Live in the Loop: Rapid Run-time Feedback for Prompts

Toni Mattis
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
toni.mattis@hpi.uni-potsdam.de

Abdullatif Ghajar
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
abdullatif.ghajar@student.hpi.uni-potsdam.de

Tom Beckmann
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
tom.beckmann@hpi.uni-potsdam.de

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
robert.hirschfeld@hpi.uni-potsdam.de

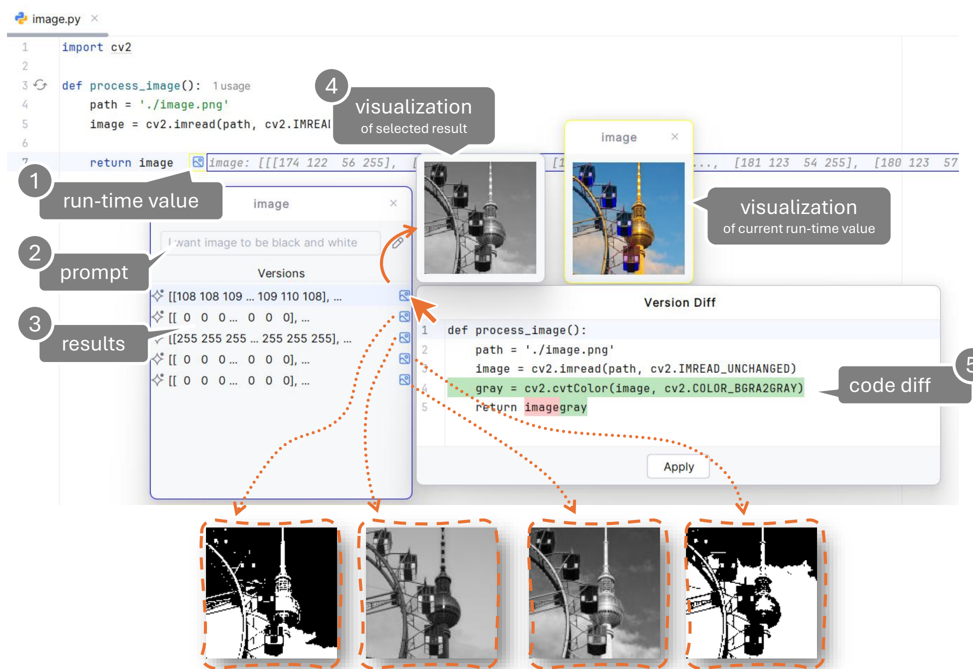

Figure 1: ReFiQ is an LLM-backed live programming tool that allows programmers to (1.) see run-time values, (2.) specify intended behavior in a prompt and (3.) select the desired behavior based on its effects on run-time values, which can be further investigated using (4.) visualizations and (5.) the code diff that would be applied to achieve the selected result.

Abstract

Programmers formulate prompts for code generation based on their understandings of problem domain and LLM ability to execute instructions. A deficiency in either understanding yields inadequate generated code, requiring programmers to revise their understandings based on deficiencies observed in the generated code.

We propose ReFiQ ("Result-first Queries"), a tool that simultaneously offers concrete run-time execution results of multiple generated code variations for one prompt. In an exploratory user study (n=8), we observed that programmers systematically compared those results to identify issues in their understandings, deliberately formulated single prompts to explore a range of options of a domain concept, and were more often encouraged to make a deliberate decision informed by their observations instead of directly applying generated code. Participants in our study required fewer iterations to arrive at a satisfying solution in ReFiQ than in our baseline, GitHub Copilot.

This work is licensed under a Creative Commons Attribution 4.0 International License.

CHI '26, Barcelona, Spain

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2278-3/26/04
https://doi.org/10.1145/3772318.3791731

CCS Concepts

• Software and its engineering → Integrated and visual development environments; **• Human-centered computing →** *Natural language interfaces*; *Graphical user interfaces*; *Empirical studies in HCI*.

Keywords

Result-first, Large Language Models, Code Generation, Live Programming, Prompts, Exploratory Study

ACM Reference Format:

Toni Mattis, Abdullatif Ghajar, Tom Beckmann, and Robert Hirschfeld. 2026. Live in the Loop: Rapid Run-time Feedback for Prompts. In *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems (CHI '26), April 13–17, 2026, Barcelona, Spain.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3772318.3791731

1 Introduction

A growing number of programmers rely on AI assistants based on large language models (LLMs) that generate and edit code in response to a natural-language prompt, such as GitHub Copilot [5], Cursor [2], or Claude Code [1]. In such a setting, code can be created with minimal manual effort, shifting the focus to prompts that instruct LLMs what to generate and how to validate it.

As prompts are in natural language, they are unconstrained and do not provide any affordances that convey what the underlying LLM can and cannot do or which information it requires to solve a task effectively. At the same time, programmers are no longer required to form a detailed formal *intent* but can attempt to express even incomplete goals. Subramonyam et al. [25] refer to this as the *gulf of envisioning* that can lead to any of three *gaps*: There is the *capability gap* where programmers might misjudge what tasks the LLM is capable of performing. For example, they might ask for too many steps at once so that the LLM erroneously omits some, or wrongly assume that the LLM has information that is actually missing from training or context, leading to hallucination. There is also the *instruction gap* where programmers might not be able to express their goals effectively in a prompt. For example, they might use vocabulary that misleads the LLM. And there is the *intentionality gap* where programmers might not engage with the problem domain sufficiently to be accurate in their prompt or evaluate whether the LLM-generated output is what they wanted. For example, their understanding might lack an important domain concept, so they neither know its name to use in the prompt, nor can they recognize generated code that uses this concept as correct.

An opportunity underlying this work is the presentation of *multiple results*. By generating multiple results to one prompt, systems can potentially help reduce those three gaps [25]. As multiple results show users a broader range of possible interpretations of their prompt, an inadequate result is less likely to have resulted from the LLM's inherent non-determinism, helping to clarify the capability gap. The multitude of interpretations further allows programmers to better identify weaknesses in their prompt that manifest as an instruction gap: when interpretations all have the same inadequate property, it is more likely that this property was not well specified.

Importantly, these gains of multiple results depend on the programmers' ability to effectively judge properties of the program's output or behavior, which is often referred to as the gulf of evaluation [13]. Prior work on live programming has demonstrated that showing results of run-time execution is one effective way to bridge the gulf of evaluation. Instead of simulating the effects of the abstract code in their head, programmers see a concrete result that can help them understand complex aspects of the source code by an example [8, 22, 23].

In this paper, we propose ReFiQ[1], a tool that presents programmers with multiple results for their queries and presents the result of the run-time execution of the proposed code first.

This creates a novel workflow in which evaluating the prompt and its run-time consequences precedes selecting and integrating generated code changes. The interface available to programmers is designed to make use of run-time values from invoking the AI to settling on a desired result, extending on previous work using run-time values only in specific situations.

The design choice to move presentation of results first presents new tradeoffs compared to other tools in the same design space. Through its design, ReFiQ promises to deliver earlier and richer feedback, as it helps programmers to isolate issues in their understanding of the LLM or the domain, thereby helping to address the three gaps.

We analyzed the effect ReFiQ has on programmers' workflows in an exploratory user study. We gave eight participants two exploratory programming tasks that required them to take decisions on the design of solutions and outputs. As a point of comparison, we asked them to perform one of the tasks using GitHub Copilot in the same live programming environment. We observed that multiple results allowed participants to refine their intention and to explicitly prompt for ranges of options. The multiple results also required them to make deliberate choices about the output they want, which we believe may help reduce the gap of intentionality. At the same time, participants indicated in a post-survey response that they had a better understanding of the code in GitHub Copilot but had more trust in the correctness of the code in ReFiQ. We did not observe a notable increase in understanding of the task domain in either tool.

2 Background and Related Work

Live Programming. Programming can be significantly aided by fast iteration cycles and immediate feedback [4, 27]. *Live programming environments* such as Smalltalk [10] make this especially convenient, which have been characterized by Tanimoto [26] as *updating as soon as changes are made* while continuing to run, and by Rein et al. [23] as *maintaining the impression of editing a running program* to include fast restarts that still yield immediate feedback. Notebook-style programming environments, like Jupyter [15], share similar properties by interleaving incremental evaluation with intermediate results while keeping run-time state alive.

When fine-grained feedback is desired, *in-situ* approaches to live programming exist that allow programmers to gain continuous run-time feedback on individual expressions. These include Babylonian Programming [22], projection boxes [17], and Exemplars [6]. A common mechanism in these approaches is the display of run-time values either next to or in line with the code producing them, for

[1]"Result-First Queries", Rafiq is "friend" or "companion" in Arabic.

example, *probes* in Babylonian Programming log an expression's value at run-time and can make use of rich representations (e.g., rendering a graphical representation or displaying an interactive part of the system). We will adapt probes that update immediately on code change as the live programming mechanism to display run-time data in this work. A key novelty will be the use of probe *output* as entry point to specify intended change and as identifier for (future) program versions.

In addition to purely responsive, edit-triggered systems, Tanimoto [26] extended the hierarchy of live programming systems to include the (back then hypothetical) classes of *tactically and strategically predictive liveness* (Levels 5 and 6) that allow the selection of desired behavior from already running future alternatives. Our work will closely approach "Level 5", replacing autonomous prediction with prompted AI generation but still providing live output of future alternatives.

Exploring Alternatives via Microversioning. As many programming activities benefit from comparing different alternative versions of a single program to obtain feedback on progress, fine-tuning a value or algorithm, or finding a version to revert to, several designs make use of *microversioning* in a broader sense. For example, CoExist [24] offers tools for immediate access to source code and run-time information of previous development states in a live programming environment, Variolite [14] introduces in-situ microversioning in Python for exploratory programming, EditTransactions [20] are a mechanism to validate changes in a live programming environment in a local scope before extending them to the entire running program, Fork It [29] allows exploring concurrent versions in a notebook, Boba [19] introduces a domain-specific language to specify variations in data analyses and supports domain-specific views that convey the differences across variations, and more recently, Exploriants [3] combines microversioning with live programming to compare differences in run-time behavior across configurable dimensions of variation.

Microversioning was originally designed to represent variation in human-edited code on the tool level, but when LLMs change a program, they can become important as a way to recover after unintended changes or explore alternative futures suggested by an LLM, as also explored in this paper.

Interactive Program Synthesis. Before LLMs reliably produced code, traditional program synthesis was already used interactively to minimize feedback loops. The SnipPy environment [9] uses projection boxes to track the evolution of state in a Python function at run-time, but allows for direct editing of a value while a program synthesizer attempts to modify the program so that it matches the desired value. Maniposynth [12] provides a second view on a functional program in which its workings are visualized using an example. The visualizations support direct manipulation that causes the program to change accordingly.

In FlashProg [21], users also interact with data to constrain program behavior. To resolve ambiguity, FlashProg asks users to choose from a list of program outputs that fulfill the current constraints but differ for a not-yet-specified case. In addition, users can click on a natural language representation of program operators and choose alternative operators that also fulfill the current list of constraints. These three systems are examples of the *programming by example (PbE)* paradigm, where programmers specify behavior in terms of concrete data they expect at certain stages of a program or algorithm.

As LLMs became effective at synthesizing code from natural language, tools like GitHub Copilot [5], Cursor [2], or Claude Code [1] entered programming workflows. Due to their generality that tries to cover a wide range of development workflows and setups, they work almost exclusively on code level and leave evaluation to the user. However, they are mostly unconstrained in their input, replacing PbE by natural language specification rather than exact values.

A notable evolution is *Code Shaping* [31] that proposes to use sketching on code and (visual) output instead of natural language prompts, which shortens the feedback loop for several interactions that can be expressed using a pen on a tablet.

Understanding LLM Suggestions. Grounded abstraction matching [18] is a mechanism for helping users understand the effect of formulations of their prompt for generating Python programs to analyze spreadsheet data. The mechanism translates generated code back to natural language based on supported abstractions in the system to help establish a predictable and systematic mapping for the user. In a between-subjects user study, grounded abstraction helped users address instruction and capability gaps.

To better explain LLM suggestions to programmers, *Ivie* [30] enriches the code suggestions with in-situ explanations in natural language, demonstrating an improved code understanding as participants navigate an unfamiliar API.

LEAP [8] integrates a live programming mechanism, projection boxes, with LLM-generated code to lower the cost of evaluation for programmers. In a between-subjects user study with fixed and open prompts, the authors find that through the lower cost of evaluation, programmers' cognitive load, as well as under- and over-trust in the generated code, was reduced. The system described in our work shares similarities with LEAP but explores a different point in the design space. While ReFiQ uses probes and LEAP uses projection boxes, we believe that the effect for the user is likely comparable regarding the LLM-based programming workflow, as both are mechanisms to display run-time data in close proximity to the code that computes them. More importantly for the workflow, LEAP displays multiple code proposals in response to a prompt, whereas ReFiQ displays multiple run-time results. In LEAP, users can inspect the result of a code proposal one after another. In ReFiQ, users can inspect the code for a result by opening a pop-up for each. This difference is significant, as ReFiQ allows users to not even engage with a code proposal that does not produce a desirable result, but may also drive users to not even engage with the code proposal they eventually accept. Similar to LEAP, our user study uses GitHub Copilot as a baseline, as it is a system that participants were already familiar with. Unlike LEAP, our study design exposes tasks of higher complexity and leaves the decision on how to approach the task to the participants, except for the choice of the tool (ReFiQ vs. GitHub Copilot). This design makes sense for our study, as we were exploring the effect of ReFiQ's design on programmers' workflows, whereas LEAP's study was designed to confirm specific hypotheses.

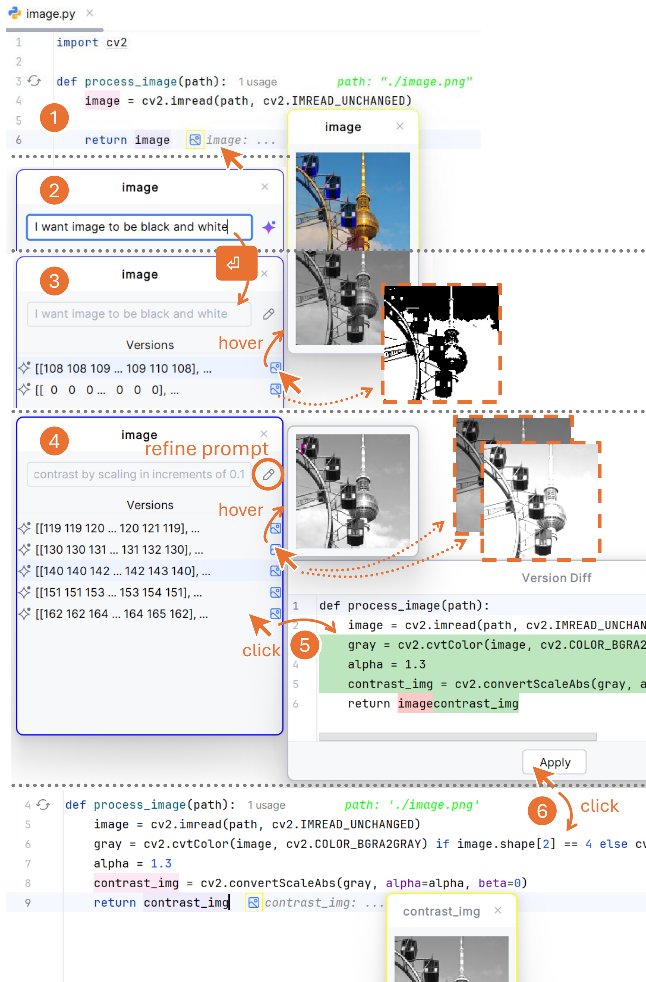

Figure 2: An example workflow in ReFiQ. (1) Starting with a Python program that loads a given image, we view the current run-time value of the image. (2) We click the inline preview to open a prompt and type how we want the run-time value to change. (3) Results from the execution of multiple generated alternatives appear. Each one can be hovered to show a preview. By comparing them, we realize that our prompt was ambiguous: the image could be grayscale or pure black-and-white. (4) We refine the prompt to ask for grayscale and ask to produce variations with different contrast. (5) We choose a version, inspect its code, and (6) finally apply it.

3 Result-first Exploration

Our approach changes the way programmers *initiate and choose* AI suggestions in a way that puts run-time values and results of AI suggestions at its center, which subsequently enables a better presentation of *multiple results*. In this section, we map this design both from a workflow and a user interface perspective.

3.1 Cognitive Walkthrough

We demonstrate our envisioned workflow in ReFiQ through a walkthrough. Our Python programmer Sage is tasked with a computer vision problem and is currently writing a pre-processing function that should supply a pipeline with a high-contrast image. She starts with an empty Python function in an IDE with ReFiQ that gets passed an example image as its argument and just returns the image for downstream processing.

A. Sage thinks that the image does not need to be in full RGB color for downstream tasks, thus she clicks the variable containing the image to open a prompt pop-up. She types "Convert the image to black and white" and hits enter.

B. Within the next second, the pop-up window is filling with *different variations* of her example image. Since Sage previously selected the image variable, the system knows which run-time data she might be interested in. She notices that one image is a bitmap of literally black and white pixels, another one is a grayscale image, and the remaining suggestions are bitmaps again, but different areas are black or white. Figure 1 illustrates this outcome in our implementation of ReFiQ in the Python IDE PyCharm.

She recognizes her prompt as ambiguous and quickly rephrases her prompt as "Convert the image to grayscale".

C. The rephrased prompt yields several grayscale variations of the example image, which are harder to discern. Sage hovers over the solutions to *see the code*, discovering that they extract different color channels, while one solution uses a built-in function that does a direct conversion and looks the most realistic. She selects the option, and the suggested code is inserted into the function.

D. Then, Sage attempts to increase contrast by invoking the prompt window at the next line, prompting "Increase contrast of the image", resulting in several variations where one barely shows any improvement, while several others exhibit large oversaturated areas. A glance at the source code reveals that the LLM suggests increasing contrast by multiplying pixel values by a constant, while the solution without visible change used histogram normalization. Sage guesses that she needs to find a better constant, prompting again "Increase contrast of the image by scaling, try scaling in increments of 0.1". Now, the suggested images show a spectrum of contrast enhancements, out of which Sage picks the best one.

Walkthough summary. In the above scenario, Sage was given an impression of how the LLM could (mis-)understand an ambiguous prompt by viewing *multiple results* simultaneously, which would have been much harder looking only at source code suggestions, and could immediately refine her goal and revise the prompt before committing to any code change. Moreover, she later controlled the range of suggestions after having understood along which dimension there is variation (strength of contrast) in her initial prompt. A shortened version of what Sage encountered on screen is shown in Figure 2.

To better illustrate the impact on programmers' workflows, we contrast a workflow in ReFiQ to a traditional workflow in Figure 3. In a traditional workflow with LLMs, programmers have to inspect and (automatically) apply code to be able to execute the program and inspect its execution results. Depending on the outcome, they

may have to recover the previous version of the code before proceeding to the next iteration of changes. This is in contrast to the workflow in ReFiQ, where the proposed change to the program is automatically executed as soon as code generation completes. Programmers can now decide to refine their prompt if needed, before committing to the code change once they deem the result and the code as adequate.

Generalizing to other domains. The walkthrough above used an inherently visual domain to highlight wide gulfs of evaluation – images not being apparent from code – and envisioning – the goal of adjusting contrast being an abstract quality until they "know it when they see it." As the system relies on components previously explored in live programming literature, such as probes and run-time values, ReFiQ shares their strengths and weaknesses in illustrating run-time data and can easily extend to different domains. Some examples are given below:

Data analysis The state of datasets can be visualized using truncated tables, summary statistics, or plots. For example, removing outliers from a dataset can be an exploratory task requiring frequent visual inspection of a histogram, which ReFiQ supports on a visual level.

String manipulation Extracting or creating textual data requires reviewing intermediate results, for example, to check output formatting for readability or see where parsing logic captured parts of textual data incorrectly. Instead of an image, text would be previewed in the respective panels of our UI.

Interactive systems While not currently implemented, the conceptual framework extends to the presentation of any run-time object in an object-oriented language. If that object has interactive behavior, e.g., a simulation, game, or UI component, rendering alternatives would make this interaction comparable across suggestions. For example, prompting to try different implementations of collision handling in a physical simulation can result in multiple simulations running simultaneously, showing the impact of each code suggestion live, and potentially even reacting differently to user input.

3.2 Designing ReFiQ

Motivated by live programming workflows and the demonstrated effectiveness of run-time values in validating individual AI suggestions, we aimed to position ReFiQ deliberately as close to concrete run-time data as possible without losing the generality of source code. This involves multiple connected design decisions we will discuss using the framework established by Lau and Guo [16].

In their mapping of the design space of AI-based coding assistants, Lau and Guo [16] identified ten dimensions, grouped into the four categories "user interface", "input", "capabilities", and "output".

User interface and interaction anchoring. In Lau and Guo's taxonomy, ReFiQ is classified as an IDE extension with single-turn prompts and user-initiated interactions. A central detail beyond the resolution of this taxonomy, however, is where the interaction is *anchored*: Shifting from chat windows at the side to prompts inside code lets programmers point to concrete locations without circumscribing the location of interest. A typical example would be

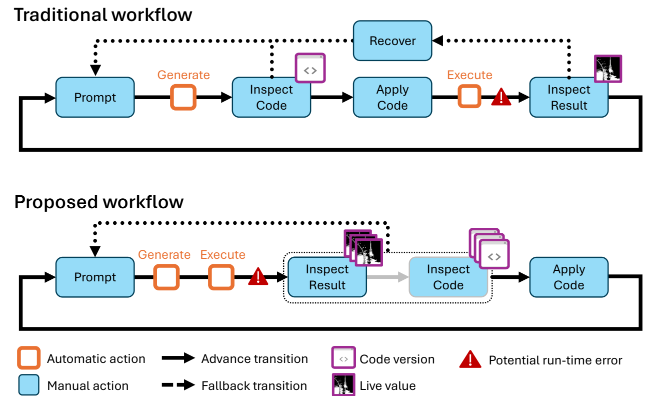

Figure 3: Traditional AI assistants like Copilot require users to manually execute generated code in order to evaluate its effectiveness. This results in additional manual work, late feedback, and potential recovery time. With ReFiQ, we propose a result-first approach, in which generated code is delivered with its output, which grants users instant feedback to their prompts. This way, users can evaluate their approach faster, allowing them to reject bad suggestions earlier. Furthermore, users don't need to undo last AI actions because they don't have to commit to a generated suggestion before they evaluated it and actively accepted it.

Copilot's in-line prompt. Our design favors interaction with run-time data. Thus, we extrapolate this shift and anchor the interaction where *data* is displayed. Using probes as one (conceptually interchangeable) mechanism to display live values, the prompt window is invoked by referring to (clicking) a live value, and specifying what the programmer *expects this value to be*, reframing the interaction as data-centered instead of code-centered. While other systems, such as LEAP, also make run-time data visible through a probe mechanism, the anchoring of the prompting interaction in, e.g., LEAP is on a source code-level, not run-time, as prompts are specified through code comments without explicit reference to run-time results.

On a technical level, the reference to concrete data allows ReFiQ to know which example data to visualize. A weakness we realized during early testing is that referencing run-time data requires an intermediate result to exist in the first place, making it easy to ask for corrective code changes but hard to ask for an entirely new processing step. ReFiQ provides two mechanisms to allow larger steps: (1.) The final return value of a function is automatically a probe, allowing programmers to work backwards and (2.) a fallback to a Copilot-style invocation outside probes, creating a probe on the fly when AI suggestions compute something new and using its output as the run-time previews.

System output. The *system output* dimensions are where ReFiQ differs most significantly from prior work: Lau and Guo [16] separate *system output* into the *output format* and *explainability* dimensions. ReFiQ generates multiple results at once and shows users results of run-time execution first, which aims to inform users about

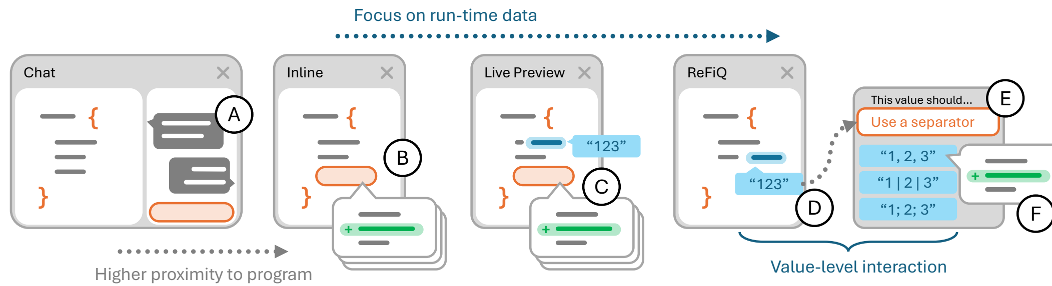

Figure 4: A dimension in the design of AI assistants is where the interaction is anchored relative to the program: (A) Chats are at a greater distance, while (B) inline prompts are close to where they act. (C) Live run-time values provide feedback as soon as suggested code is included (e.g., LEAP [8]) (D) These run-time values provide a novel anchor, at which ReFiQ provides the affordance to (E) prompt within a run-time value framing. ReFiQ remains on the level of run-time values as its primary feedback mechanism before (F) revealing the underlying code change via progressive disclosure.

ambiguity in their prompt and allows them to explicitly leverage ambiguity or language suggesting multiple solutions to generate a range of options to choose from. This is in contrast to other approaches to *explainability*—e.g., LEAP also generates multiple results but shows the code first and augments the code with the run-time results of one selected generated code option. Displaying results first encourages different usage patterns, the effects of which we analyze in our exploratory user study.

During repeated testing, we learned that juxtaposing visual outputs quickly uses up screen space and decided to present large visual representations of multiple results one after another. However, we opted to leave the visualization of the probe from which the interaction started on screen, so that the consequences of each change can be assessed in relation to the original value.

System input and capabilities. Our *system inputs* are freeform text, using the current file and recent exploration steps as context. This implementation of ReFiQ does not feature any personalization. ReFiQ's *system capabilities* are basic: it does not act autonomously beyond filtering out generated suggestions that produce an error during execution, and its system actions are limited to editing files and executing them using examples. This is comparable to other systems evaluated in this space, like LEAP.

In summary, ReFiQ occupies a unique point in the design space of AI programming assistants that extrapolates recent findings on the effectiveness of live programming mechanisms and applies them consistently to programming tasks.

4 The ReFiQ Prototype

To evaluate ReFiQ, we implemented its result-first AI assistant concept as an extension to *PyCharm*, a widely used IDE for Python.

4.1 User Interface Overview

Figure 1 shows different aspects of the interface. The user executes any global Python function via a gutter icon and supplies function parameters as needed. To receive feedback, the user selects text ranges and marks them as *probes* - through the right-click menu or a shortcut. Each probe logs its live value as inlay text on the same code line. These live values refresh only on demand, via a shortcut, to avoid distraction and preserve interactive performance. For objects with hard-to-interpret string representations (e.g., images showing up as array of numbers), the user can trigger *previews* - floating, resizable pop-ups to visualize objects like images, pandas data frames, long texts, and object trees, allowing comparing outputs of different probes in the same program version side-by-side. (Note that we decided to show AI suggestions one after another by mouse interaction to manage screen space, but each AI suggestion can still be compared to the original version of that run-time value.)

Clicking on a live value reveals a *version menu* showing a complete history of the current probe. Each entry shows both the historical value and the code state that produced it. From the same interface, the user can issue a natural-language prompt describing a desired result for the selected probe. The system calls an LLM to generate multiple alternative implementations. The number of suggestions is not predefined, but is limited to 5 by default, unless the user asks for more. Each suggestion is executed immediately; so users see the resulting probe values first. Users can inspect both past and suggested code versions via a diff view alongside the current state. Not only does it support quick code application, it also allows for prompt revision before committing to a specific suggestion.

4.2 Technical Implementation

We forked PyCharm Community 2025.1 to modify IDE behavior; most components can be compiled into a portable plugin, but we tested within a custom PyCharm build. We leverage JetBrains' *XDebugger API*[2] to interact with the *PyDev debugger*[3] at a high level. In a nutshell, we place a temporal breakpoint at each probe, which allows us to extract the underlying Python object at run-time. AI-generated code can also be executed directly through the debugger. Since XDebugger's design is language-agnostic and unified for all JetBrains IDEs, it allows for straightforward cross-language extensions in other programming languages, for example, Java (via IntelliJ).

[2] https://github.com/JetBrains/intellij-community/tree/master/platform/xdebugger-api
[3] https://github.com/fabioz/PyDev.Debugger

Although this had no implication for our user study, side effects present multiple challenges for our concept. Since in our implementation, we execute suggestions using the debugger rather than in a sandbox, code with side effects can alter the external state and lead to inconsistent or unwanted states. This could be mitigated, for example, by parallel worlds [28] or Babylonian's replacements [22], a concept to mock resources while re-executing a running example. Similarly, execution of LLM-generated code outside a sandboxed environment can expose users' systems and data to damage or exfiltration. Consequently, a proper deployment of the system should use at least one of the suggested methods for isolating side effects.

AI use and system prompt. For the AI assistant, we used OpenAI GPT4.1 through the official OpenAI API[4]. To define the output format for multiple suggestions, we included the system prompt shown in the appendix (see Appendix A). The prompt itself was designed by frequent testing and isolating different failure modes and unwanted behavior we observed, which explains the multitude of statements prohibiting the AI from taking any shortcuts or making far-reaching changes. Moreover, although the LLM cannot know the output of its code, asking it for variants that yield different outputs prevents it from generating duplicate changes where just code formatting or naming differ. Including available dependencies in the system prompt is beneficial to guide the abstractions it will use in its responses; it is currently hard-coded to evaluate numerical and image-related tasks but could be automated in a future version. In its final version, the prompt performed error-free during the study.

5 Evaluation

The goal of our study is to characterize changes in AI-assisted programming; thus, our leading research question is:

> How does providing run-time results of multiple LLM suggestions upfront change coding and prompting activities compared to a state-of-the-art LLM assistant?

We hypothesize that the result-first presentation supports faster feedback loops [23] and incremental solution discovery [9] in line with live programming as we transfer its qualities to *future changes*. As live programming improves feedback on code and has been demonstrated to extend its benefits to LLM-generated code [8], our design has the potential to shorten feedback loops for *prompts*, too, as programmers can skip code interaction.

We further expect that providing multiple results supports exploration consistent with tools that afford *concurrent (micro-)versions* [3, 14] as we now compare possible *future versions*. As these variations are not created by programmers, we hypothesize that they convey information not yet anticipated, such as variation points left ambiguous by the prompt.

5.1 Methodology

To evaluate our system's effectiveness in supporting exploratory programming, we conducted an exploratory user study with eight participants. Given the novel nature of our approach, this study aimed to gather initial evidence for our hypotheses while identifying emerging patterns and opportunities for future investigation.

[4]https://openai.com/api/

Table 1: Demographics of study participants: gender, years of experience programming, frequency of use AI use (1-never to 5-very often), AI tools used of Browser Chat, Inline Code Completion, IDE Chat, autonomous agents.

P	Gender	Exp. [Y]	AI use	AI tools used
1	m	6	5	all options
2	m	9	3	Browser Chat, IDE Chat
3	m	7	5	all options
4	f	9	4	Browser Chat, IDE Chat
5	f	6	3	Browser Chat
6	m	7	2	Browser Chat
7	m	10	3	Browser Chat, IDE Chat
8	m	12	4	all options

Study conditions. We employed a within-subjects design where each participant experienced our tool and a baseline, allowing us to gather richer comparative insights from all participants. Each participant completed two exploratory programming tasks — one using ReFiQ and one using GitHub Copilot as a baseline. The baseline condition retained non-AI features from ReFiQ, which include *probes and microversioning*, while replacing our AI component with the GitHub Copilot plugin for PyCharm[5], including inline completions and all chat functionalities, but excluding agentic mode. Both systems were configured to use OpenAI GPT-4.1 as the underlying language model. In this way, we limited the comparison to AI interaction paradigms rather than system or model capabilities, and provide a basic level of liveness through probes in both conditions to avoid any effects introduced by probes alone.

Balancing. The combination of task ordering and condition ordering was drawn at random for each participant without replacement, so that each combined task and condition order would be drawn exactly once every eight participants, making this a balanced cross-over study design controlling the expected order-dependent learning and fatigue effects and ensuring each task is solved both with Copilot and ReFiQ.

5.1.1 Study Setup and Procedure.

After initial introductions, consent procedures, and demographic survey completion, the study proceeded with two main task blocks. For each task block, participants first received an introduction to the relevant tool (either ReFiQ or GitHub Copilot) and had the opportunity to explore its features through two practice examples and ask clarifying questions for about 10 minutes. Following this familiarization phase, participants were introduced to the task requirements and began implementation with a 20-minute time limit. Participants who completed early received a bonus requirement to continue exploration and helping us gather more insights, while those exceeding the time limit were asked to stop. Each task block concluded with a post-task survey. After completing both tasks, we conducted a semi-structured interview for 10-15 minutes to explore participants' comparative experiences and preferences.

[5]Version 1.5.48-243

5.1.2 Participants. We recruited eight computer science students at the master's level, with one participant planning to start a master's study in less than 3 months. Prior to the main study, we conducted a pilot study with two co-authors not yet familiar with the tool and concrete tasks to refine our protocol, survey instruments, and data collection procedures. Participants received compensation for the 90-minute session in line with regulations and best practices at our institution independently on whether they completed their participation. Demographics and self-reported experience with LLM-supported tools are shown in Table 1.

5.1.3 Tasks. Both tasks were designed to encourage exploration while maintaining clear success criteria:

OCR This task required preprocessing scanned documents to improve OCR accuracy on a degraded image. The sample image chosen for the task contained five lines of text from an invoice header with artificially introduced noise and rotation. The OCR algorithm itself was provided as a black box.

Topography This task involved segmenting an artificially generated geographic image to distinguish rivers from lakes. Participants should highlight the two types of water bodies differently, working around unrelated geographical features.

These tasks were selected to require different image processing techniques (denoising/straightening vs. segmentation), ensuring that they are methodologically independent, while remaining accessible to participants without specialized domain knowledge. The difficulty of each task (e.g., type and magnitude of image degradation) has been adjusted through a pilot study.

5.1.4 Measurements. We employed a mixed-methods approach to capture both quantitative metrics and qualitative insights:

Surveys. The demographic survey assessed participants' educational background, programming experience, familiarity with AI coding tools, image processing expertise, and general exploration preferences. Post-task surveys measured perceived task characteristics (clarity, creativity, difficulty), cognitive load using NASA-TLX[11], system usability[7], and programming-specific dimensions including perceived efficiency, control, confidence, exploration support, and code understanding (see Appendix B for their exact wording).

As our intra-subject design generates paired data, we used the Wilcoxon signed-rank test with Bonferroni correction for testing all survey questions individually for a significant difference between ReFiQ and our baseline.

Recordings. Throughout both coding tasks, participants were encouraged to think aloud to verbalize their thought processes. With participants' consent, we recorded all coding sessions, capturing screen activity and think-aloud verbalizations, which were subsequently transcribed and anonymized for analysis. Additionally, the semi-structured interview protocol explored participants' comparative experiences between tools and contextual preferences for each approach. All recordings were transcribed and combined with prompts for thematic analysis using open coding with subsequent thematic clustering.

IDE telemetry. We instrumented our IDE setup with comprehensive logging infrastructure to track all interactions with our system,

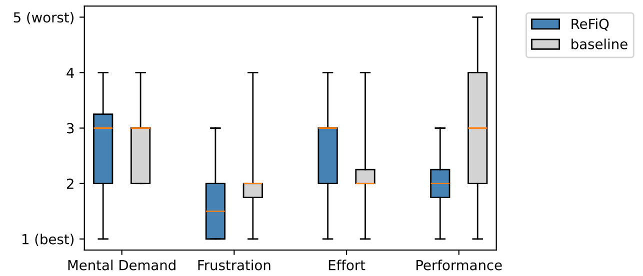

Figure 5: Mental load self-assessment using TLX questions. Lower is better.

including probe creation, version navigation, AI prompts, and suggestion generation. The resulting data allows derived quantitative measurements, such as computing frequencies and durations of AI and code interactions, and qualitative insights by augmenting the thematic analysis with activity timelines, individual prompts issued by participants, and the results they encountered. Although correctness is not a focus of this study, we used the recordings to reconstruct the results each participant achieved in each task and classified them according to their completeness and quality.

5.1.5 Threats to validity. As both tasks in our study involved image processing, our findings might have limited validity outside visual domains. Regarding data collection, Copilot sometimes does non-prompted code completion, which creates LLM suggestions that we cannot map back to a concrete prompt. The low number of participants (8) will affect statistical tests on survey responses, which is due to the exploratory focus. Participants' experience with LLM-supported tools was also mixed. Due to the limited time participants had with our tool, our findings are only an initial capture of their workflows. As we performed an exploratory, post-hoc analysis of the data, statistically significant results only indicate a potentially interesting effect that should be reproduced in a more controlled setting.

6 Results

In this section, we first report the data gathered in our survey, then results of a post-hoc analysis of gathered interaction data, and finally results from our thematic analysis over observations of use, think-aloud statements, and post-experiment interviews.

6.1 Survey

Mental Load. As we assessed the mental load in our experiments via the corresponding NASA TLX questions, we did not measure any statistically significant difference explained by using ReFiQ over our baseline (see Figure 5). All tasks were moderately demanding with relatively low frustration, and the introduction of ReFiQ most likely did not introduce notable mental overhead.

Usability. The usability of ReFiQ was not rated significantly different than our baseline (see Figure 6). Both tools are already reasonably well integrated and easy to learn and use. A tendency to use ReFiQ more frequently is visible, but not statistically significant.

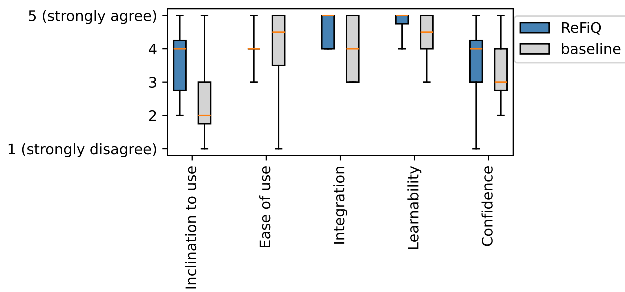

Figure 6: Self-reported usability scores based on five SUS questions. A Likert scale was used. Higher is better.

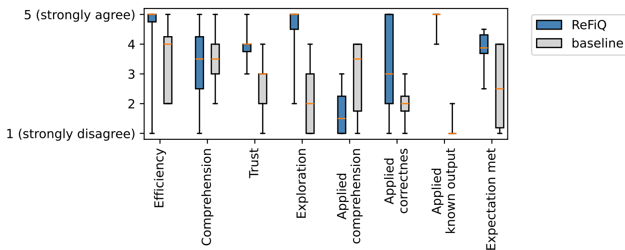

Figure 7: Self-reported dimensions related to the programming activity and workflow. "Applied" questions rated the participant's confidence in code specifically in the moment programmers applied a code change. A Likert scale was used except for "Expectation met", which participants could answer in percent. More information about the questionnaire is available in Appendix B.

Workflow-related questions. In Figure 7, we report how participants perceived their programming workflow. We need to emphasize that neither code comprehension nor correctness is a focus of this study, but tendencies reported here appear to be consistent with our thematic analysis in subsection 6.4.

In general, participants perceived a slight improvement in programming efficiency in ReFiQ, about the same degree of code comprehension, and slightly higher trust in generated code. As expected, ReFiQ allowed for better exploration of the solution space.

When asked whether they understood the code specifically *before applying a suggestion*, ReFiQ tends to score lower. However, participants were slightly more confident about the correctness and consistently more confident about knowing the output of the suggestion before applying it. In most cases, the code eventually accepted from ReFiQ met the participants' expectations, whereas GitHub Copilot showed mixed results.

In summary, survey responses suggest a tendency that ReFiQ might negatively impact self-reported code comprehension in general, but participants had more confidence that the code suggestions generated the output they expected. This is expected in a design that progressively discloses code only after participants have already seen run-time values and possibly made a decision which effect they want the LLM to achieve. This trade-off will be explored in detail in our thematic analysis.

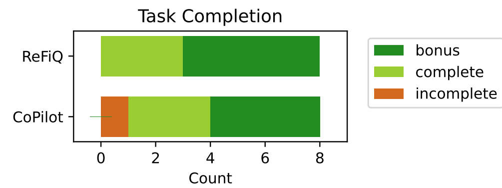

Figure 8: Task completion across experimental conditions. There is no significant difference between groups.

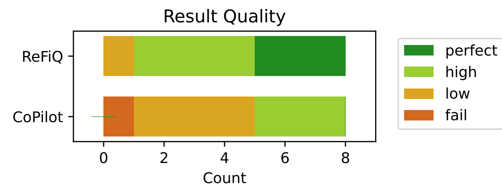

Figure 9: Output quality of the final program across experimental conditions. ReFiQ users created programs making fewer classification and OCR errors.

6.2 Task Completion and Correctness

We analyzed the solutions for each task regarding the degree of task completion and the correctness of the output created by each program. Since the tasks were designed to motivate open-ended exploration and elicit ambiguity when put in a prompt literally, there is no unambiguous test to determine the correctness of the solutions. We therefore scored the progress made by participants along two axes: Completion (within the time limit) and result quality, with their criteria described below.

Completion levels:

- **Bonus (best)** *Topography:* Rivers and lakes are marked visually distinguishable, and the image contains a legend describing which highlight corresponds to which water body type. *OCR:* Detected text rendered next to the original.
- **Complete** *Topography:* Rivers and lakes are marked visually distinguishable. *OCR:* Detected text could be obtained as a string after image processing.
- **Incomplete** Completeness criteria not fulfilled.

Quality levels:

- **Perfect quality (best)** *Topography:* Visual highlights indicate no misclassifications, *OCR:* Detected text corresponds to the original (excluding spacing and punctuation).
- **High quality** *Topography:* Not more than two misclassifications (e.g., missed lakes, islands in river marked as river) *OCR:* Not more than three alphanumeric characters missing or misclassified.
- **Low quality** *Topography:* Visible highlights but more than two errors *OCR:* Text detected but more than three alphanumeric errors.
- **Fail** *OCR:* Not recognized any text (The topography task had no complete failures)

Table 2: User interaction statistics determined in post-hoc log analysis. An iteration is the sequence of user interactions until saving the file and updating run-time values in probes.

	ReFiQ	Baseline	Ratio
Code iterations	159	255	62%
— applying LLM suggestion	71	75	95%
— manual editing	88	180	49%
Median iteration duration	30.1s	21.2s	142%
— applying LLM suggestion	89.6s	57.3s	156%
— manual editing	8.2s	12.4s	66%
Prompts	70	64	109%
Acceptance rate	61%	85%	0.72

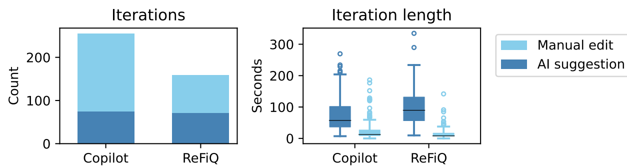

Figure 10: Count and length of participants' iterations. While the overall number of iterations needed to complete the tasks is lower with ReFiQ, this difference is entirely explained by fewer manual revisions. At the same time, ReFiQ displays longer iteration cycles, as more time is spent refining the prompt.

While we observed no significant ($p = 0.55$) difference between conditions in task completion rate (see Figure 8), one participant did not complete a baseline OCR task within the time limit. Comparing the quality of the generated output, we observed significant ($p = 0.022$) differences in errors. Programs created with ReFiQ provided more accurate results across both tasks (see Figure 9). Since this is a post-hoc analysis on a small sample size, this result should be interpreted as an early indication demanding further investigation in future work.

6.3 IDE Interactions

Post-hoc analysis of interaction logs revealed a notable difference in iteration behavior between the two conditions. For this analysis, we defined an iteration as the sequence of interactions until participants saved the code and triggered an execution of their program, which automatically updates run-time values in probes.

While we initially hypothesized that ReFiQ makes it easier to iterate, we observed that **participants went through fewer code iterations** (about one third) while solving a task despite issuing a similar number of prompts.

As we break down the types of iterations based on what caused the code change – applying an LLM suggestion or editing manually – we observe that this difference is almost entirely caused by manual edits. Additionally, participants spent more time prompting and eventually applying the LLM suggestion in ReFiQ, but more time editing in Copilot. This suggests that *Copilot required more manual intervention*, whereas **ReFiQ allowed participants to take fewer, but more effective steps** at the code level.

This is consistent with the lower acceptance rate of code suggestions (61% in ReFiQ vs. 85% in Copilot), as the result-first presentation made it easier to reject suggestions that did not meet expectations, thereby avoiding the need for some manual corrections.

IDE log statistics are summarized in Table 2 and a direct comparison of iterations and the distribution of their durations can be seen in Figure 10.

6.4 Thematic Analysis

We identified three thematic areas our participants were concerned with: (1.) the prompts themselves, (2.) strategies to validate suggestions, and (3.) the tools' user experience. Note that some prompts and most transcripts were in the participants' native language and translated to match their wording as closely as possible.

6.4.1 Prompts. We identified two prompting strategies mostly unique to ReFiQ: *Prompting for variation* to specifically leverage multiple results, and referring to *run-time values in prompts*. We also discovered two distinct modes of improving prompts: *in-place revision* before any code was accepted, and *follow-up correction* to materialize an exploration state in code and continue with a fresh prompt with the intent to replace the temporary code. However, *backtracking* (undo and redo) across prompts and code changes was not well supported in both conditions.

Variation. A key difference between ReFiQ and our baseline was that participants intentionally **prompt for variation**. For example, to reduce noise in the OCR task, P8 used "try different image kernels" while P1 used "try different thresholds" at the end of their prompts. Some went as specific as "can you give me in steps of 2 from 110 to 150" (P4) when adjusting a threshold in the topography task, attempting to use the tool to create a spectrum of previews. Variation was almost always introduced as a refinement of an initial prompt where participants noticed that a solution included a variation point they needed to fine-tune. Exceptions are high-level prompts that ask for diverse solutions rather than fine-tuning, e.g., "I want to highlight blue parts of the picture in some way. Can you give 5 possible ways" as the first prompt by P4. In total, at least five participants used this strategy at least once with ReFiQ, while we found no occurrence when using Copilot.

In-place Revision. The way participants **incrementally revise the same prompt** changed with ReFiQ. A general pattern was that a previous prompt was slightly modified and reissued after discarding previous suggestions, either **rewording** their prompt or appending **more requirements**.

For example, P6 prompted Copilot during the OCR task "can you remove the small dots in the img?" ("img" being a variable name) followed by "can you remove all connected areas of black pixels that are below average.", after seeing that the first prompt yielded an erosion filter that damaged text. P1 issued the prompt "I want processed to filter out background noise by detecting big connected pieces and filtering out all under some threshold. Try different thresholds" (again, "processed" referring to a variable),

and re-prompted two more times, adding "The content I want to keep is black text on a white background.", followed by "Binarize it first" to the growing prompt.

While participants used in-place refinement across both conditions, they refined their prompt in a much tighter feedback loop, looking directly at results in ReFiQ and rarely applying the code change until their final revision. In support of this observation, P4 mentioned the fact that they could see what code does prior to committing to it as helpful to "try different things exploratively and understand afterwards what it does".

Follow-up Correction. A second type of refinement happened in **corrective follow-up prompts**, where participants committed to a *good enough* solution and referenced a deficiency and what they expected instead in the next prompt. For example, P6 prompted in the topography task "i want to replace all blue in edited with a bright red.", discovered that the applied suggestion includes colors with a high blue channel (e.g., magenta) and re-prompted "threshold the blue by HSV instead of BGR" to target only shades of blue-ish colors in another color space. Users also fell back to this strategy when their in-place revised prompt became complex as a way to materialize their current understanding in code and resume with a fresh prompt. In fact, corrections occurred almost immediately in Copilot when the first result was deemed unsatisfactory, while ReFiQ users varied in their strategies, often finding some close match in the results and continuing from there, or explicitly asking for variation in their revised prompt. Notably, Copilot responded with entirely different solutions to a corrective prompt despite being close to an acceptable result.

Run-time Values as Subjects. When using ReFiQ, we observed the tendency to **refer to run-time values** in their prompt. For example, P8 revised one of their OCR prompts to include "try to replicate the line breaks from text exactly" after comparing the run-time content of the string variable `text` with the text rendered in the image. P4 prompted in the topography task that "There are two blue circles wrong." and proceeded to describe their location in detail to the LLM.

Backtracking. Across both tools, participants related negatively to their **workflow for revising prompts**. For example, P1 mentioned during the OCR task "I want the prompt back where I referred to the lines [of text]", while P2 mentioned in the interview, that "you cannot simply replace [the Copilot code] with a new prompt at the same place", clarifying that they envisioned "ctrl+z-ing" their way back to the prompt. However, participants noted that with ReFiQ, they could at least revise the prompt after seeing results before applying a change: P2 said that it felt "safer" to prompt, knowing they could still change it before committing to an approach. This suggests that some users of inline LLM assistants perceive prompting as first-class edit action that deserves the same treatment as code edits regarding undo/redo operations.

6.4.2 Validation.

Run-time Consequences of Prompts. Participants positively commented on several different ways in which viewing run-time results helped them. The **immediacy of feedback** was perceived as inviting iteration: P7 mentioned that ReFiQ's "feedback loop was very fast" and P2 said "I see immediately when nothing will come if it. Then I specified in greater detail and viewed it again". The **concreteness** of the results and their closeness to the domain aided comprehension; P7 acknowledged this stating that they are "not good at translating HSV colors in their head". Results also served as **progress indication**, for example, P7 highlighted the usefulness when task completion is not a binary decision: "Now I found three of five points, that's progress since previously I had only one". Users found it useful to see how already known run-time values change in response to a prompt: "What was very, very practical was that I could view the image before and after" (P7), referring to the feature that allows for comparing new run-time results to those of the current version. Live results helped **exploration**. For example, P8 compared them to their known workflow, stating "In Cursor I could also say 'make five suggestions', but if I want to view all five results, then I have to accept one and have a look at it, revert, then accept the other" Overall **trust** in a solution increased. P6 confirmed they had "more trust since one has immediately seen what it does", while P3 noted that they "didn't need to look into code", suggesting a certain level of *blind* trust.

Multiple Results. The presentation of concurrent responses to the prompt helped participants **resolve ambiguity**. For example, P7 requested a "greyscale" image in one prompt, then noted that "it was apparent that this was greyscale, that was only black and white". Several participants felt the effects of making a **forced decision**. P8 said that "I must select an option intentionally; then I also looked somewhat into the code and not just the result", indicating a multifaceted decision process. P2 noted that "If you waited longer and the prompt was too unspecific, the tool basically forced you to look at multiple possibilities".

Explanations and Generalization. Participants gave mixed feedback on how well ReFiQ explains suggestions. While live feedback was highlighted as positive (see Run-time Consequences of Prompts), several participants mentioned a **lack of comments** compared to other LLM-based tools. P3 said "I'd like that it shortly hints at what it did and what its reasoning was because I'd read that", while P2 directly asked in a prompt "what does cv2.inRange do". P4 mentioned that "If one doesn't have an idea what to do, I'd find it easier to interact with a chat" to obtain an initial plan in natural language. P4 was also missing "comments to match [what happens] in a structured way" and doubted the **generalizability** of their concrete code.

In general, we observed the need to acquire **domain knowledge** to complement the concrete run-time results, which ReFiQ was not yet designed to support.

Probes. Since ReFiQ's interaction is centered around probes and their run-time values, it allowed participants to set a **precise scope** in which the LLM acts and work **incrementally**. P8 mentioned that "I'm setting a strict scope. [...] Here is my probe, and it's about this probe and not so much about 'here is a [code] file that changes this and that' ". P2 said "it felt more like programming", working their way forward from a specific variable. However, participants

mostly used probe-less prompts[6] to specify their next step rather than a new requirement for an existing value.

6.4.3 User Experience. Regarding the user experience of ReFiQ, participants criticized that the **lifetime of different UI elements** was unclear, e.g., P7 contrasted the pop-up windows with browser tabs, where the latter is reasonably stable while the small windows feel ephemeral enough to cause anxiety that their state might get lost ("I can't ponder calmly"). Chat history was frequently highlighted as another example that persisted context long enough and was readily available. Copilot had its own issues, however, as its ghost text was even more short-lived ("Either I save it or it is lost"). A minor aspect mentioned by one individual participant (P2) was that the background execution of LLM suggestions introduced notable **latency**, whereas Copilot suggestions felt almost instant to them.

7 Discussion and Outlook

In this section, we discuss our results and derive areas for potential future work.

Prompts as First-Class Programming Artifact. Revisiting our initial hypotheses, we could demonstrate that a result-first approach impacts feedback loops, but in a more nuanced way than expected: Programmers iterated less often in terms of code changes, but more often on prompts, covering more ground with each single LLM-assisted change. This highlights the importance of the prompt as *materialization of programmers' current domain and problem understanding* and the need for similar first-class tool support previously afforded to coding activities, such as live programming.

We need to emphasize that participants had access to live programming features (probes) across both conditions, which indicates the effects were actually due to the result-first approach and not an effect of introducing live programming.

Multiple Suggestions. LEAP [8] demonstrated that live feedback supports programmers in evaluating individual LLM suggestions. ReFiQ extrapolates this direction by skipping code as the artifact to be selected and offering direct presentation of multiple run-time behaviors.

It became clear that result-first presentation and the multiple-suggestion feature are synergistic, as programmers can more easily perceive differences between suggestions in terms of different run-time effects. Key properties of this synergy are that it invited intentional experimentation and fine-tuning already at the prompt level, highlighted dimensions of variability, and conveyed multiple ways in which ambiguity can be understood.

Configuring ReFiQ to control what is shown first (code or result) could help isolate the effects of showing each first in a follow-up study and help understand the differences in the decision process when choosing between concrete results and code suggestions.

Consequently, we can consider this approach as one step toward bridging the *gulf of envisioning* by demanding programmers to intentionally select *what they meant exactly*. At the same time, the design can mitigate some of the non-determinism underlying LLMs as the systematic absence of a good suggestion is more likely due to an inadequate prompt than chance.

There are opportunities to improve the presentation of several suggestions. To use screen space efficiently, ReFiQ resorts to hiding visualizations behind mouse-hover, allowing comparison in a one-after-another fashion while side-by-side presentation like in Explorians [3] is sometimes more useful. As this requires careful consideration of how to use screen real estate, domain-specific overlays and visualizations of variability can be considered, such as those in Boba [19].

Microversioning and Backtracking. Across both conditions we studied, we observed the need to backtrack immediately or after reaching a dead end. ReFiQ does support fine-grained microversioning at the run-time-value level and was used by some participants to locate a specific state or parameter setting from a few iterations earlier. Other participants, however, appeared to prefer returning to a whole editing state, including code version and currently edited prompt, as easily as "ctrl+z" rather than identifying a previous version by searching through run-time history and losing prompts. The latter can also be explained by unfamiliarity with microversioning and its tooling in ReFiQ.

For future work, we suggest a design that includes prompting not as a separate tool through which programmers indirectly change their program, but as an integral part of the code editor, supporting undo/redo, copy and paste, and versioning. As much as incomplete code represented a working state for programmers before, code and in-progress prompt together represent an intermediate working state now.

Trade-offs in Code Comprehension. While forcing programmers to choose among multiple LLM suggestions leads to more engagement with differences among solutions and the result-first presentation increases trust in the LLM suggestion for the task at hand, code comprehension and confidence that the solution generalizes to other inputs ultimately suffer. This finding was consistent across self-reported code comprehension scores in the survey and suggestions by participants that they would benefit from explanation.

This finding also relates to LEAP's study [8] where participants that did not have access to live programming had a tendency to show under- and over-trust in generated results. In their study, some participants also stated that live programming appears helpful to assess multiple suggestions quickly. Our study included live programming in both conditions but compared the effect of showing multiple run-time values first, before code. Our findings thus suggest that placing even more emphasis on the run-time values than in LEAP presents a trade-off between comprehension and quality for the specific task.

To provide a better trade-off, designs like Ivie [30] that provide *in-situ explanations* as programmers engage with LLM-suggested code are important building blocks. From a live programming perspective, using *multiple examples* to illustrate how changes impact edge cases or other requirements has proven useful [3, 22] and motivates a design in which each LLM suggestion is explained in terms of its run-time effects in diverse scenarios.

Consequences for Live Programming. Live programming itself is taking a new shape in the presence of LLM-generated code, as

[6] A shortcut that does not require a probe in existing code but creates one on the last variable introduced by the LLM to preview its results

its core qualities of providing tangible and immediate run-time feedback are now extending from code changes to higher-level natural-language interactions, such as prompts and chats. Tanimoto [26] envisioned the future of live programming as allowing programmers to select from running predicted behaviors. By being able to study programmers using such capabilities now, we were able to better characterize the trade-offs involved in one particular design. The preview of run-time values is not limited to passively displaying data and visuals, but can extend to interactive scenarios, as demonstrated by Exploriants [3]. Integrated with our design, prompts like "Vary gravity in steps of 0.1" could preview several simultaneously playable game levels, allowing users to select the suggestion with the best experience.

Conclusion

As the use of code-generating LLMs to create and edit programs becomes more widespread, the focus shifts from writing code to writing effective prompts. In this work, we reiterated the importance of supporting this novel activity through tools and presented ReFiQ, a live programming environment with an LLM assistant whose design prioritizes run-time results over code and multiple results over a single suggestion. Our tool helped practitioners explore the solution space at the prompt level and arrive at more intentional prompts by providing concrete feedback and encouraging them to choose between several interpretations of their prompt. Although our users needed fewer steps and fewer manual corrections to solve a task compared to a state-of-the-art LLM assistant, our study also highlighted the risk of trading deeper code comprehension for a more surface-level progress.

Acknowledgments

We are grateful for the anonymous reviewers' helpful feedback. This work was supported by SAP and the HPI–MIT "Designing for Sustainability" research program[7].

[7] https://hpi.de/en/research/cooperations-partners/research-program-designing-for-sustainability.html

References

[1] Anthropic. 2025. Claude Code. https://www.anthropic.com/claude-code.

[2] Anysphere. 2023. Cursor. https://cursor.com.

[3] Tom Beckmann, Joana Bergsiek, Eva Krebs, Toni Mattis, Stefan Ramson, Martin C. Rinard, and Robert Hirschfeld. 2025. Probing the Design Space: Parallel Versions for Exploratory Programming. *The Art, Science, and Engineering of Programming* 10, 1 (Feb. 2025), 5:1–5:33. doi:10.22152/programming-journal.org/2025/10/5

[4] Mary Beth Kery and Brad A. Myers. 2017. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 25–29. doi:10.1109/vlhcc.2017.8103446

[5] Christian Bird, Search about this author, Denae Ford, Search about this author, Thomas Zimmermann, Search about this author, Nicole Forsgren, Search about this author, Eirini Kalliamvakou, Search about this author, Travis Lowdermilk, Search about this author, Idan Gazit, and Search about this author. 2022. Taking Flight with Copilot. *ACM Queue* 20, 6 (Dec. 2022), 35–57. doi:10.1145/3582083

[6] Gilad Bracha. 2021. Enhancing Liveness with Exemplars in the Newspeak IDE. https://newspeaklanguage.org/pubs/newspeak-exemplars.pdf.

[7] John Brooke. 1996. *SUS – a quick and dirty usability scale.* CRC Press, 189–194.

[8] 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 2024 CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for Computing Machinery, New York, NY, USA, 1–8. doi:10.1145/3613904.3642495

[9] 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 (UIST '20)*. Association for Computing Machinery, New York, NY, USA, 614–626. doi:10.1145/3379337.3415869

[10] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley Longman Publishing Co., Inc., USA.

[11] Sandra G. Hart and Lowell E. Staveland. 1988. Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research. In *Human Mental Workload*, Peter A. Hancock and Najmedin Meshkati (Eds.). Advances in Psychology, Vol. 52. North-Holland, 139–183. doi:10.1016/S0166-4115(08)62386-9

[12] Brian Hempel and Ravi Chugh. 2022. Maniposynth: Bimodal Tangible Functional Programming. In *36th European Conference on Object-Oriented Programming (ECOOP 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 16:1–16:29. doi:10.4230/LIPIcs.ECOOP.2022.16

[13] Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. 1985. Direct manipulation interfaces. *Hum.-Comput. Interact.* 1, 4 (Dec. 1985), 311–338. doi:10.1207/s15327051hci0104_2

[14] Mary Beth Kery, Amber Horvath, and Brad Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. Association for Computing Machinery, New York, NY, USA, 1265–1276. doi:10.1145/3025453.3025626

[15] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, Carol Willing, and Jupyter development team. 2016. Jupyter Notebooks - a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, Fernando Loizides and Birgit Scmidt (Eds.). IOS Press, Netherlands, 87–90. doi:10.3233/978-1-61499-649-1-87

[16] Sam Lau and Philip J. Guo. 2025. The Design Space of LLM-Based AI Coding Assistants: An Analysis of 90 Systems in Academia and Industry . In *2025 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE Computer Society, Los Alamitos, CA, USA, 300–313. doi:10.1109/VL-HCC65237.2025.00041

[17] 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 (CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–7. doi:10.1145/3313831.3376494

[18] Michael Xieyang Liu, Advait Sarkar, Carina Negreanu, Benjamin Zorn, Jack Williams, Neil Toronto, and Andrew D. Gordon. 2023. "What It Wants Me To Say": Bridging the Abstraction Gap Between End-User Programmers and Code-Generating Large Language Models. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 598, 31 pages. doi:10.1145/3544548.3580817

[19] Yang Liu, Alex Kale, Tim Althoff, and Jeffrey Heer. 2021. Boba: Authoring and Visualizing Multiverse Analyses. *IEEE Transactions on Visualization and Computer Graphics* 27, 2 (Feb. 2021), 1753–1763. doi:10.1109/TVCG.2020.3028985

[20] Toni Mattis, Patrick Rein, and Robert Hirschfeld. 2017. Edit Transactions: Dynamically Scoped Change Sets for Controlled Updates in Live Programming. *The Art, Science, and Engineering of Programming* 1, 2 (April 2017), Article 13. doi:10.22152/programming-journal.org/2017/1/13

[21] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. 2015. User Interaction Models for Disambiguation in Programming by Example. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology (UIST '15)*. Association for Computing Machinery, New York, NY, USA, 291–301. doi:10.1145/2807442.2807459

[22] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-Style Programming. *The Art, Science, and Engineering of Programming* 3, 3 (Feb. 2019), 9:1–9:39. doi:10.22152/programming-journal.org/2019/3/9

[23] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2018. Exploratory and Live, Programming and Coding. *The Art, Science, and Engineering of Programming* 3, 1 (July 2018), 1:1–1:33. doi:10.22152/programming-journal.org/2019/3/1

[24] Bastian Steinert, Damien Cassou, and Robert Hirschfeld. 2012. CoExist: Overcoming Aversion to Change. In *Proceedings of the 8th Symposium on Dynamic Languages* (Tucson, Arizona, USA) *(DLS '12)*. Association for Computing Machinery, New York, NY, USA, 107–118. doi:10.1145/2384577.2384591

[25] Hari Subramonyam, Roy Pea, Christopher Pondoc, Maneesh Agrawala, and Colleen Seifert. 2024. Bridging the Gulf of Envisioning: Cognitive Challenges in Prompt Based Interactions with LLMs. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for Computing Machinery, New York, NY, USA, 1–19. doi:10.1145/3613904.3642754

[26] Steven L. Tanimoto. 2013. A Perspective on the Evolution of Live Programming. In *Proceedings of the 1st International Workshop on Live Programming (LIVE '13)*. IEEE Press, Piscataway, NJ, USA, 31–34.

[27] Jason Trenouth. 1991. A Survey of Exploratory Software Development. *Comput. J.* 34, 2 (01 1991), 153–163. doi:10.1093/comjnl/34.2.153
[28] Alessandro Warth, Yoshiki Ohshima, Ted Kaehler, and Alan Kay. 2011. Worlds: Controlling the Scope of Side Effects. In *ECOOP 2011 – Object-Oriented Programming*, Mira Mezini (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 179–203.
[29] Nathaniel Weinman, Steven M. Drucker, Titus Barik, and Robert DeLine. 2021. Fork It: Supporting Stateful Alternatives in Computational Notebooks. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3411764.3445527
[30] Litao Yan, Alyssa Hwang, Zhiyuan Wu, and Andrew Head. 2024. Ivie: Lightweight Anchored Explanations of Just-Generated Code. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems (CHI '24)*. Association for Computing Machinery, New York, NY, USA, 1–15. doi:10.1145/3613904.3642239
[31] Ryan Yen, Jian Zhao, and Daniel Vogel. 2025. Code Shaping: Iterative Code Editing with Free-form AI-Interpreted Sketching. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*. Association for Computing Machinery, New York, NY, USA, 1–17. doi:10.1145/3706598.3713822

A System Prompt

```
You are an expert Python 3 refactoring assistant.

### CONSTRAINTS
- Output **only** valid Python 3 code; *do not* wrap it
    in Markdown fences.
- Do **not** rename the function or any existing
    identifiers.
- Write only full lines of code, *do not* write partial
    lines or fragments. Don't concatenate lines with
    `;`.
- If the task can be solved in multiple ways return
    multiple rewrites of the function separated by **\
    $RESPONSE\_SEPARATOR**; otherwise return exactly
    one. 5 variants max.
- Provide multiple variants **only** when they yield
    different outputs.
- The <add-code-here> tag is a hint where to add code.
    You may edit code before or after if needed.
- Make code changes as minimal as possible to achieve
    the objective. Don't change the code structure or
    logic unless necessary.
- The following imports are already available but
    optional:
  import numpy as np
  import cv2

### OUTPUT FORMAT
<function 1>
<next>
<function 2>
<next>
and so on ...
```

B Survey

The reported Likert scale results were based on the following questions. Bold text refers to the labels in Figure 7, non-bold text was shown to participants (translated into English):

Efficiency The AI assistant helped me complete the task efficiently.
Comprehension The AI suggestions were easy to understand.
Trust I trusted the AI suggestions.
Exploration This tool allowed me to explore multiple solution ideas.
Applied When I applied AI generated code…
- **Comprehension** I understood the code before.
- **Correctness** I was sure about its correctness.
- **Known output** I knew exactly what its output would be.

Expectation met How often did the AI-generated code meet your expectations? (in %)