Answering Developer Questions with Annotated Agent-Discovered Program Traces

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

Jeffrey Tao
jefftao@seas.upenn.edu
University of Pennsylvania
Philadelphia, PA, USA

Lydia Chilton
chilton@cs.columbia.edu
Columbia University
New York, NY, USA

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

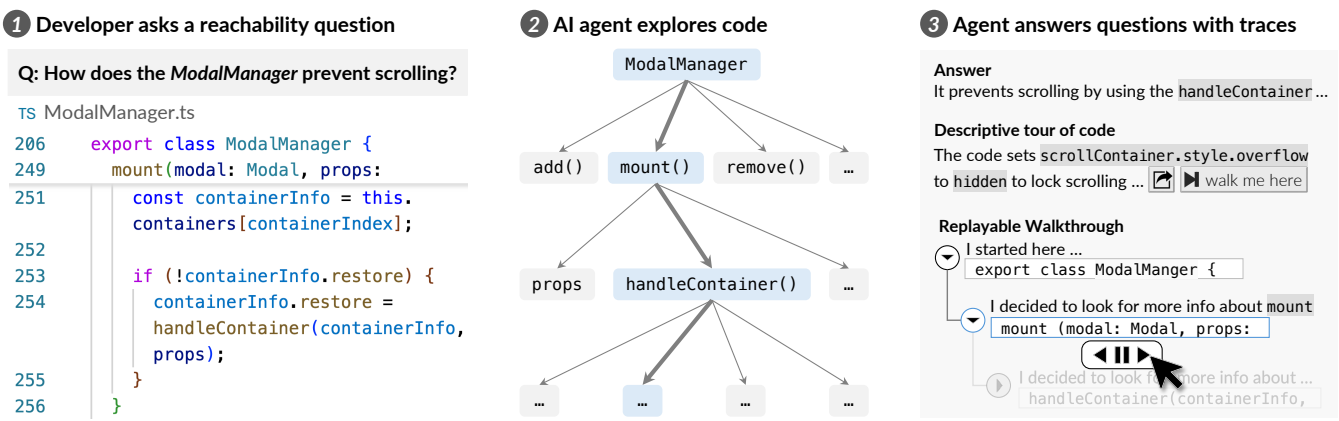

Figure 1: Overview of Trailblazer. Trailblazer helps developers answer questions that cut across a code base (❶). It explores the code with an agent that performs incremental static analysis over the source code (❷). The tool presents answers as a combination of a concise natural language summary, a descriptive tour of key discovered snippets, and replayable annotated walkthroughs (❸). By combining flow analysis and AI, Trailblazer accumulates answers as annotated, replayable program traces that help developers understand not just what the answer is, but how it can be reached.

Abstract

Developers often find themselves asking questions that cut across a code base. Answering these questions requires gathering relevant facts and tracing flow through the program. Yet today's tools offer limited support for answering these questions. Developers can either use imprecise AI tools that ignore flow or flow-tracing tools that impose a great number of choices. In this paper, we introduce a new kind of tool that answers questions better by bringing together elements of both AI and flow. We instantiate this idea in Trailblazer, a system underpinned by an AI agent that simulates an information forager, iteratively tracing program dependencies in search of answers. Then, Trailblazer packages information it found into an answer digest, which includes interactive, annotated traces of exploration. These traces can be stepped through to help developers orient to the code and find where the answer is distributed within it. In a lab study, Trailblazer helped participants answer questions more efficiently and gain greater familiarity with program flow than an AI question answering baseline. This shows how AI agents can leverage program flow to bring additional structure and clarity to its answers.

UIST '25, Busan, Republic of Korea

CCS Concepts

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

Keywords

programming assistants, reachability questions, information foraging, program traces

ACM Reference Format:
Litao Yan, Jeffrey Tao, Lydia Chilton, and Andrew Head. 2025. Answering Developer Questions with Annotated Agent-Discovered Program Traces. In *The 38th Annual ACM Symposium on User Interface Software and Technology (UIST '25), September 28-October 1, 2025, Busan, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3746059.3747652

1 Introduction

A central challenge in understanding modern code bases is answering reachability questions [26, 27], which ask how control or data flows across a program. For example, a developer might ask of a UI components library, *"How does this code disable scrolling of the window when a modal dialog is showing?"* Answering these questions requires developers to explore code paths from where a behavior begins to its outcomes and effects. These questions are both common and challenging to answer [26, 27].

One reason reachability questions are so difficult to answer is the complex structure of real-world programs. Nested function calls, callbacks, event handlers, and dynamic dispatch add indirection that make it hard to see how a behavior is implemented at a glance.

Answering such questions is therefore characterized by backtracking, keeping track of multiple paths, and constant judgments about which paths to follow [26]. For situations like these, IDEs provide tools for tracing flow, from primitives like "Go to Definition" to tools that afford stepping through the data flow [1]. Yet more advanced affordances have been developed in research tools [3, 28, 44]. While these tools accelerate tracing code flow, they require a great deal of judgment from the user for search to progress.

More recently, AI-based tools (e.g., [10, 17]) have been integrated into IDEs to provide question answering support. These tools reduce constant judgment calls to wording a question and reviewing an answer. They supply answers in the form of readable text with snippets from the code. However, the current implementations of these tools may not guarantee that the answers actually reflect the code, nor do they help paint an adequate picture of the flow of the program as it pertains to the user's question.

Taken together, these shortcomings suggest the need for a new kind of tool that blends the precision of flow analysis with AI's ability to decide and distill (Figure 1). In this paper, we develop Trailblazer, an intelligent interface for the IDE that answers developer questions with annotated, replayable, agent-discovered program traces. Trailblazer works by simulating the behavior of an information forager, using tools the developer has in the IDE to trace flow (e.g., "go to definition" and others). In doing so, it discovers program traces that answer questions. The innovation of Trailblazer is in its presentation of results. Answers are shown as replayable, annotated step-by-step traces. Developers replay this trace to see how an answer was found and to understand the structure of the code that connects the trace together. The entry point to these traces is a program "tour," or a set of waypoints from the agent's exploration that together work to answer the question.

To understand Trailblazer's effect on question answering, we conducted a within-subjects lab study. 20 participants answered reachability questions using either Trailblazer or a modern question-answering programming assistant. When reviewing answers given by Trailblazer, participants completed search tasks more efficiently. They demonstrated better recall of the program flow essential to the answer in a surprise code ordering task. These findings suggest the value of "trailblazing" agents that discover answers by crawling complex search graphs and demonstrating their findings in familiar representations of search to the user.

This paper makes the following contributions:

- A formative study that clarifies obstacles and information needs encountered with typical code analysis IDE tools.
- Trailblazer, a tool that answers reachability questions with replayable, annotated, agent-discovered program traces.
- Evidence from a lab study that suggests Trailblazer leads to answers more efficiently, with greater acquired familiarity of the program flow.

2 Background and Related Work

2.1 Reachability Questions

When working in a code base, developers often need to find out how disparate parts of the code connect to each other. For example, they ask questions like "What is the original source of this data?" or "What parameter values does each situation pass to this method?" [27] These questions require identifying feasible paths through a program and locating the code structures where relevant behavior occurs. They are often called reachability questions [26].

Reachability questions are common in many development tasks, including debugging, implementation, code review, and refactoring [24, 26, 42]. Prior work has revealed them as not just frequent, but also central to progressing in programming tasks [26, 27]. However, reachability questions can be difficult to answer. The information needed to answer them can be scattered across files, buried in layers of indirection, or hidden behind patterns like event dispatch or dynamic dispatch [26, 27]. Developers must decide where to begin, which paths to follow, and how far to go, often without strong cues to guide them [41]. While code navigation tools in the IDE can help developers answer them, they also require developers to piece together answers from long lists of low-level results [26].

2.2 Information Foraging Theory

In our work, we describe information foraging theory (IFT) as providing a template for designing autonomous components of tools for answering reachability questions. IFT models search as the iterative exploration of information patches with continuous judgments of which patches to explore [37, 38]. Some of its earliest applications focused on modeling web browsing behavior [7, 8], and since it has become an influential model for describing program debugging [16, 29] and code navigation [34].

We borrow the following constructs from IFT. In IFT, there is a predator (in our case, a developer) searching for prey (such as where a value is used in a program). The predator attends to cues in the code (e.g., identifier names, comments). These cues offer "scent," or a sense of how likely further investigation will lead to useful information. Developers follow scent to information patches, or sections of code, that may contain the prey. As they do so, they attempt to maximize insight while minimizing navigation cost.

2.3 Tools for Answering Developer Questions

Given the difficulty of answering reachability questions, tools have been developed to help developers answer them. One class of tools helps programmers trace lines of computation across their programs. For instance, slicing tools like Whyline [21, 23] and Flowistry [9] allow programmers to query for all lines of code related to a selected snippet. The tools use dependency analysis to find lines affected by or affecting the snippet [22]. Other tools [3, 28, 50] have supported navigation and review of programs as if they were trees or graphs. While such tools can be useful, there is also only so much support they can provide. Graph-based views scale poorly as a code base becomes structurally complex. Program slices can also become quite long if strict in its dependency analysis, assuming 10-50% of the source program length [4]. Trailblazer aims to get around these issues by combining a rough dependency analysis approach with an AI that filters what users are shown. Code structure is conveyed through a representation of a program as a pruned tree.

Another way to support developers in answering reachability questions is by structuring their workspace to better expose a code base and interrelationships between modules. Several tools have been built to do just this [6, 12, 18, 43]. Structural information can also be brought to where developers need it. For instance,

ReachHover [52] reveals structural context in hover popups with the intent to convey context about a code element under inspection. Trailblazer shares a similar goal of bringing information about far-flung code into one place, in its case in the form of a digested code tour and an annotated walkthrough of agent exploration.

2.4 Agents in Software Development

There is a growing body of work on developing agents to assist with software development tasks.In this paper, we use the term *agent* to refer to an autonomous system that makes sequential decisions when performing complex, exploratory tasks. Agents have been developed not just for code generation [10, 17], but also for tasks like debugging [2], test creation [20], and pull request creation [32]. Some tools [10, 17] bring suggestions and interactive editing affordances into the editor. Recent research has introduced new methods for agents to perform exploration and multi-step planning [25], hierarchical localization of repository-scale issues [48], patch generation [45], timeline-based multi-agent coordination [14], and structured workflows for navigating technical documents [13].

HCI researchers have meanwhile been exploring how to incorporate AI into effective programming workflows. One class of tools explored is autonomous agents that collect context, make decisions across steps, and interact with the environment. ROBIN [2] does this to assist with debugging. Pail [53] does this to stage design alternatives for the programmer [53]. Fang et al. [15] add to this space by visualizing the evolution of code and data with a history graph that supports interactive exploration. One challenge has been to expose an AI's "thought process." These tools expose the current focus of an AI collaborator in the code [39], and visualize the steps taken by an AI in assistive data analysis [49].

Complementing these technologies are intelligent interfaces that have sought to help developers understand their code, whether decomposing algorithmic programming problems [31], relating code to outputs and explanatory text [30], or explaining the code that the AI has generated [51]. Closely related to our work is DEDALE [11] which, given a code base, provides an explanatory map that follows along the lines of control and data flow. Where DEDALE explanations are offline and per-code base, Trailblazer provides answers to specific questions accompanied by walkthroughs and tight interconnections with the code in the editor.

3 Formative Study

In the first stage of our research, we took stock of the challenges to using existing dependency and search-based IDE tools to answer reachability questions. We recruited ten programmers through LinkedIn, X, and a graduate student mailing list at the University of Pennsylvania. All participants had prior experience searching in large code bases. Four participants regularly worked with repositories containing hundreds of thousands of lines of code, and six worked with repositories containing tens of thousands of lines.

Participants worked to answer five reachability questions in a substantial Java code base (the IntelliJ Community repository). We focused on Java due to the substantial code search tools available in the IDE. In particular, this allowed us to provide participants the "Data flow to/from here" feature, which allows programmers to step forward and back in program slices starting from a position of interest. Participants were asked to think aloud. Afterward, we asked how they typically search unfamiliar code, perceptions of current search tools, and perspectives on AI programming assistants. We analyzed the sessions following a thematic approach, where one researcher conducted an initial open and axial coding pass, with a second researcher reviewing the themes between the two passes.

Our study showed the following challenges in using traditional IDE search tools to answer reachability questions.

Too many options. One challenge participants faced was dealing with an overwhelming amount of information, particularly when exploring all references to a code entity, data flow views, or call hierarchies. Eight participants (P1, P3, P4, P6–P10) described struggling with too many paths to consider, making it hard to know which direction to pursue. Several participants pointed out the difficulty of distinguishing relevant branches, with P9 likening the process to "pulling a ball from a bag" without knowing what you will get.

As the branches grew deeper and more numerous, decision-making burden increased. P9 reflected, "I'm expanding this far, it's already overwhelming. And then there's more branches down here to also look at." While some were willing to explore up to 10 layers deep in the data flow (P9, P10), others reported fatigue after just 2 or 3 levels (P7, P8). Participants also desired a limit to the number of branches available to explore. When asked about the ideal branch factor for the data flow analysis tool, four of them cited five as the upper bound for what felt manageable. As P7 put it, "If I'm gonna explore by myself, five should be enough for me."

Seven participants (P1–5, P7, P8) desired the ability to filter results. The tools they used often lacked the ability to hide irrelevant results or highlight particularly relevant ones. Participants wanted to filter based on only relevant code variable names (P2, P5, P7, P8), constructs like assignments or function calls (P2–5, P7), or classes of files (P1, P3).

Insufficient context. Relatedly, the results returned by tools like "Find Usages" could lack the necessary cues to steer choice. Many results showed a single line of code without additional contextual lines of code. P9 wished for "a short description of what this line does… like this line transforms or checks if a parameter is not null, just a very short phrase." Others wanted semantic summaries that aligned with their question or goal. P8 suggested, "If we find the right place, [the tool] can also explain it to me… help me to understand more about this parameter object." Five participants (P4, P6, P8–10) imagined intelligent systems that could summarize possible paths or highlight the most relevant ones based on their current question. P6 described, "Maybe the natural language interface can help me look for one of these branches… and at least generate some code explanations."

Four participants (P3–5, P7) desired visual representations of code relationships, especially when navigating complex call graphs or tracing data flow. Instead of inspecting raw lists or deeply nested hierarchies, they wanted tools that laid out relationships between functions, classes, and data flows to make exploration more intuitive. P4 remarked wanting representations of program flow where "you could see them diverging and then coming back together, like a system." P7 asked for visual cues to help them focus on significant parts of code snippets, such as where data is mutated.

Tracking progress. Participants sometimes found it difficult to keep track of their progress. Five participants (P2, P4, P8–10) described losing their place or struggling to return to paths they had previously explored. P4 shared, "I'm having a really hard time keeping track of what path I've been down, what path I want to go down next… And it looks like it will never end." Tools could also lead to cycles, where developers were confronted with locations they had already searched. Six participants (P2–5, P7, P8) mentioned that circular paths and duplicated results were difficult to detect with the existing tools. P2 described the suitability of breakpoint debugging-style exploration for this kind of exploration, telling us "The breakpoints are actually better because they give us a top-down instead of a bottom-up approach. With breakpoints, we can go forward or backward in execution, but static trace tools just expand everything without showing the sequence clearly."

4 Design

Based on our formative study, we derived several motivations for improved editor tooling for answering reachability questions:

D1. Reduce choice. Tools should reduce the decision space. It might do so by filtering out irrelevant paths and prioritizing high-value paths. Effective tools may rely on knowledge of what lies in a candidate path to make well-informed decisions.

D2. Contextualize findings. When showing search results, tools need to provide sufficient information for the user to put them in context. This might be done with brief summaries, code context, or cues of how snippets connect to each other.

D3. Support orientation. One role a tool can play is in helping a programmer orient to the code base. They can record starting points of search, support logical stepping through the code, convey progress through a path, and avoid loops.

4.1 A Notional Model for a Trailblazing Assistant

Based on these motivations, we designed Trailblazer as a tool that brings together the tools of the IDE and AI to assist in the answering of reachability questions. Given that developers do not currently have tools like this, we found it useful to articulate a notional model of how such a tool works. Here is that notional model.

Should a developer have a question, they can ask it of Trailblazer.[1] Trailblazer launches an intelligent agent. The agent explores the code base to answer the user's question. It starts at the user's selection and then begins to look for answers to the question. As it does so, it refines the question into smaller and more concrete questions. The agent uses the standard tools of the developer's IDE, like "Go to Definition" and "Find All References" to look for code to explore. As it does, it tracks relevant findings and shares updates with the developer. It shares the fruits of its search in the form of a conventional answer, a "tour" of key code locations, and annotated, replayable walkthroughs of paths to those key locations.

[1]A metaphor for trailblazers that pass through the thick woods while blazing a path behind them that will be easier for those behind them to traverse.

4.2 Interface

The primary innovation of Trailblazer is its interface for making sense of code. The interface includes several key components. Their arrangement is depicted in Figure 2, for a developer who has asked the question "How is scrolling disabled when a modal is shown?" about the Material UI React components library.[2]

Question asking. Trailblazer is an extension to VS Code. As such, it is invoked from the standard command palette. When invoked, a dialog appears where a developer asks their question.

Status monitoring. To make developers aware of progress, Trailblazer provides continuous real-time updates (Figure 2, ❶). It reports when it is searching for new code snippets, deciding what to do next, formulating answers, and other essential steps.

Answer. The entry point to Trailblazer's output is a natural language response designed to resemble the typical format of AI responses. It serves as a high-level summary answer and gateway into deeper exploration. The answer is designed to be very concise, as developers are meant to save deep engagement for the tour and walkthrough. An example answer appears in Figure 2, ❷.

Descriptive tour of code. Trailblazer provides a concise tour of code snippets that collectively answer the developer's question with discovered snippets from the code. It is designed to present a small, representative set of locations to reduce the burden of making decisions (D1) while preserving essential context. For example, in the scenario in Figure 2, the tour conveys a couple of locations related to how scroll locking occurs when modals appear. One location checks whether scroll locking is enabled. Another location describes the function call that adjusts container styles. And a third location presents the statement that sets `overflow = 'hidden'` on the "locked" content container (Figure 2, ❸). Together, these locations form a coherent explanation that no single snippet can provide on its own.

Jump to the code. Each finding in the descriptive tour includes a link to open the corresponding location in the code editor (Figure 2, ❺), helping get more familiar with the code. Viewing the full code in the editor puts the high-level takeaway in context with the broader structure of the program (D2).

Walkthrough of the code. When answering a reachability question, developers often need to know not just where the answer is, but how it was found. The walkthrough feature in Trailblazer supports this need (Figure 3). It presents replayable traces of the agent's exploration up to each of the key snippets represented in the descriptive tour. Starting from the code location where the developer invoked the tool, the walkthrough steps through the shortest acyclic path to a selected code snippet from the descriptive tour.

Each step in the walkthrough shows a small snippet of code with a few surrounding lines of context. It is also accompanied by a brief explanation of how the agent reached that location (e.g., "I found `props`, which looked important and is related to

[2]A more scenario-centric view of the interface can be accessed in the usage scenario in the Appendix B and the accompanying video figure.

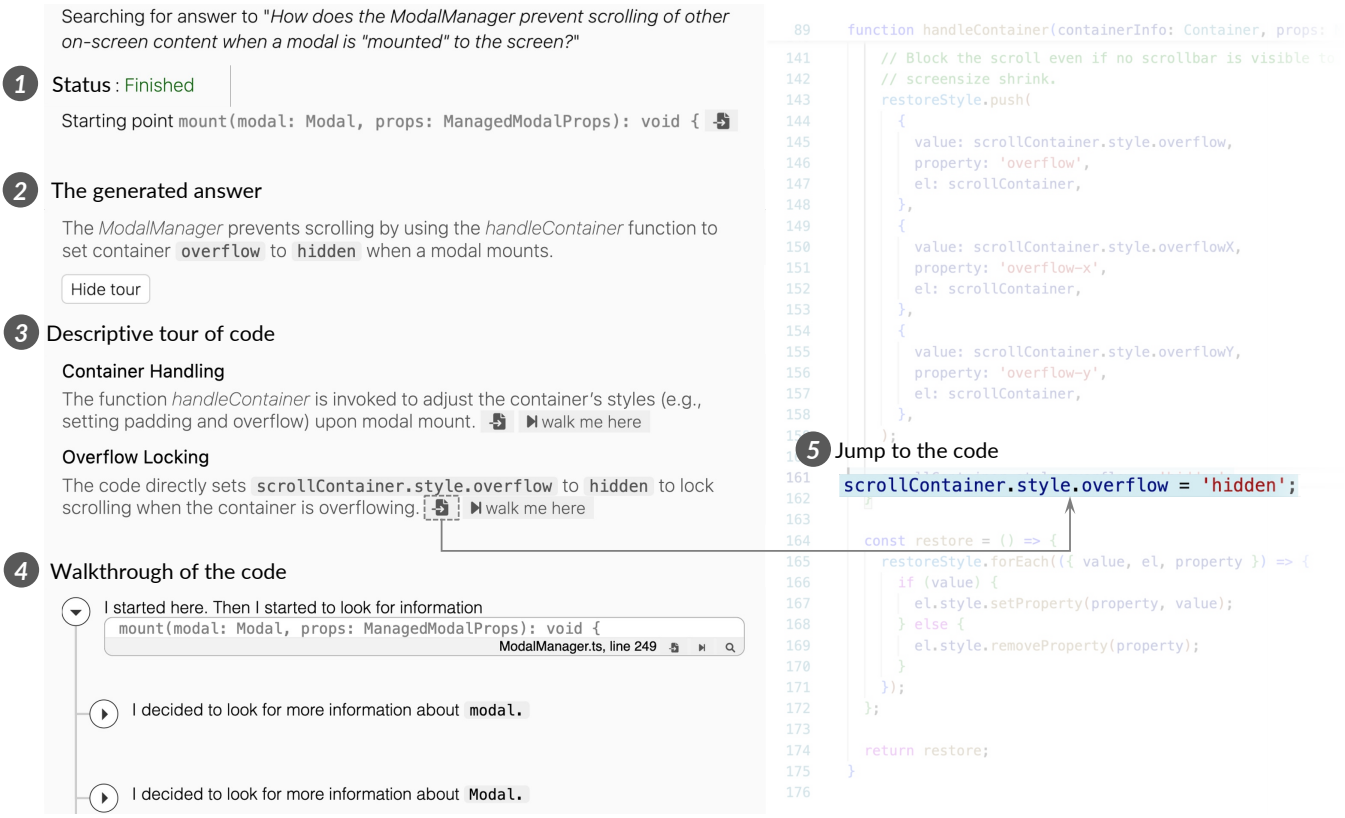

Figure 2: Trailblazer's interface. On the left is the Trailblazer interface, and on the right is the developer's code. The interface shows continual readouts of its status during search (❶). When it finds an answer, it first presents a concise, natural language summary (❷). Then it provides a descriptive tour of the code that elaborates on the answer with a tour of key locations in the code that contribute to the answer (❸). Finally, it provides an annotated, replayable walkthrough tracing from the site of the question through relevant variable usages, function calls, and control structures to elements of the answer (❹, an expanded annotated walkthrough appears in Figure 3). The developer can jump to snippets in context by clicking a button for a finding in the tour or a snippet in the walkthrough (❺).

`handleContainer`."), and a link to open the snippet in the code editor. If the snippet appeared in the tour, its description appears alongside the code listing in the walkthrough. These descriptions clarify the purpose of the snippet in relation to the trace (e.g., "The function `handleContainer` is invoked to adjust the container's styles…"). As the developer navigates forward or backward through the steps, Trailblazer brings up and highlights the corresponding code in the editor, helping them follow the path in context. In these ways, the walkthrough puts its findings in rich context (D2).

By default, a walkthrough begins at the point in the code where the developer asked a question, allowing them to retrace the agent's path from a familiar anchor. In the scenario from Figure 2, the walkthrough begins where the developer asked their question—at the `mount` method definition—and then steps to the answer by way of a call to `handleContainer()` function, eventually ending at the line where `scrollContainer.style.overflow = 'hidden'` is set. This walkthrough is minimal to the extent possible, representing the shortest acyclic path the agent found to the target snippet. This trace as a whole aims to help developers orient to the code, while also reducing disorientation that might arise from less linear exploration alternatives (D3).

Incremental updates. Instead of requiring the developer to wait for a complete answer, Trailblazer displays preliminary results as the agent explores. As soon as the agent identifies an initial path that may lead to an answer, it presents a preliminary answer. These updates continue in real time as the agent refines its understanding and uncovers additional evidence. When the agent has gathered enough relevant information, it finalizes the answer and walkthrough. Oftentimes, even preliminary findings are sufficient for developers to get the rest of the way to the answer.

4.3 Implementation

We implemented a proof-of-concept version of Trailblazer to demonstrate the feasibility of our interaction model. The system was tuned for TypeScript and TSX code, which offer strong typing that helps IDE tools reliably find definitions and references. While not a production-ready system, this prototype reflects the overarching architecture we believe is appropriate for answering reachability questions. Our implementation draws inspiration from information foraging theory (IFT, see Section 2.2,), modeling the agent's behavior as a forager that identifies prey, selects promising patches, makes local decisions based on context, and presents findings with

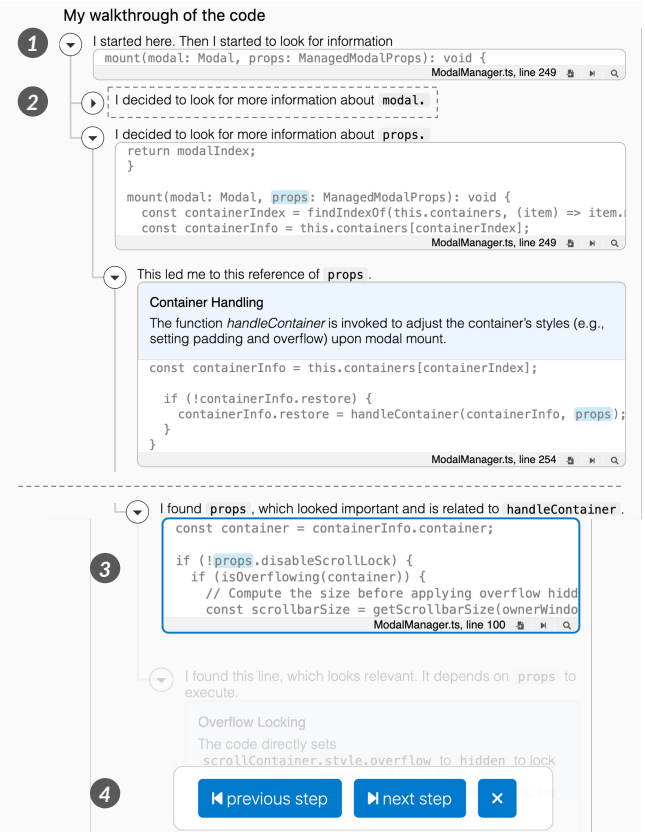

Figure 3: Replayable walkthrough of code exploration. The walkthrough begins at the code location selected by the programmer and reveals only the relevant steps leading to the target snippet (❶). Brief annotations at key steps explain how each location was reached (❷). At each step, the corresponding portion of the code snippet is automatically pulled up and highlighted with a light blue border (❸). Programmers can navigate through the walkthrough using forward and back buttons, enabling them to trace the path taken by the agent and inspect each step along the way (❹).

a walkable trace. Here, we provide an overview of how the system works, described in terms of the constructs of IFT.[3]

Prey. In Trailblazer, the prey is the answer to a developer's reachability question, such as "How is scrolling disabled when a component is shown?" To guide the search more effectively, we first refine the question using a language model, narrowing it to something like "Which functions are responsible for disabling scrolling when the modal is mounted?"[4] This helps orient the agent toward a more clearly specified target for exploration.

Patches. In IFT, a patch represents a local area of information worth exploring. In Trailblazer, patches correspond to specific locations in the code that may help answer the developer's question. For each line the agent considers as a candidate patch, the agent collects surrounding context (typically three lines before and after the

[3]Our implementation is available at https://github.com/YanLitao/Trailblazer.
[4]Prompts used for question refinement and exploration selection are included in `Prompts.pdf` in the supplemental material.

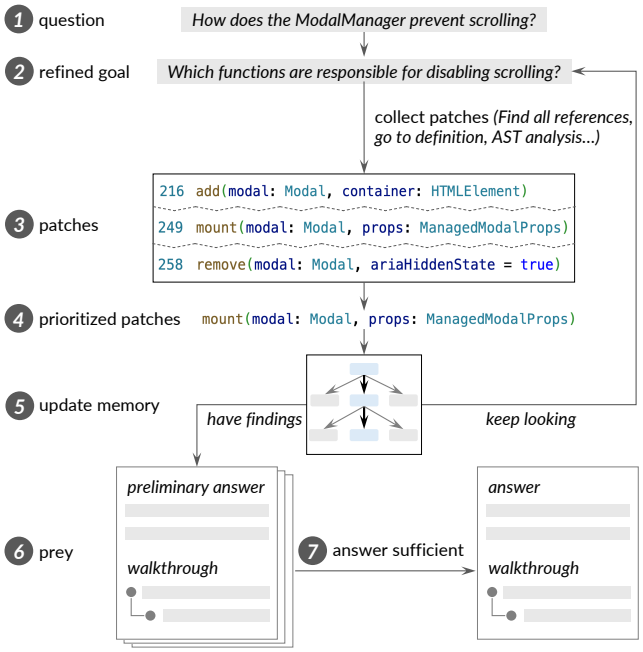

Figure 4: Implementation. Trailblazer simulates an information forager to answer reachability questions. It refines the user's question (❶ and ❷), identifies patches using IDE tools (❸), and selects which ones to explore next (❹). The system maintains a memory of visited paths (❺) and assembles its findings into a generated answer, descriptive code tour, and step-by-step walkthrough (❻). It reports out once the accumulated findings are sufficient to answer the question (❼).

line) to help programmers better understand the meaning. Patches are created using static code dependencies gathered by IDE-based tools. These include reference lookups (e.g., where a function is called or a variable is used), definition jumps (e.g., where a variable or function is defined), and assignment traversals (e.g., tracking dependencies across both sides of an assignment). The agent also descends into classes, function bodies, and control structures to discover new patches. For example, when the agent encounters a conditional like `if (props.disableScrollLock)`, it follows into the body of the condition and inspects each statement within the block, treating them as potential patches to explore.

Choice. In IFT, foragers make sequential decisions about which patches to pursue, guided by perceived information scent that indicates the likelihood of finding valuable information. In Trailblazer, this decision-making process is simulated by a large language model, which selects which patches to explore next from among a list of patches. The agent does not exhaustively explore every path. Instead, it selects a small number of promising patches to follow. Unlike traditional slicing or graph exploration techniques that can traverse all reachable paths, this approach seeks to approximate high-value exploration with fewer targeted steps.

Memory. Trailblazer maintains two forms of state throughout exploration. First, it constructs a complete program graph that records all visited code locations and their relationships. This graph serves

as a persistent log of the exploration process. Second, the language model receives a compact context that includes the original question, any refined sub-questions, and a digest of previously visited patches. Each patch is summarized with a short natural language explanation and a relevance score indicating how closely it aligns with the current goal. The system also tracks which variables have been explored, which findings contribute toward a possible answer, and which paths have already been followed. These representations help avoid redundant steps and support incremental progress toward assembling a complete answer.

Reporting out. As exploration progresses, the agent regularly evaluates whether it has collected sufficient evidence to answer the question. It is asked whether the current findings form a complete and coherent trace of the relevant execution path, without major gaps in data or control flow. It produces a concise natural language summary that serves as the generated answer, a set of labeled and annotated code snippets used to construct the descriptive tour of code, and a trace that links these snippets together into a walkthrough of the paths through the code base. This walkthrough is derived by computing the shortest acyclic path from the starting code to each tour snippet in the program graph, minimizing extraneous detours. If the current findings are potentially useful but insufficient to fully answer the question, the agent generates a preliminary answer as well as formulates a more specific follow-up question and continues its search.[5]

Optimizations. To improve speed and scalability, Trailblazer applies several optimizations. When the number of candidate patches is more than 25, it splits them into smaller batches and processes them in parallel using multiple LLM calls. This helps avoid input length limits and reduces latency.[6]

In addition, Trailblazer uses different models for different stages. A smaller, faster model (`GPT-4o mini`) handles exploration steps, while a reasoning model (`OpenAI o3-mini`) is reserved for generating final answers, where response quality is more important than speed. `GPT-4o mini` was selected as the default exploration model because it responds 5–10 seconds faster than the full `GPT-4o` model on longer prompts [47], offering a better tradeoff between latency and reasoning fidelity. We selected `OpenAI o3-mini` for final reasoning steps due to its strong performance on logical reasoning benchmarks like MMLU [35]. These optimizations reduced total processing time from several minutes to roughly ten seconds, allowing developers to receive preliminary answers quickly and iterate without long waits.

5 Study Design

To evaluate the impact of Trailblazer on answering reachability questions in unfamiliar code bases, we conducted a controlled within-subjects study. We compared Trailblazer to an AI question answering baseline. Our first question was:

RQ1. Does the tool help people answer reachability questions?

We were also interested in whether interacting with the tool would help developers better orient to the code (D3). This felt particularly important given that reachability questions often arise as part of broader program exploration [26, 27], where developers wish to build familiarity with how code is structured and how data or control flows through it. We therefore asked:

RQ2. Does the tool affect familiarity with the program?

Finally, we hoped to do a preliminary probe into situations in which Trailblazer would be the most useful in practice. So, we additionally asked:

RQ3. For what kinds of situations do programmers anticipate Trailblazer as being useful?

5.1 Participants

We recruited 20 programmers from academic mailing lists in the computer science department at the University of Pennsylvania. Of these, 11 identified as male and 8 as female. Participants included 2 bachelor's students (10%), 11 master's students (55%), 8 doctoral students (40%), and 1 academic researcher (5%). In terms of overall programming experience, 47.4% of participants had more than 5 years, 26.3% had between 3 and 5 years, and 26.3% had between 1 and 2 years. Because the code base used in the study tasks was written in JavaScript and TypeScript, we also collected their self-reported experience with those languages. Among the participants, 36.8% identified as beginners, 57.9% as proficient, and 5.3% as advanced.

Participants reported frequent use of core code navigation and debugging tools in their development environments. For example, the majority used go to definition daily or more (55%), with 85% using it at least weekly. Participants were also asked about their use of programming tools and AI assistants. 85% reported using AI to help with programming on a daily basis, and the remaining 15% used it weekly. The most commonly used AI tool was ChatGPT (used by 95% of participants), followed by GitHub Copilot (30%).[7]

5.2 Baseline

We selected Cursor as our baseline because it performed best in answering reachability questions during pilot testing. Cursor is an AI-enhanced code editor built on VS Code [10], with a chat interface that allows developers to ask natural language questions grounded in project context. It supports code selection for added context and returns answers with inline explanations and clickable code links. During the study, Cursor used the GPT-4o model, its most capable option at the time of testing. Prior to the study, we allowed Cursor to index the full code base to ensure fair comparison. We did not include a separate condition for using standard IDE tools alone, because participants were allowed to use these tools freely in both study conditions. This choice reflects realistic usage patterns, where they often use IDE tools with other resources, such as AI assistants, to answer complex questions.

5.3 Procedure

Each participant completed a one-hour study session. To minimize demand characteristics, we referred to tools with neutral names ("Tool A" for baseline and "Tool B" for Trailblazer). We counterbalanced the order in which participants used them. After consenting and filling out a background questionnaire, participants moved

[5] See Section A.6 for preliminary timing measurements of answer generation.
[6] See Appendix A.5 for a preliminary evaluation of batching efficiency.
[7] See Appendix C.1 for participants' reported use of programming tools.

through a tutorial, two code question answering tasks, a code ordering puzzle, and an open-ended exploration task to explore their intended use cases and question strategies with Trailblazer.

Tutorial. Before beginning the tasks, participants completed a guided tutorial on the editor and available tools. To ensure familiarity with basic navigation features, we first introduced VS Code's built-in tools, including Go to Definition and Find All References. Each study tool was introduced immediately before the task it was used in. All tutorials used the same short code file, where participants used the tool to answer a warm-up question.

Timed Code Question Answering Tasks. We asked participants to answer reachability questions under a time limit to answer RQ1. Each participant completed two code question answering tasks (Task A and Task B), one with each tool. Task order was counterbalanced to mitigate order effects. In each task, participants were given a reachability question about a real React component from the Material UI code base, along with a starting point in the code. They were asked to search the code to find where the behavior occurred and say the answer aloud when they found it, pointing to the relevant code. If a participant's answer was incomplete or incorrect, the facilitator prompted them to continue exploring until they identified the correct location and rationale. We chose these questions based on our own curiosities while exploring the Material UI code base. Both questions required multiple hops through the code to answer, spread across either 1 or 2 files. Each question typically engaged a few files spanning approximately 300 to 500 lines of code each.

One example question asked participants to investigate how the `ModalManager` component prevents background scrolling when a modal dialog is displayed. The task began at the mount method within the `ModalManager` file and required tracing how props were used to configure this behavior.[8] To prevent sessions from running over, tasks were limited to 10 minutes, though most participants completed them in less time. We stopped the timer only after participants finished explaining their answer and the facilitator confirmed it was correct. Participants were requested to use the assigned tool to do the task.

Code ordering puzzle. After both question answering tasks, participants completed surprise tasks designed to answer RQ2. Participants completed puzzle that required them to recall the code that constituted the answer, in order. The style of puzzle was Parsons puzzles [36]. Each puzzle presented 10 lines of code—4–5 lines from the correct answer and 5–6 fabricated distractors. Participants needed to identify the lines from the correct answer and then arrange them in the correct execution order. Each puzzle had a 10-minute time limit. Participants always completed the puzzle for Task A first, followed by Task B, regardless of tool or task order for the question answering tasks. These puzzles were used as a proxy for what developers learned about the structure and flow of the code. Participants could submit their answers multiple times and received feedback after each attempt, allowing them to revise their solution until it was correct. This enabled us to measure both accuracy and the number of trials needed to reach a correct answer.

[8]All task instructions can be found in Appendix C.2.

Open-ended usage. Participants spent the remaining time in a prompted exploration task (typically 10 minutes or less). They were asked to formulate their own questions about the Material UI code base and investigate them using Trailblazer. If they were unsure what to ask, the facilitator offered example topics to help them get started. They were asked to think aloud about their findings and reflections on where they thought the tool could be useful.

Questionnaires. Participants completed three sets of questionnaires during the session. After each question answering task, they completed the NASA-TLX workload assessment [33] and reported the usefulness of the tool and its individual features on 7-point Likert scales and an open-ended question. After completing both code ordering puzzles, they filled out a final questionnaire.

Measurements and analysis. We applied linear mixed-effects models to analyze task completion time and the number of trials needed to complete the tasks for RQ1 and RQ2. These models estimated the effects of several experimental factors on performance. Models included fixed effects for tool, task, tool order, and task order, interaction between tool and task, and a random intercept for participant ID. Significance was determined using F-tests with Satterthwaite's approximation of degrees of freedom [40]. For Likert-scale questions, we used two-tailed Wilcoxon signed-rank tests [46]. All p-values were adjusted using the Holm-Bonferroni method [19] across all tests, and the threshold for statistical significance was set at $\alpha = 0.05$. We analyzed open-ended responses through a thematic analysis process [5, Chapter 5]. One author performed initial open coding to identify recurring ideas, which were then refined through discussion with another author to derive key themes.

6 Results

Our findings for each research question were as follows.

6.1 RQ1: Improvements in Answering of Reachability Questions

Question answering performance. All participants completed the task within the 10-minute limit when using Trailblazer. In contrast, 7 participants (35%) did not finish on time with the baseline. A linear mixed-effects model revealed task completion time was significantly lower with Trailblazer ($\mu = 4.8$ min, $\sigma = 1.7$ min) than with the baseline ($\mu = 7.3$ min, $\sigma = 3.0$ min, $F = 33.4$, $p < 0.001$). No other fixed effects, including task, tool order, task order, participants' proficiency with JavaScript/TypeScript, or overall programming experience, were significant ($p > 0.2$), and there was no significant interaction between tool and task ($F = 1.4$, $p = 0.72$).

NASA TLX and tool ratings. Following the question answering tasks, participants rated Trailblazer as significantly less mentally demanding, effortful, and frustrating than the baseline, while reporting higher feelings of success and lower time pressure. All differences were statistically significant at $p < .01$, based on Wilcoxon signed-rank tests. Figure 6 shows the distribution of TLX ratings, and detailed statistics are available in Appendix C.3.

Participants were also asked to rate each tools' usefulness. They rated Trailblazer as significantly higher ($\mu = 6.6$, $\sigma = 0.6$) than the baseline ($\mu = 5.0$, $\sigma = 1.5$; $W = 3.5$, $p = 0.001$), and also reported it

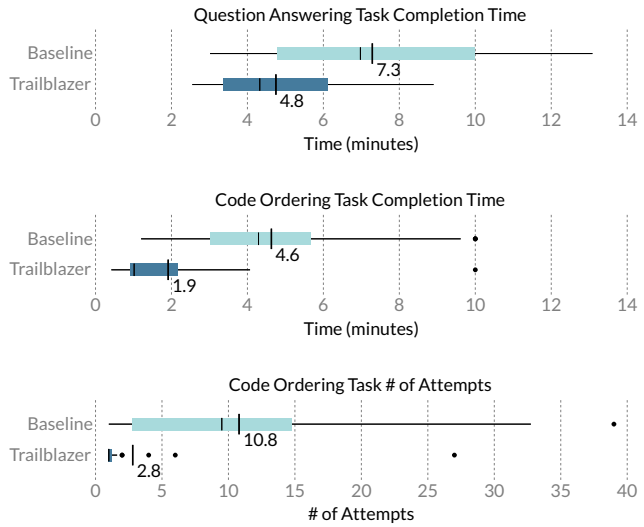

Figure 5: Task performance by tool. Box plots show the distribution of completion times for the code question answering tasks and code ordering puzzles, along with the number of attempts required to solve the puzzles. Gray vertical lines within each box indicate the mean. Participants completed the question answering tasks significantly faster using Trailblazer compared to the baseline. Similarly, participants solved the code ordering puzzles in fewer attempts and less time with Trailblazer.

as better integrated into their workflow ($\mu = 6.5$, $\sigma = 0.8$ vs. $\mu = 4.7$, $\sigma = 1.4$; $W = 0.0$, $p = 0.002$).

6.2 RQ2: Increasing Familiarity with the Trace

Code ordering performance. One participant failed to complete the puzzle in time with Trailblazer, compared to three with the baseline. Puzzle completion was significantly faster with Trailblazer ($\mu = 1.9$ min, $\sigma = 2.1$ min, $Mdn = 1.0$ min) than with the baseline ($\mu = 4.6$ min, $\sigma = 2.7$ min, $Mdn = 4.3$ min). A linear mixed-effects model found a significant effect of tool on completion time ($F = 28.7$, $p < 0.001$). No significant interaction between tool and task was found ($F = 3.8$, $p = 0.28$).

Participants also required significantly fewer attempts to solve the puzzles when using Trailblazer ($\mu = 2.8$, $\sigma = 5.8$, $Mdn = 1.0$) compared to the baseline ($\mu = 10.8$, $\sigma = 9.5$, $Mdn = 9.5$), with a significant effect of tool ($F = 31.4$, $p < 0.001$). The mixed-effects model revealed that neither JavaScript/TypeScript proficiency nor overall programming experience had a significant effect on puzzle completion time ($p = 0.41$, $p = 0.42$) or number of trials ($p = 0.73$, $p = 0.47$). Notably, 75% of participants solved the puzzle in a single trial with Trailblazer, while only 10% did so with the baseline.

Self-reports of effects on awareness of the code. Almost all participants ($P1$, $P3$–8, $P10$, $P12$, $P14$–20) described Trailblazer as helping them understand the code base as a whole. For example, $P17$ wrote that the ability to jump to relevant code while reading the walkthrough "was very useful in contextualizing how the code executes". Some participants remarked that Trailblazer informed them beyond their questions, such as $P1$ who wrote that the walkthrough "would bring me to interesting code lines that I would miss without using the tool". $P6$ wrote that the walkthrough could "help me understand the answer and even the program".

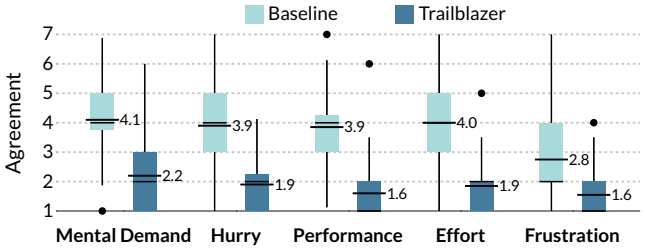

Figure 6: Perceived task load by tool. Box plots show participants' ratings on five NASA TLX dimensions following each code question answering task, grouped by tool. Lower values indicate lower mental demand, frustration, effort, and time pressure, and greater perceived success. Across all items, participants reported lower load when using Trailblazer compared to the baseline.

6.3 RQ3: Situations of Anticipated Utility

Types of questions asked with Trailblazer. During open-ended usage, participants used Trailblazer to ask a range of questions about unfamiliar code. Many of these targeted high-level understanding. Several participants described using the tool to get oriented when first encountering a new code base, posing questions such as "what does this function do?" (P1, P4, P5, P10), "what does this class do?" (P9, P11), or "what is the goal of ModalManager?" (P3, P11). Others asked for broader overviews to understand system behavior or design intent, such as "can you give me an overview or example?" (P16). In addition to these general questions, participants used the tool to ask more localized questions about variables, parameters, or usage patterns. For instance, "what does the variable `hidden` mean?" (P3), "what are valid values for this field?" (P10), or "how do I use this component in my code?" (P14). Some participants also asked process-oriented questions about implementing functions or tracing behavior, such as "find out all the related context of this function, and tell me how I should implement it" (P19), or "help me locate the key parts of the code in a long function" (P2).

As one proxy of accuracy on questions from this section, we analyzed participants' open-ended feedback and found that 8 participants explicitly reported that Trailblazer successfully answered their questions. These included questions like "What does this function (P1, P4, P5, P10) or class (P9, P11) do?", "What parameters are defined for the Button, and what are their uses?" (P6), and "How do I use this switch component in my code?" (P14). These examples are suggestive of the kinds of questions programmers might ask if provided a tool like Trailblazer with similar training.

Anticipated usage scenarios. Participants imagined using Trailblazer in a range of real-world situations. Some said they would use it when joining a new project (P5, P15) or exploring an unfamiliar code base (P1, P10, P14, P19), using it to identify important functions (P15), understand high-level structure (P1), or learn how parts of the system work together (P10, P14). Others said they would use it to follow how specific variables or components behave across the code base (P1, P8, P17, P20), or to trace how certain features are

implemented (P4, P9, P16). Some appreciated that Trailblazer could help refine early ideas into clearer answers (P3, P6, P12).

Some also described how Trailblazer might be used synergistically with other tools. For example, P17 used it to locate relevant code before switching to another LLM for more elaborate explanations. P18 and P19 preferred copying Trailblazer's answer into tools including ChatGPT and Cursor to summarize or act on its findings. P20 envisioned pairing Trailblazer with a code generator to understand functionality first, then generate implementation. These examples suggest that for these participants, Trailblazer could serve a complementary role to other assistive tools.

7 Discussion

In our study, Trailblazer led participants to answer reachability questions more efficiently than the baseline (Section 6.1, RQ1). Participants demonstrated greater familiarity of the relevant program trace in a surprise assessment (Section 6.2, RQ2). They also envisioned using multiple uses of Trailblazer, including to answer questions across unfamiliar or complex code bases (Section 6.3, RQ3). Below, we put our findings in context, first revising the design motivations posed in Section 4.

7.1 Reexamining the Design Motivations

Behavioral and self-report data from the studies help us to assess whether our design lived up to its motivations.

D1. Reduce choice. Trailblazer was designed to reduce decision volume by narrowing the search space to high-relevance paths. While we did not directly measure decision volume, we can examine participant tool use as a rough proxy of the actions required by participants. To characterize the gap between the possible search space and what participants actually engaged with, we analyzed graphs generated by the agent when run on the same question answering tasks by an author. In Task A, the agent explored 176 unique lines of code, constructing a graph with 313 nodes and 698 edges; in Task B, it explored 307 lines, forming a graph with 336 nodes and 810 edges. Each node in this graph represents a distinct code entity, like a variable or a function call, with edges representing the paths connecting code entities.

Where the agent made hundreds of transitions, participants exhibited far fewer. They jumped to a median of 18.5 and 17.5 lines of code (including repeated visits) in Task A and B, respectively. They made little use of in-editor navigation tools (median of 0 uses of search, Go to Definition, and Find All References when using Trailblazer). That participants still answered questions correctly suggests that Trailblazer's set of options took a large search space and distilled it to an appropriate set of code locations. While the agent's path includes some branches a human developer might not choose to follow, the system's filtering of its exploration appears to have been effective.

D2. Contextualize findings. Trailblazer contextualizes findings by ordering them in several ways. It provides tours and walkthroughs, annotates them, and allows jumping from snippets and answers to the code. The ability to jump to code in context was commented on widely. A majority of participants (*P*1–3, *P*6, *P*7, *P*9, *P*12, *P*14–16, *P*18) wrote that they valued being able to easily jump from explanations to corresponding code. The ability to jump to code referenced in the answer was a common point of direct comparison between Trailblazer and Cursor, with the majority mentioning that being able to click on links to locate code was desirable. 9 participants (*P*2, *P*4, *P*5, *P*7, *P*14, *P*15, *P*17–19) described having to manually search for code from the code blocks inside Cursor's answer because it did not give them a direct link that they could click. 5 participants used Cursor's clickable inline code references (similar to Trailblazer's "Go to code" links) and expressed appreciation that they made it easy to locate relevant code.

D3. Support orientation. Our clearest indicator that Trailblazer supports orientation is improved recall of the program flow during a surprise assessment (see Section 6.2). Participants' feedback suggests congruence between the design and how programmers think of code. Six participants (*P*5, *P*7, *P*10, *P*11, *P*16, *P*20) remarked that Trailblazer "thought" how they would think. *P*11 wrote that "I was very impressed by Trailblazer's ability to step through the code base in a manner similar to what [I] would have done as a human". 10 participants (*P*1, *P*3, *P*4, *P*8–10, *P*12, *P*15, *P*16, *P*18) mentioned that the structure of Trailblazer's answer helped them understand the answer to their question. *P*12 emphasized that the structure "[helps] me understand its thinking process, allowing me to understand the answer with intermediate steps, and making it less confusing and mentally burdensome". *P*1 wrote that the "logical flow and visualization" of the walkthrough helped "build up the mental model to help me build the connection between each related code line". This stood in contrast to the baseline, where 5 participants (*P*7, *P*9–12) described that Cursor's output was disorienting, citing its unstructured paragraphs and lack of navigation cues.

7.2 Limitations

Our findings have several limitations. First, our evaluation focused on two reachability questions that were heavily used as examples in the system's development (Task A from the start of development, and Task B at the end). As such, these tasks represent a best-case scenario for Trailblazer's current capabilities. We intentionally tested the system on questions we knew it could answer well, which allowed us to isolate and assess the value of its interaction model. However, this also means we do not know how Trailblazer would behave on the full gamut of possible reachability questions. Questions that span multiple files or involve more indirect dependencies may challenge the current agent and present more tangled answers for users in the current design.

Second, our participants were primarily graduate students. Their experience may differ from that of professional developers. For instance, they may have less exposure to large-scale code bases, more varied tool usage habits, and respond differently to time pressure in lab studies. That said, most participants reported regular use of modern IDEs and AI coding tools. While we believe that our study captures meaningful behaviors, future evaluation with industry developers is needed to understand how well these patterns hold in professional settings.

7.3 Future Work

Generalizing Trailblazers for other domains. While Trailblazer was designed for code base question answering, we envision that the underlying interaction pattern could extend to other domains. The core of our design pattern is to present agent-driven search as a trace that users can inspect and replay. This pattern is well-suited to settings where the problem space can be represented as a graph, the agent can access domain-specific search tools, the search process involves many decisions, and users need to understand what the agent did and how its choices led to an answer. This pattern could support exploration in any domain where people seek to recover and make sense of paths through graph-structured information. Examples include navigating the web, tracing citation chains in academic literature, exploring entailment trees in logical reasoning, or reviewing decision histories in workplace communication threads.

To apply Trailblazer to a new domain, several adaptations would be required. First, the system must be able to access or construct a graph that encodes meaningful relationships between units of information in that domain. Second, the agent must be able to operate domain-specific tools (e.g., academic search engines or simulation APIs) that enable search or data retrieval. Third, the search paths must be simplified so as to avoid showing untenably long or circuitous paths. Finally, the path should support walkthrough-style interaction where intermediate steps can be inspected with contextual cues and connections to the answer. Ideally, walkthroughs would resemble the paths that users themselves might naturally follow when exploring the space.

Consider the domain of scientific discovery. A Trailblazer-like system could help a researcher investigate why a new material exhibits a particular property by first building a graph of related concepts, prior findings, and outputs from material simulations. The agent could use domain tools such as academic search engines or computer simulation libraries to gather evidence, traversing the graph in order to identify leads. Like Trailblazer, it would follow informative paths while avoiding redundant or low-value branches, then assemble its findings into a step-by-step walkthrough. This walkthrough would include the final explanation as well as intermediate steps, such as which hypotheses were explored, which papers were read, and what simulation results supported the answer. The system would reflect a researcher's own inquiry process, while keeping its reasoning transparent and easy to inspect.

Collaborative human-agent exploration. Improving collaboration between developers and AI agents is an important direction for research in AI-based software engineering. While Trailblazer currently operates autonomously, developers often want to stay involved in the search process. They may prefer to perform much of exploration themselves. In other places, they may be particularly well-posed to find a logical leap inaccessible to an agent. One area for future work is designing systems where the agent can turn to the human for help when it encounters ambiguous or low-confidence situations. For example, when faced with multiple plausible directions, the agent could pause and present a decision point for the developer to weigh in. Conversely, perhaps a developer could invoke agents to do search in the background in increments while they themselves search, with opportunities for the two to continually share their findings. Building these kinds of interactions raises design questions about how to represent uncertainty, how to manage interruptions and task coordination, and how to maintain shared understanding of the current search state.

Scaling Trailblazer to explore large code bases. To make Trailblazer more usable beyond a proof of concept, it must be able to handle larger and more complex code bases. In repositories with millions of lines of code, the system may need to reason over thousands of exploration options, requiring performance optimizations to ensure responsiveness at scale. This will require integrating a more robust static analysis engine that can handle cross-language dependencies, resolve dynamic code references, and support efficient selection among large sets of candidate choices.

8 Conclusion

In this paper, we introduce Trailblazer, an interactive tool that helps developers answer reachability questions by presenting code base exploration findings as a step-by-step trace through the code. Rather than returning a flat response or a list of locations, Trailblazer presents answers as replayable, annotated traces that reveal how each piece of evidence was found and how they connect. Our study showed that this approach enabled participants to answer questions more efficiently and develop more familiarity of the program flow compared to an AI programming assistant baseline. In this way, our work provides a template for trace-based explanations of AI-assisted search in a domain that involves complex, structured information where transparency of the search process is essential.

Acknowledgments

We thank Daniel Fried for helpful conversations and feedback. This research was developed with funding from the Defense Advanced Research Projects Agency's (DARPA) SciFy program (Agreement No. HR00112520300). The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

References

[1] F. E. Allen and J. Cocke. 1976. A program data flow analysis procedure. *Commun. ACM* (1976), 137.

[2] Yasharth Bajpai, Bhavya Chopra, Param Biyani, Cagri Aslan, Dustin Coleman, Sumit Gulwani, Chris Parnin, Arjun Radhakrishna, and Gustavo Soares. 2024. Let's Fix this Together: Conversational Debugging with GitHub Copilot. In *2024 IEEE Symposium on Visual Languages and Human-Centric Computing*. 1–12.

[3] Mike Barnett, Robert DeLine, Akash Lal, and Shaz Qadeer. 2014. Get me here: Using verification tools to answer developer questions. *Microsoft Research, Tech. Rep. MSR-TR-2014-10* (2014), 10 pages.

[4] David Binkley and Mark Harman. 2004. A survey of empirical results on program slicing. *Advances in Computers* 62, 105178 (2004), 105–178.

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

[6] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. 2010. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2503–2512.

[7] Ed H. Chi, Peter Pirolli, Kim Chen, and James Pitkow. 2001. Using information scent to model user information needs and actions and the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 490–497.

[8] Ed H. Chi, Adam Rosien, Gesara Supattanasiri, Amanda Williams, Christiaan Royer, Celia Chow, Erica Robles, Brinda Dalal, Julie Chen, and Steve Cousins. 2003. The bloodhound project: automating discovery of web usability issues using the InfoScentπ simulator. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 505–512.

[9] Will Crichton, Marco Patrignani, Maneesh Agrawala, and Pat Hanrahan. 2022. Modular information flow through ownership. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, 1–14.
[10] Cursor. 2024. Cursor: The AI-Powered Code Editor. https://www.cursor.so/ Accessed April 1, 2025.
[11] DEDALe. 2024. DEDALe: Understand Your Code Like Your Best Engineers. https://dedale.dev Accessed April 1, 2025.
[12] Robert DeLine and Kael Rowan. 2010. Code canvas: zooming towards better development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*. ACM, 207–210.
[13] Moritz Dück, Steffen Holter, Robin Shing Moon Chan, Rita Sevastjanova, and Mennatallah El-Assady. 2025. Finding Needles in Document Haystacks: Augmenting Serendipitous Claim Retrieval Workflows. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*. ACM, Article 1003, 17 pages.
[14] Will Epperson, Gagan Bansal, Victor C Dibia, Adam Fourney, Jack Gerrits, Erkang (Eric) Zhu, and Saleema Amershi. 2025. Interactive Debugging and Steering of Multi-Agent AI Systems. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. ACM, Article 156.
[15] Hanxi Fang, Supawit Chockchowwat, Hari Sundaram, and Yongjoo Park. 2025. Enhancing Computational Notebooks with Code+Data Space Versioning. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*. ACM, Article 154, 17 pages.
[16] Scott D. Fleming, Chris Scaffidi, David Piorkowski, Margaret Burnett, Rachel Bellamy, Joseph Lawrance, and Irwin Kwan. 2013. An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks. *ACM Trans. Softw. Eng. Methodol.*, Article 14 (2013), 41 pages.
[17] GitHub. 2024. GitHub Copilot. https://github.com/features/copilot Accessed April 1, 2025.
[18] Emily Hill, Lori Pollock, and K. Vijay-Shanker. 2007. Exploring the Neighborhood with Dora to Expedite Software Maintenance. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. ACM, 14–23.
[19] Sture Holm. 1979. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics* (1979), 65–70.
[20] Dong Huang, Jie M Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010* (2023).
[21] Amy J. Ko and Brad A. Myers. 2004. Designing the whyline: a debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 151–158.
[22] Amy J. Ko and Brad A. Myers. 2008. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the 30th International Conference on Software Engineering*. ACM, 301–310.
[23] Amy J. Ko and Brad A. Myers. 2009. Finding causes of program output with the Java Whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1569–1578.
[24] Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering* 32 (2006), 971–987.
[25] Jing Yu Koh, Stephen McAleer, Daniel Fried, and Ruslan Salakhutdinov. 2024. Tree Search for Language Model Agents. *arXiv preprint arXiv:2407.01476* (2024).
[26] Thomas D. LaToza and Brad A. Myers. 2010. Developers ask reachability questions. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. ACM, 185–194.
[27] Thomas D. LaToza and Brad A. Myers. 2010. Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools*. ACM, Article 8, 6 pages.
[28] Thomas D. LaToza and Brad A. Myers. 2011. Visualizing call graphs. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing*. 117–124.
[29] Joseph Lawrance, Christopher Bogart, Margaret Burnett, Rachel Bellamy, Kyle Rector, and Scott D. Fleming. 2013. How Programmers Debug, Revisited: An Information Foraging Theory Perspective. *IEEE Transactions on Software Engineering* 39 (2013), 197–215.
[30] Yanna Lin, Leni Yang, Haotian Li, Huamin Qu, and Dominik Moritz. 2025. InterLink: Linking Text with Code and Output in Computational Notebooks. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*. ACM, Article 51, 15 pages.
[31] Shuai Ma, Junling Wang, Yuanhao Zhang, Xiaojuan Ma, and April Yi Wang. 2025. DBox: Scaffolding Algorithmic Programming Learning through Learner-LLM Co-Decomposition. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*. ACM, Article 585, 20 pages.
[32] Sanwal Manish. 2024. An autonomous multi-agent llm framework for agile software development. *International Journal of Trend in Scientific Research and Development* 8, 5 (2024), 892–898.
[33] NASA Ames Research Center. 2023. TLX @ NASA Ames - Home. https://humansystems.arc.nasa.gov/groups/TLX/ Accessed September 15, 2023.
[34] Nan Niu, Anas Mahmoud, and Gary Bradshaw. 2011. Information foraging as a foundation for code navigation. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 816–819.
[35] OpenAI. 2024. OpenAI o3-mini. https://openai.com/index/openai-o3-mini. Accessed: 2025-06-16.
[36] Dale Parsons and Patricia Haden. 2006. Parson's programming puzzles: a fun and effective learning tool for first programming courses. In *Proceedings of the 8th Australasian Conference on Computing Education-Volume 52*. 157–163.
[37] Peter Pirolli and Stuart Card. 1995. Information foraging in information access environments. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 51–58.
[38] Peter Pirolli and Stuart Card. 1999. Information foraging. *Psychological review* 106, 4 (1999), 643.
[39] Kevin Pu, Daniel Lazaro, Ian Arawjo, Haijun Xia, Ziang Xiao, Tovi Grossman, and Yan Chen. 2025. Assistance or Disruption? Exploring and Evaluating the Design and Trade-offs of Proactive AI Programming Support. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*. ACM, Article 152, 21 pages.
[40] Franklin E. Satterthwaite. 1946. An approximate distribution of estimates of variance components. *Biometrics bulletin* 2, 6 (1946), 110–114.
[41] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. 2006. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 23–34.
[42] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. 2008. Asking and Answering Questions during a Programming Change Task. *IEEE Transactions on Software Engineering* 34 (2008), 434–451.
[43] Vineet Sinha, David Karger, and Rob Miller. 2005. Relo: helping users manage context during interactive exploratory visualization of large codebases. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology EXchange*. ACM, 21–25.
[44] Justin Smith, Chris Brown, and Emerson Murphy-Hill. 2017. Flower: Navigating program flow in the IDE. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing*. 19–23.
[45] Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2025. CodeRAG-Bench: Can Retrieval Augment Code Generation?. In *Findings of the Association for Computational Linguistics: NAACL 2025*. ACL, 3199–3214.
[46] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in Statistics: Methodology and Distribution*. Springer, 196–202.
[47] Workorb. 2024. Comparing Latency of GPT-4o vs. GPT-4o Mini. https://www.workorb.com/blog/comparing-latency-of-gpt-4o-vs-gpt-4o-mini. Accessed: 2025-07-13.
[48] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489* (2024).
[49] Liwenhan Xie, Chengbo Zheng, Haijun Xia, Huamin Qu, and Chen Zhu-Tian. 2024. WaitGPT: Monitoring and Steering Conversational LLM Agent in Data Analysis with On-the-Fly Code Visualization. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*. ACM, Article 119, 14 pages.
[50] Litao Yan, Elena L. Glassman, and Tianyi Zhang. 2021. Visualizing Examples of Deep Neural Networks at Scale. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. ACM, Article 313.
[51] 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*. ACM, Article 140, 15 pages.
[52] James Yoo and Gail C. Murphy. 2023. Breaking the Bento Box: Accelerating Visual Momentum in Data-flow Analysis. In *2023 IEEE International Conference on Software Maintenance and Evolution*. 306–316.
[53] J.D. Zamfirescu-Pereira, Eunice Jun, Michael Terry, Qian Yang, and Bjoern Hartmann. 2025. Beyond Code Generation: LLM-supported Exploration of the Program Design Space. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*. ACM, Article 153, 17 pages.

A Implementation Details

A.1 Model Assignment by Task

Trailblazer uses a modular prompting architecture, assigning different models to different steps of the pipeline based on task complexity. Lightweight models such as `GPT-4o Mini` are used for fast-response stages, including refining user questions into concrete sub-questions (Task A, see Section A.3), selecting which variables to explore and which tools to apply (Task B), and scoring the relevance of exploration findings (Task C). For reasoning-intensive steps like

generating the final answer and deciding whether exploration is complete (Task D), the system employs more powerful models such as `o3-mini`.

A.2 Patch Mining

To simulate a foraging process over the code base, Trailblazer generates small, structured units of code context as patches. It uses a combination of IDE APIs and static analysis. Patches are centered around specific lines of code, typically including three lines of code both before and after the relevant line, providing surrounding context to aid understanding. These patches are generated by invoking standard VS Code APIs, including "Go to Definition" and "Find References." These tools allow the agent to resolve relationships between code entities, such as linking a variable to its declaration or finding usages of a function.

In addition to IDE tools, Trailblazer extracts dependency relationships from the code by analyzing assignment statements. Specifically, it traverses both sides of an assignment (e.g., in `x = y + z`) to identify dependencies. This is implemented using the TypeScript Compiler API, which provides access to the abstract syntax tree (AST) for each file. The system descends into control structures such as `if`, `class`, and `function` blocks to recursively analyze the variables and statements nested within. It also handles destructuring assignments and parameters in function calls, expanding these patterns into their constituent variable references.

A.3 Prompting Workflow

Trailblazer operates through a modular prompting pipeline, with each stage responsible for a distinct step in the exploration process. The workflow consists of four primary tasks:

- **Task A: Refine question.** The system refines the user's initial query into a more precise and actionable sub-question. This step helps the agent decompose high-level questions into manageable goals, often phrased in terms of specific functions, variables, or behaviors.
- **Task B: Choose patches to explore next.** Given a refined sub-question and a set of previously identified code locations, the agent decides which variables to explore next. For each variable, it chooses an appropriate tool (either "Go to Definition" or "Find References") and justifies its decision based on how it might help answer the question.
- **Task C: Rank patches.** After exploration, the agent receives the resulting patches and assesses their relevance. It assigns a binary score indicating whether each patch is useful and generates a concise summary of what was learned.
- **Task D: Reporting out.** The agent evaluates whether it has enough information to answer the original question. If so, it constructs a structured output including a short summary, annotated citations of relevant code, and a trace linking them together. If not, it refines the question and returns to Task B to continue the loop.

In addition to these core prompts, Trailblazer includes mechanisms for error handling, schema enforcement, and generating fallback strategies when outputs are incomplete. These measures ensure robustness and allow the agent to adapt when exploration does not go as expected.

A.4 Exploration Graph Structure

As Trailblazer explores the code base, it constructs a graph to store how different patches relate to one another. This exploration graph helps structure the final walkthrough shown to users.

The graph includes nodes and edges. Each node in the graph represents a specific code entity, such as a variable or a function name. Nodes store metadata such as the file URI and line number, the code context at that location, the key variable or function, and the tool used to reach it. Edges capture the relationships between nodes, reflecting how exploration flows across the code. It contains a source node, a sink node, and the tool used to reach the sink node.

When generating the walkthrough, the traces are generated by building a minimum spanning tree through the graph, prioritizing coverage and simplicity. Each trace is the minimal path from the starting point to the node shown in the tour. To prevent redundant or confusing paths, Trailblazer minimizes cycles by tracking visited nodes and merging duplicates as they appear.

A.5 Latency Comparison for Batch Processing

To evaluate the effect of batching on latency, we conducted a simple experiment simulating the conditions of Task A in our user study. One of the authors ran the tool on a question similar to Task A, comparing two configurations: processing 100 patches sequentially and in batches of 10. In the sequential setting, the tool processed patches one at a time using separate LLM calls. In the batched setting, the patches were grouped into 10 batches of 10 and processed concurrently. We observed that sequential processing took approximately 16 seconds, while batching reduced the latency to around 4 seconds. This evaluation clarifies why it is useful to batch decisions in Trailblazer.

A.6 Answer Generation Time

To estimate how long Trailblazer takes to generate answers in practice, an author recorded the duration of answer generation during each user study session. Since Trailblazer successfully reached the correct answer in all sessions, these measurements offers a rough indicator of its performance on questions of similar complexity. On average, Trailblazer generated a preliminary answer in 23 seconds (σ = 4s) for Task A and 31 seconds (σ = 7s) for Task B. The total time to produce a finalized answer was 95 seconds (σ = 8s) for Task A and 94 seconds (σ = 22s) for Task B.

B Usage Scenario

We demonstrate the major features of Trailblazer with an example of its usage in understanding a new code base. In this scenario, Mia is a new developer contributing to the Material UI library. She is trying to understand the `Modal` component, which shows a pop-up dialog. In particular, she wants to understand how showing a Modal component prevents the rest of the page from scrolling.

Asking a question. Mia has already done some investigation and is currently examining `ModalManager.mount()`, which is a method that seems to be invoked when the `Modal` is rendered. Instead of continuing to trace code manually, Mia decides to use Trailblazer to continue the investigation. To do so, she selects the code for `mount()` as the starting point and asks Trailblazer "How does the

ModalManager prevent the component from scrolling after the Modal is shown?"

Searching in progress. After Trailblazer is dispatched to answer the question, it works in the background while Mia is free to continue browsing code. As it does, it shows a side panel containing its current findings. If Mia thinks the current progress does not look promising, she can cancel the search.

Reading the answer. Mia first looks at the *generated answer*, which summarizes Trailblazer's findings. The generated answer says that scrolling can conditionally be disabled by setting a certain container element's `overflow` style to `hidden`. Mia doesn't fully understand this answer, so she scrolls down to the *descriptive tour*, which describes the different important pieces of code that contribute to the answer. Each element of the tour comes with a link to jump to the relevant code, allowing Mia to get a better understanding by comparing the high-level summary to the code. The two elements of the descriptive tour are "Conditional Lock Check" (whether the modal scroll lock is enabled) and "Scroll Lock Activation" (how the scroll lock is actually applied).

Digging deeper. Mia wants to know more about how scrolling is disabled, so she clicks on *Walk me here* next to the "Scroll Lock Activation" section. This activates an interactive walkthrough, where Trailblazer shows each piece of code that it encountered to arrive at a result, in the order of discovery. Since Trailblazer determines its answer by using code tracing tools, the process resembles how a human would manually perform this code exploration, which makes it easier for Mia to follow.

Since Mia initiated the search from the `ModalManager.mount()` method, the walkthrough begins there. Trailblazer's exploration agent chose to follow the `props` variable through multiple hops, eventually finding the conditional check for whether to disable scrolling. Within that conditional block, the container's `overflow` style is set to `hidden`.

As Mia follows this walkthrough, she learns that the `ModalManager` can track multiple containers, that it modifies the styles on the container by accessing the HTML element directly, and that it restores the container's original styles when the modal is dismissed. By following this walkthrough, Mia acquires an understanding of the code base beyond the answer to her question.

C Lab Study Details

C.1 Participants' Familiarity with Tools

"Find all references" was used daily or weekly by 75% of participants. Text-based search was especially common: 85% used "search within a file" daily or more, and 65% used "search across files" with the same frequency. For debugging, 80% relied on print statements daily or more, whereas breakpoint debugging was used less often, with only 20% using it daily. Participants also reported prior experience with other AI tools: Cursor (25%), DeepSeek (25%), Claude (15%), and Gemini (10%).

C.2 User Study Tasks

Both tasks were designed to be approximately equal in difficulty. When using manual exploration, each required 4–5 steps involving a combination of navigation tools, such as "Find All References", "Go to Definition", and inspection of conditionals or function bodies to locate relevant code. In both cases, the answer was located within a single file containing more than 300 lines of code, and identifying the key snippet typically required scanning through functions exceeding 100 lines. These structural similarities contributed to a comparable level of challenge across tasks. Here are the instructions for Task A and Task B used in our study:

Instructions for Task A

ModalManager.ts line 249

```
mount(modal: Modal, props: ManagedModalProps): void {
```

Context: The file you are in is for the `ModalManager` class. `ModalManagers` manage "modals". Modals are pop-up dialogs that appear on the screen. The `ModalManager` manages how these modals behave. For instance, one of the things it does is prevent the rest of the content on the screen from scrolling when a modal dialog appears. Many of its methods take "props" (React's standard parameter name for component options). The `ModalManager` reads these props to configure behavior of the modals.

Task Description: Find out how the `ModalManager` prevents scrolling of other on-screen content when a modal is "mounted" to (or activated on) the screen. Remember the scenario — you are a junior Material UI developer looking to understand this behavior so that you can later change it.

Instructions for Task B

FocusTrap.tsx line 136

```
function FocusTrap(props: FocusTrapProps): React.JSX.Element {
```

Context: The file you are in is the `FocusTrap` component.

First, what is focus? In a UI, "focused" widgets are those that can receive user interface events. If a widget is unfocused, a user cannot interact with it (i.e., click it, enter text, etc.).

Second, what is a `FocusTrap` component? A `FocusTrap` component "traps" the focus on a set of UI widgets. When you wrap a widget in a focus trap, it forces focus to stay on that widget. In doing so, it keeps focus from going to other UI elements outside of that widget, so that the rest of the UI becomes effectively non-interactive. When a `FocusTrap` is dismissed, it has the possibility of returning focus to where it was in the UI before.

Task description: Find out how the `FocusTrap` restores focus to the UI widgets that were previously focused when it is closed. Remember the scenario — you are a junior Material UI developer looking to understand this behavior so that you can later change it.

C.3 Additional Quantitative Results

NASA TLX items. Participants rated Trailblazer as significantly less mentally demanding ($\mu = 2.2$, $\sigma = 1.2$) than the baseline ($\mu = 4.1$, $\sigma = 1.4$; $W = 6.0$, $p = 0.002$), and reported feeling less hurried ($\mu = 1.9$, $\sigma = 1.0$ vs. $\mu = 3.9$, $\sigma = 1.3$; $W = 3.0$, $p = 0.001$), more successful ($\mu = 1.6$, $\sigma = 1.1$ vs. $\mu = 3.9$, $\sigma = 1.4$; $W = 0.0$, $p = 0.001$), and expending less effort ($\mu = 1.9$, $\sigma = 1.0$ vs. $\mu = 4.0$, $\sigma = 1.2$; $W = 7.0$, $p < 0.001$). They also reported lower frustration with Trailblazer ($\mu = 1.6$, $\sigma = 0.8$) compared to the baseline ($\mu = 2.8$, $\sigma = 1.4$; $W = 0.0$, $p = 0.004$).