a brief semi-structured interview.

5.4 Measures

To answer our research questions, we measured the following:

[3] All questionnaires, task instructions, starter code, and assessments can be viewed in the supplemental material.

1. Code comprehension. We measured programmers' accuracy and speed in replying to comprehension questions.

2. Attention. We measured attention as the amount of time a programmer fixated on generated code. A Tobii Pro Spark eye tracker [59] was used to collect gaze position during all of the programming tasks. The code editor was instrumented to log the positions of all generated code, as well as the baseline's chat window, and all of the labels that Ivie showed. The eye tracker was calibrated before each of the programming tasks.[4]

3. Distraction. Programmers answered Likert scale questions about how distracting they found Ivie's expression explanations, its block explanations, and the chat baseline. They reported task load using the NASA-TLX index.

4. Comparisons to baseline. Programmers answered Likert scale questions comparing Ivie to the baseline, and were asked to elaborate on the comparative benefits of the two tools.

5.5 Analysis

All comprehension questions were assessed using linear mixed-effects models. These models incorporated the tool, the order of tools, task order, and interactions as fixed effects, and participant ID as a random effect. Statistical significance was assessed using an F-test with Satterthwaite's estimate of effective degrees of freedom [48], with the Holm-Bonferroni method [28] to correct p-values. For comparisons of Likert scale responses, we assess significance with a two-tailed Wilcoxon signed-rank test [65]. For all tests, the threshold for statistical significance was $\alpha = 0.05$. Qualitative themes from questionnaires and interviews were determined following a thematic analysis process [5, Chapter 5], wherein one author performed an initial open coding and axial coding pass, a second author revised the complete results, and then the first author validated and made slight adjustments the revised results before writing reports that appear below.

6 RESULTS

6.1 RQ1. Ivie improves code understanding

Comprehension questions. Programmers answered comprehension questions significantly more correctly for tasks completed with Ivie than with the baseline (see Figure 6) ($F = 23.6$, $p < 0.001$). When using, Ivie, they answered an average of 90.2% of questions correctly, in contrast to 65.0% with the baseline ($\sigma = 26.7\%$). Programmers also answered questions more quickly with Ivie ($F = 9.82$, $p = 0.011$), answering questions in an average of 2.8 seconds ($\sigma = 1.0$) about code they had seen in the Ivie condition, versus 3.6 seconds ($\sigma = 1.3$) for the baseline condition.

Self-reported understanding. In their Likert scale feedback, programmers agreed that they understood the code when using Ivie (*median* = 7 out of 7 on a Likert scale, $\sigma = 0.92$), and found the explanations helpful for clarifying the code (*median* = 7, $\sigma = 0.87$). They reported significantly higher agreement than for the baseline for both questions (*median* = 6, $\sigma = 1.69$, $W = 22$, $p = 0.002$, and *median* = 5.5, $\sigma = 1.94$; $W = 26$, $p = 0.001$) (see Figure 8).

[4]With the exception of the first five participants, for whom the eye tracker was calibrated only once at the beginning of the first timed programming task.

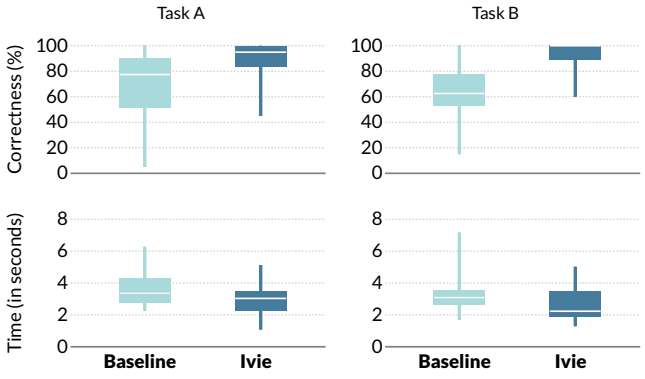

Figure 6: Comprehension results. Each data point for a box plot corresponds to a programmer's score on the assessment or their average question response time. Data is clustered by which tool (Ivie or baseline) was used for the programming task (Task A or B) associated with the assessment. For correctness, higher values are preferred. For time, lower values are preferred. The differences of Ivie vs. chat baseline on both correctness ($p < 0.001$) and time ($p = 0.011$) are statistically significant.

Task progress. An indirect measure of code understanding is programmers' ability to successfully perform the programming task. We observed that 4 programmers ended a task in an error state with the baseline, whereas no programmers did so in the Ivie condition. Appendix C further discusses that, for 2 of 4 configurable parameters in the APIs under study, programmers arrived significantly closer to the target values.

6.2 RQ2. No observed effect of Ivie on attention

Duration of attention on generated code. Programmers spent less time less time looking at generated code in the Ivie condition ($\mu = 3.36$ minutes, $\sigma = 1.56$) in comparison to chat baseline (chat baseline: $\mu = 4.13$ minutes, $\sigma = 1.59$). However, the test did not identify this difference as statistically significant. ($F = 4.25$, $p = 0.14$).

Self-reported effect on attention. Some programmers reported that Ivie influenced the way they looked at code when answering the open-ended questions, and in particular that they more closely examining the generated code with Ivie (P12, P31, P32). P32, for instance, told us that "*when using [Ivie] , I carefully examine the completions instead of quickly accepting them.*" P12 described themselves as "checking everything" when using Ivie. Programmers sometimes felt that Ivie encouraged the behavior of carefully examining code (P1, P26).

6.3 RQ3. Ivie is not (too) distracting

Task load. Programmers reported task load following each programming task. Task load was assessed using five dimensions from the NASA task load index: mental demand, hurry, performance, effort, and frustration. On all dimensions of task load, Ivie was seen as imposing less load than the baseline, including mental demand (Ivie: *median* = 2, $\sigma = 0.92$ vs. baseline: *median* = 3.5, $\sigma = 1.62$; $W = 4.5$, $p < 0.001$), hurry (Ivie: *median* = 2, $\sigma = 1.23$ vs. baseline: *median* = 4, $\sigma = 1.61$; $W = 10$, $p < 0.001$), performance (Ivie: *median* = 1, $\sigma = 1.15$ vs. baseline: *median* = 4, $\sigma = 1.70$;

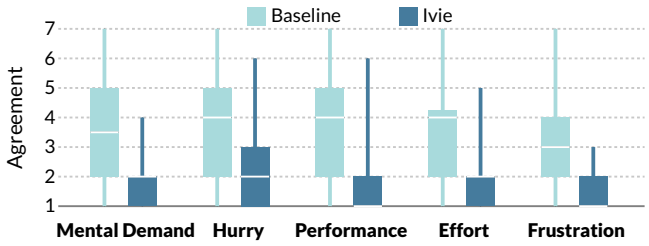

Figure 7: Task load results. Shown are programmers' responses to 5 items from the NASA Task Load Index, collected after each timed programming task. Responses are grouped by which tool was used in the task. For all items, lower values are preferred.

$W = 40$, $p < 0.001$), effort (Ivie: *median* = 2, $\sigma = 1.02$ vs. baseline: *median* = 4, $\sigma = 1.56$; $W = 16$, $p < 0.001$) and frustration (Ivie: *median* = 1, $\sigma = 0.56$ vs. baseline: *median* = 3, $\sigma = 1.79$; $W = 0$, $p < 0.001$) (see also Figure 7).

Self-reported distraction. After each task, programmers were asked to report how distracting they found the explanations provided by the tool in that task. Ivie's explanations were reported as not distracting (*median* = 1 out of 7 on a 7-point scale, $\sigma = 1.01$), and significantly less distracting than the explanations from the chat baseline (*median* = 3, $\sigma = 1.72$; $W = 30$, $p < 0.001$, also see Figure 8). In the closing questionnaire, programmers reported that Ivie was less overwhelming (*median* = 2, $\sigma = 0.94$) and distracting (*median* = 2, $\sigma = 1.02$) than the chat baseline (where 1 indicated that the chat baseline was worse, and 5 indicated that Ivie was worse, also see Figure 9).

6.4 RQ4. Ivie complements chat-based AI code comprehension aids

Direct comparisons to baseline. As mentioned above, programmers reported Ivie as being significantly less distracting and overwhelming than the chat baseline, and reported better understanding the code. When comparing the two tools on a 5-point Likert scale (with 5 indicating a preference for Ivie) programmers also reported that Ivie's explanations were clearer (*median* = 5, $\sigma = 0.87$). Qualitative feedback painted a picture of Ivie as complementary to tools like the chat baseline. In the words of P3, they were both "great tools to use, both have their utilities. Hard to choose only one out of them. Would be helpful to use in conjunction." Discussion Section 7.2.1 talks at length about the perceived benefits of Ivie, including the conciseness and anchored nature of the explanations. Perhaps for these reasons, participants largely indicated that if they were to choose one of the two tools for future tasks involving AI-generated code, they would prefer Ivie (*median* = 4.5 of 5, $\sigma = 0.99$) (see also Figure 9). The comparative advantages of the chat baseline were its support for acquiring conceptual understanding for a task (P22), and deciding what code to write (P6, P26), level of detail in explanations (P3, P29, P32), and the ability to ask follow-up clarification questions (P5, P8, P10, P21). The ability to ask follow-up questions seemed a particularly resonant feature—22 of 32 programmers indicated in their questionnaires that the ability to ask follow-up questions of the chat baseline was either somewhat or very useful.

7 DISCUSSION AND FUTURE WORK

7.1 Limitations

Our study findings are limited in the following ways. First, our results represent usability among a limited sample of the broader programmer population. The participants in our study were primarily master's and doctoral students. About two-thirds of participants had not used Copilot before the study. We anticipate that experiences with Ivie would vary for programmers who are less experienced at programming (and so less able to progress generally), more experienced (and so more knowledgeable about their toolsets), and those who have more established workflows of writing code with programming assistants (and therefore perhaps more resistant to extensions to those programming assistants).

Second, our metric of fixation time does not necessarily capture the aspects of attention that are important. We measured attention as the total amount of time spent looking at generated code. This is a coarse-grained measure, in that it is does not differentiate between desirable attention—like the first read-through of code—from undesirable attention—like time spent debugging code that was not properly understood. In retrospect, we note that more nuanced measures may be necessary to assess whether generated code is attracting the kind of attention it should.

Finally, the study only examined a very limited subset of tasks. The tasks were narrow, focusing on the understanding of individual APIs in short generated programs. To assess the utility of Ivie in supporting programming practice more broadly requires evaluation on a broader set of tasks of various domains and levels of complexity.

7.2 Design implications for instructive copilots

From our design and study, what do we now know about the effective design of instructive copilots?

7.2.1 Reexamining the design goals. A first question is: are the design goals we posed in Section 3 useful guides for instructive copilots? Qualitative feedback provides validation for these goals as useful guides for instructive copilots in this domain:

D1. Anchored. One of the benefits of Ivie was that its explanations were "visually accessible" (P15). Participants appreciated that they were targeted to specific places in the code (P6) and "dissected the parameters." (P4) This stood in contrast to chat-based help, which was seen as giving "long paragraphs of general ideas" (P26) and requiring one to move their "sight outside of my code editor, which was pretty annoying." (P11).

D2. Lightweight. Another frequently-mentioned advantage of Ivie was the lightweight nature of the explanations. Participants appreciated the conciseness of explanations (P4, P6, P15, P18, P22), describing them as easier to understand (P22, P25), simpler (P23), and less overwhelming (P4). In contrast, the chat explanations were often seen as providing too much information (P2, P6, P9, P24).

D3. Easy to invoke. Participants appreciated that Ivie's explanations appeared instantaneously (P11, P19), and in particular that the explanations appeared right after Copilot generated the code (P12, P28). One questionnaire item asked participants how useful they found the ability to receive explanations instantaneously; 20 of 32 participants reported it to be at least somewhat useful.

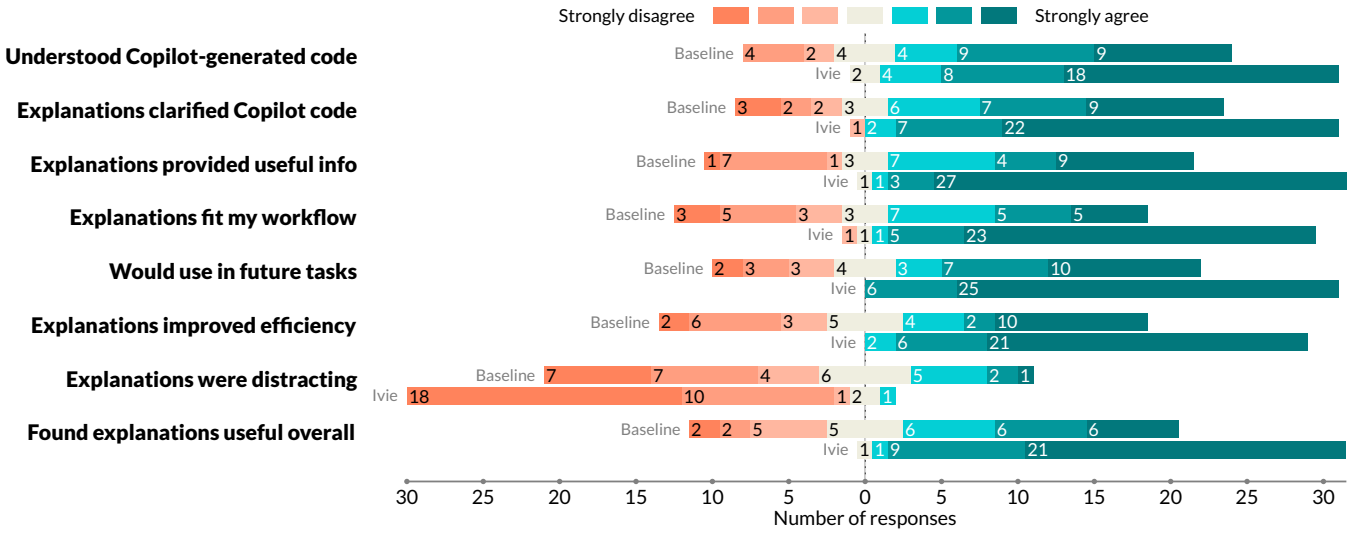

Figure 8: Self-report reflections on tool usability following timed programming tasks. Shown are programmers' responses to 8 self-report questions asked on a 7-point Likert scale after each timed programming task.

D4. Easy to dismiss. Participants generally did not comment on the dismissal mechanism. To us, this indicates that for most participants the design was unremarkable and fit adequately. That said, some participants wished explanations were always-on (i.e., never dismissed) (P17, P24, P26), and P32 wished for a toggle button to turn explanations on and off.

D5. Accessible anytime. Following the Ivie task, 25 of 32 participants conveyed that it was at least somewhat useful to access explanations at any time. Some participants (P19, P26) conveyed that they would have liked if there was even less friction to bring explanations back up.

7.2.2 Expanding the design goals. Our study also revealed opportunities to extend our notion of the instructive copilot. We pose three additional design goals, following participant feedback. Namely, generated explanations should be:

D6. Expandable. Explanations should let programmers ask for more details. Several participants wished for the ability to expand explanations (P12, P18, P20). As envisioned by P12, "ideally, everything starts with a short explanation. If I don't understand something, I could click for more details on that parameter." One way to expand explanations is to let programmers submit follow-up questions about explanations to their AIs (P6, P8, P11, P21, P23).

D7. Adaptable. Explanations should be adapted to the programmer. Some participants desired that Ivie adapted explanations to them (P11, P28). Explanations were seen as unnecessary for code that programmers already familiar with (P27, P30, P31). When an explanation did not convey any useful information, it was suggested that the explanation was not shown (P6, P31); for instance, P6 singled out one such explanation, saying that "Descriptions like 'object' are unhelpful. Filtering such terms would improve [Ivie's explanations]." Future instructive copilots could be selective about what is explained and how, if they were extended to have a reasonable notion of programmer knowledge and needs.

D8. Controllable. It should be possible for users to configure what content is explained and how it is explained. Some participants asked for controls that allowed them to influence the level of detail in explanations (P5, P22). They also desired control over the granularity at which code was explained (i.e., at the expression- or block-level) (P19, P22). Furthermore, some wished for the ability to request explanations for specific code selections (P6, P17, P18).

7.2.3 Bringing instructive copilots outside of the code editor. While in this paper we validate the idea of the instructive copilot as an augmentation to the code editor, we envision that instructive copilots could be useful in other AI generation settings as well. In particular, an instructive copilot could be useful in any setting where there is a timely opportunity for a user to learn more about AI-generated content. For example, perhaps a digital artist would wish to better understand how they reproduce a visual effect that was performed by an AI. For that artist, an instructive copilot might annotate the graphical objects it edited, and reveal the sequence of tools that could be used to attain that effect manually. Another example is writer who is working with AI to compose a passage that is full of references to external work. Perhaps the instructive copilot would describe the nature of the referenced work for any of the references that it generates of which the writer is unfamiliar. We suspect that all such settings would benefit from the same design goals of anchored, lightweight explanations that are easy to invoke and accessible anytime. Where we believe there is interesting variation to explore between applications is in three aspects of an explanation:

Why are explanations needed? For programming assistants, instructive copilots are useful for helping people understand unfamiliar parts of generated code. In other applications, we foresee several reasons to explain AI-generated content. The first reason is that the generated content is confusing, as in the case of unfamiliar code or, say, generated math notation. A second reason is if a user wants to verify AI-generated content, as is the case of a writer who wishes to check a set of generated citations, or follow along with

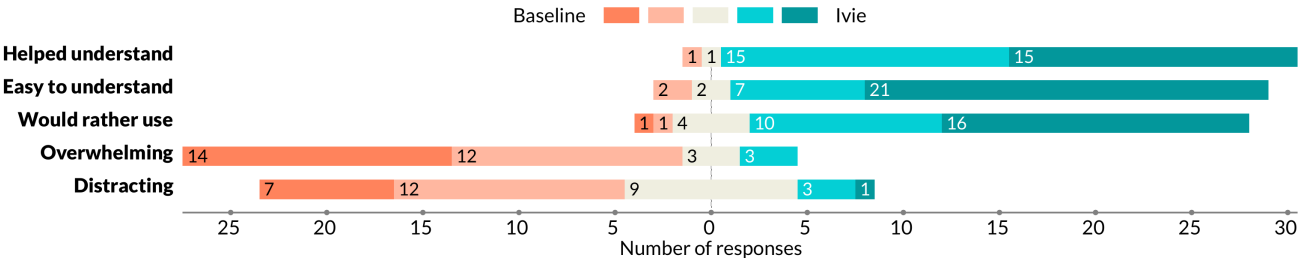

Figure 9: Summative self-report reflections. Programmers compared Ivie to the baseline on a 5-point ordinal scale according to five different dimensions of usability (see labels on left). Pictured is an overall consensus that Ivie better helped programmers understand generated code, Ivie's explanations were easier to understand, programmers would rather use Ivie for future tasks involving AI-generated code, and the explanations from the chat baseline were more overwhelming and distracting.

an AI-generated proof. A third reason is to know how to reproduce AI-generated content, as is the case of the artist who wants to know how an AI achieved a visual effect. The reason for explanation influences the next decisions of what gets explained and how.

The unit of explainable material. In the context of programming assistants, we foresaw that programmers would need to understand individual expression (i.e., components of API calls) and blocks of code (e.g., multi-line idioms). In other scenarios, users would need explanations of generated graphical objects and citations; of texts at the level of individual words (like jargon), phrases (claims), full passages (chains of reasoning), and even multimedia content like segments of audio and video (e.g., effects produced in creativity tools). There needs to be a mechanism for segmenting AI-generated content (as we do with an external LLM) to identify explainable units and allows them to serve as anchors.

The explanation. The content of an explanation follows from the purpose of explanation. If an explanation is meant to clear up confusing material, it might provide generated descriptions or links to external references. If it is instead meant to aid in the verification of AI-generated content, it might instead link to additional documents that provide supporting or conflicting information. And if it is meant to help someone reproduce AI-generated content, then it might provide procedures for doing so. In the case of programming assistants, we recommended that explanations be extremely concise, in part because the explanations would be interspersed between code generations that may be just seconds apart. In other settings, it may be less necessary for explanations to be concise, particularly if they are providing complex external information that may be supportive in verifying AI-generated content.

Dismissal mechanism. Our design makes use of automatic dismissal when a user clicks away from generated content. This may not be the only appropriate choice. In other cases, where a user wants to continually refer back to the explanation (as in a writer who may want to continue looking at clips from a cited document), it may be better to preserve explanations until the user moves their focus away from the current paragraph or section. In other circumstances, it may be appropriate for explanations to be always-on, as was the desire of some programmers in our study.

7.2.4 A critique of using AI to explain AI-generated content. The possibility of prototyping a tool like Ivie has only recently become possible with the release of contemporary AI tools. As with all AIs, those used by Ivie make mistakes. If the AI for Ivie makes a mistake, a programmer may draw inappropriate conclusions about the behavior of their code. In the worst case, this could lead to significant bugs and negative side effects in the generated code. In less severe cases, it could lead the programmer to reject useful suggestions, or slow them down as they attempt to comprehend code. In this way, one of the motivations of this tool—to better inform users about their AI-generated content—is undermined in part because AI-generated explanations may be themselves incorrect.

We see these potential downside as further motivating research in AI to produce validated texts. In the meantime, Ivie might still be deployable in settings where they provide value despite inaccuracies. Programmers already use tools like ChatGPT [11] as a code understanding aid. Conventional documentation itself contains inaccuracies and outdated information [46, 60], and programmers adjust to this reality. Amidst inaccuracies, we see Ivie as playing a role in supporting more exploratory tasks where the potential damage of misunderstanding is limited, and in being consulted alongside up-to-date documentation for higher-stakes development tasks.

8 CONCLUSION

In this paper, we propose the notion of an instructive copilot, a generative AI assistant that provides just-in-time explanations of its generations. We explore this idea in the setting of programming assistants, developing a tool called Ivie that explains unfamiliar APIs in generated code. Our goals in designing Ivie were to provide explanations that were anchored to expressions in generated code, lightweight, easy to invoke, easy to dismiss, and accessible anytime. In a usability study, Ivie led to better comprehension of unfamiliar APIs in generated code versus a chat AI baseline. Ivie also reduced task load and self-reported distraction. Programmers preferred Ivie and saw Ivie's concise explanations as complementary to longer-form programming help like AI chat aids. Furthermore, our study revealed opportunities to improve explanations by making them expandable, adaptable, and configurable. Altogether, this work shows the value of lightweight, anchored AI support as a tool in the programming help-seeking toolkit.

ACKNOWLEDGMENTS

We thank our colleagues from Penn HCI for their feedback on prototypes, study designs, and paper drafts. We also thank Sorin Lerner for advice on how to implement overlays for VSCode.

REFERENCES

[1] Naser Al Madi. 2023. How Readable is Model-generated Code? Examining Readability and Visual Inspection of GitHub Copilot. In *Proceedings of the International Conference on Automated Software Engineering*. ACM, Article 205.

[2] Amazon CodeWhisperer. Retrieved July 25, 2023 from https://aws.amazon.com/codewhisperer/

[3] Shraddha Barke, Michael B. James, and Nadia Polikarpova. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proc. ACM Program. Lang.*, Article 78 (April 2023).

[4] 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 (2022), 35–57.

[5] Ann Blandford, Dominic Furniss, and Stephann Makri. 2016. *Qualitative HCI research: Going behind the scenes*. Morgan & Claypool Publishers.

[6] Ruven Brooks. 1983. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies* 18, 6 (1983), 543–554.

[7] Brian Burg, Amy J. Ko, and Michael D. Ernst. 2015. Explaining visual changes in web interfaces. In *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 259–268.

[8] John M Carroll. 1990. *The Nurnberg funnel: Designing minimalist instruction for practical computer skill*. MIT Press.

[9] Paul Chandler and John Sweller. 1992. The split-attention effect as a factor in the design of instruction. *British Journal of Educational Psychology* 62, 2 (1992), 233–246.

[10] Kerry Shih-Ping Chang and Brad A. Myers. 2012. WebCrystal: Understanding and Reusing Examples in Web Authoring. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, 3205–3214.

[11] ChatGPT. Retrieved September 15, 2023 from https://chat.openai.com

[12] Codeium. Retrieved July 25, 2023 from https://marketplace.visualstudio.com/items?itemName=Codeium.codeium

[13] William Perry Crichton. 2022. *Revisiting Program Slicing with Ownership-Based Information Flow*. Ph. D. Dissertation.

[14] Françoise Détienne. 2001. *Software Design–Cognitive Aspects*. Springer Science & Business Media.

[15] Thomas Dohmke. 2022. GitHub Copilot is generally available to all developers. Retrieved July 25, 2023 from https://github.blog/2022-06-21-github-copilot-is-generally-available-to-all-developers/

[16] Ekwa Duala-Ekoko and Martin P. Robillard. 2012. Asking and answering questions about unfamiliar APIs: An exploratory study. In *Proceedings of the International Conference on Software Engineering*. IEEE, 266–276.

[17] EasyCode. Retrieved July 25, 2023 from https://www.easycode.ai/

[18] Kasra Ferdowsi, Ruanqianqian Huang, Michael B. James, Nadia Polikarpova, and Sorin Lerner. 2023. Live Exploration of AI-Generated Programs. (2023). arXiv:2306.09541 [cs.HC]

[19] Thomas Fritz, Andrew Begel, Sebastian C. Müller, Serap Yigit-Elliott, and Manuela Züger. 2014. Using Psycho-Physiological Measures to Assess Task Difficulty in Software Development. In *Proceedings of the International Conference on Software Engineering*. ACM, 402–413.

[20] Genie AI. Retrieved July 25, 2023 from https://marketplace.visualstudio.com/items?itemName=genieai.chatgpt-vscode

[21] GitHub Copilot X. Retrieved July 25, 2023 from https://github.com/features/preview/copilot-x

[22] GitHub Copilot · Your AI pair programmer. Retrieved July 25, 2023 from https://github.com/features/copilot

[23] GitHub Next | Code Brushes. Retrieved September 10, 2023 from https://githubnext.com/projects/code-brushes/

[24] Elena L. Glassman, Tianyi Zhang, Björn Hartmann, and Miryung Kim. 2018. Visualizing API Usage Examples at Scale. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, Article 580.

[25] Andrew Head, Codanda Appachu, Marti A Hearst, and Björn Hartmann. 2015. Tutorons: Generating context-relevant, on-demand explanations and demonstrations of online code. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*. IEEE, 3–12.

[26] Joshua Hibschman and Haoqi Zhang. 2016. Telescope: Fine-tuned discovery of interactive web UI feature implementation. In *Proceedings of the Symposium on User Interface Software and Technology*. ACM, 233–245.

[27] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. 2018. Augmenting Code with In Situ Visualizations to Aid Program Understanding. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, Article 532.

[28] Sture Holm. 1979. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics* (1979), 65–70.

[29] Amber Horvath, Michael Xieyang Liu, River Hendriksen, Connor Shannon, Emma Paterson, Kazi Jawad, Andrew Macvean, and Brad A. Myers. 2022. Understanding How Programmers Can Use Annotations on Documentation. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, Article 69.

[30] Amber Horvath, Brad A. Myers, Andrew Macvean, and Imtiaz Rahman. 2022. Using Annotations for Sensemaking About Code. In *Proceedings of the Symposium on User Interface Software and Technology*. ACM, Article 61.

[31] Philipp Kather, Rodrigo Duran, and Jan Vahrenhold. 2021. Through (Tracking) Their Eyes: Abstraction and Complexity in Program Comprehension. *ACM Transactions on Computing Education*, Article 17 (November 2021).

[32] Amy J. Ko, Robert DeLine, and Gina Venolia. 2007. Information needs in collocated software development teams. In *Proceedings of the International Conference on Software Engineering*. IEEE, 344–353.

[33] Sorin Lerner. 2020. Projection Boxes: On-the-Fly Reconfigurable Visualization for Live Programming. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, Article 367.

[34] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. 2024. A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges. In *Proceedings of the International Conference on Software Engineering*. IEEE/ACM, Article 52.

[35] Mark Liffiton, Brad E. Sheese, Jaromir Savelka, and Paul Denny. 2024. CodeHelp: Using Large Language Models with Guardrails for Scalable Support in Programming Classes. In *Proceedings of Koli Calling International Conference on Computing Education Research*. ACM, Article 8.

[36] Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2024. Reading Between the Lines: Modeling User Behavior and Costs in AI-Assisted Programming. (2024). To appear.

[37] Takao Nakagawa, Yasutaka Kamei, Hidetake Uwano, Akito Monden, Kenichi Matsumoto, and Daniel M. German. 2014. Quantifying programmers' mental workload during program comprehension based on cerebral blood flow measurement: A controlled experiment. In *Proceedings of the International Conference on Software Engineering*. ACM, 448–451.

[38] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad A. Myers. 2024. Using an LLM to Help With Code Understanding. In *Proceedings of the International Conference on Software Engineering*. To appear.

[39] Jakob Nielsen. 1994. Enhancing the explanatory power of usability heuristics. In *Proceedings of the CHI conference on Human Factors in Computing Systems*. ACM, 152–158.

[40] OpenAI. 2023. GPT-4 Technical Report. (2023). arXiv:2303.08774 [cs.CL]

[41] OpenCV. Retrieved September 15, 2023 from https://opencv.org/

[42] Martin T Orne. 2017. On the social psychology of the psychological experiment: With particular reference to demand characteristics and their implications. In *Sociological methods*. Routledge, 279–299.

[43] Nancy Pennington. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology* 19, 3 (1987), 295–341.

[44] Hannah Potter, Ardi Madadi, René Just, and Cyrus Omar. 2022. Contextualized Programming Language Documentation. In *Proceedings of the International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM.

[45] PyGame. Retrieved September 15, 2023 from https://www.pygame.org

[46] Martin P. Robillard and Robert DeLine. 2011. A field study of API learning obstacles. *Empirical Software Engineering* 16 (2011), 703–732.

[47] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. 2015. How developers search for code: a case study. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*. ACM, 191–201.

[48] Franklin E. Satterthwaite. 1946. An approximate distribution of estimates of variance components. *Biometrics bulletin* 2, 6 (1946), 110–114.

[49] Teresa M. Shaft and Iris Vessey. 1995. The Relevance of Application Domain Knowledge: The Case of Computer Program Comprehension. *Information systems research* 6, 3 (1995), 286–299.

[50] Bonita Sharif and Jonathan I. Maletic. 2010. An Eye Tracking Study on camelCase and under_score Identifier Styles. In *Proceedings of the International Conference on Program Comprehension*. IEEE, 196–205.

[51] Ben Shneiderman. 1976. Exploratory experiments in programmer behavior. *International Journal of Computer & Information Sciences* 5 (1976), 123–143.

[52] Ben Shneiderman and Richard Mayer. 1979. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences* 8 (1979), 219–238.

[53] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. 2014. Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the International Conference on Software Engineering*. ACM, 378–389.

[54] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. 2006. Questions Programmers Ask during Software Evolution Tasks. In *Proceedings of the International Symposium on Foundations of Software Engineering*. ACM, 23–34.

[55] Elliot Soloway and Kate Ehrlich. 1984. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering* SE-10, 5 (1984), 595–609.

[56] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. IntelliCode Compose: Code Generation using Transformer. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on*

the Foundations of Software Engineering. ACM, 1433–1443.
[57] 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. In *Proceedings of the Annual Workshop at the Intersection of PL and HCI.*
[58] TLX @ NASA Ames - Home. Retrieved September 15, 2023 from https://humansystems.arc.nasa.gov/groups/TLX/
[59] Tobii. Enter the world of eye tracking with Tobii Pro Spark - Tobii. Retrieved September 15, 2023 from https://www.tobii.com/products/eye-trackers/screen-based/tobii-pro-spark
[60] Gias Uddin and Martin P. Robillard. 2015. How API documentation fails. *IEEE Software* 32, 4 (2015), 68–75.
[61] 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 CHI Conference on Human Factors in Computing Systems*. ACM, Article 332.
[62] Visual Studio Code. Retrieved September 15, 2023 from https://code.visualstudio.com/
[63] A. von Mayrhauser and A. M. Vans. 1993. From program comprehension to tool requirements for an industrial environment. In *Proceedings of the Workshop on Program Comprehension*. IEEE, 78–86.
[64] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 1069–1088.
[65] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in Statistics: Methodology and Distribution*. Springer, 196–202.
[66] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E. Hassan, and Shanping Li. 2017. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering* 44, 10 (2017), 951–976.
[67] Frank F. Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-IDE Code Generation from Natural Language: Promise and Challenges. *ACM Trans. Softw. Eng. Methodol.* 31, Article 29 (March 2022).
[68] Litao Yan, Elena L. Glassman, and Tianyi Zhang. 2021. Visualizing Examples of Deep Neural Networks at Scale. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. ACM, Article 313.
[69] Litao Yan, Miryung Kim, Bjoern Hartmann, Tianyi Zhang, and Elena L. Glassman. 2022. Concept-Annotated Examples for Library Comparison. In *Proceedings of the Symposium on User Interface Software and Technology*. ACM, Article 65.

A TIMED COMPREHENSION ASSESSMENT

The comprehension assessments asked two kinds of questions:

- **Identify the API**. The participant was told to imagine that they were trying to achieve a particular image manipulation goal and then shown the name of an API. They were asked whether the API could be used to achieve that goal. 8 questions were shown, 1 with the correct API from the programming task, and 7 with incorrect APIs either entirely made up or taken from the OpenCV documentation.
- **Identify the purpose of a parameter**. The participant was shown the API signature without parameter names (e.g., "`cv.Canny(❶, ❷, ❸)`"). They were directed to a specific parameter (e.g., "❶"), shown a phrase (e.g., "higher threshold") and asked if the phrase described that parameter. 4 questions were shown for each parameter, with 1 correct description and 3 incorrect ones. We asked 12 such questions.

The test interface and procedure was designed so that response time measured only time to think and respond, to the extent possible. Responses were entered by pressing numeric pad keys "1" (for yes), "2" (for no), or "3" (for unsure), and then the Enter key to confirm. The programmer was trained in this system on sample questions before answering any of the questions we planned to analyze. They were told to answer questions as quickly as they were able. Screenshots of the assessment interface and a listing of all questions can be viewed in the supplemental material.

B OPEN-ENDED PROGRAMMING TASK

For the open-ended programming task, participants were asked to create a lightweight version of the classic Mario platformer game using Pygame [45], a 2D game development library for Python. Participants were given no starter code. Rather, they were provided with a blank code Python script, a terminal from which they could run that script, and a folder containing graphics they could use in their game. Links to these graphics are listed in the supplemental material. Participants had access to Ivie, EasyCode (the baseline chat AI), and the web. The task was designed to require programmers to see a significant number of unfamiliar API methods and Pygame-specific programming idioms. 78% participants reported having no prior experience with Pygame, and 22% reported being beginners. Most participants were unfamiliar with the domain of game development—72% were not at all familiar, 19% a little familiar, and 9% somewhat familiar. Participants were not expected to finish, but rather to just make some progress. About half of participants got to a stage of development where a Mario sprite appeared on the screen and could be controlled with arrow keys.

C ASSESSING PROGRESS ON TIMED PROGRAMMING TASKS

A supplementary measure of task success was the extent to which participants' final code resembled a reference implementation. Each task required programmers to configure a set of parameters for an image processing API to control visual effects like blur or edge tracing to replicate a target image. In both the Ivie and baseline condition, we collected the values of parameters in participants' code at the time they were cut off. For 2 of 4 parameters, participants were significantly closer to the target values when they used Ivie versus using the baseline (see Figure 10). For the other 2 parameters, the differences were not significant. Significance was assessed by conducing an unpaired t-test of the L1 distance of parameter values to the target values for 1D parameters, and L2 distance for 2D parameters (e.g., the `ksize` tuple). Detailed results are as follows: For task A, there were two parameters. The first parameter, "`ksize`," had a target value of `(21, 21)`. The L2 distance of programmers' final parameters to this target value was 1.41 (σ = 11.8) in the Ivie condition versus 14.8 (σ = 16.3) in the baseline condition; this difference was statistically significant (p = 0.004). No significant difference was seen for the distance to the target value for the second parameter "`sigmaX`" between Ivie (μ = 2.56, σ = 2.89) and the baseline (μ = 7.81, σ = 29.84; p = 0.19). For task B, programmers were significantly closer to the target value of the "`threshold_low`" parameter in the Ivie condition (μ = 10.62, σ = 11.16) than in the baseline condition (μ = 50.31, σ = 46.38; p = 0.003), though not for the for the "`threshold_high`" parameter (Ivie: μ = 8.44, σ = 39.79 vs. baseline: μ = 20.0, σ = 46.77; p = 0.472).

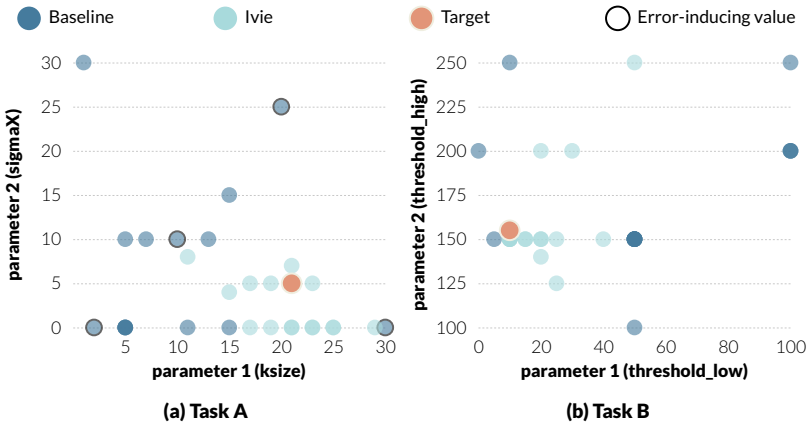

Figure 10: Convergence on parameter values. Shown are plots conveying how far participants were from the reference solution when they ran out of time in each timed programming task. Each plot visualizes a parameter space for the key API function that the programmer need to use in a task. Each task required configuration of two parameters. All parameters are 1-dimensional numeric, except for "`ksize`" ("parameter 1" in (a)), which was a tuple of two values, which were often set to be equal; we show only one of the values from the tuple. Red dots mark target configurations, which are (21, 5) for task A and (10, 155) for task B. Light blue dots represent values achieved when programmers used Ivie, and dark blue plots represent values achieved in the baseline condition. Some points overlap: for instance, there are 5 overlapping baseline points in (a) at (5, 0); 8 overlapping baseline points in (b) at (50, 150) (this was frequently the default initial generated configuration); and 3 overlapping baseline points in (b) at (100, 200).