Editrail: Understanding AI Usage by Visualizing Student-AI Interaction in Code

Ashley Ge Zhang
University of Michigan
Ann Arbor, Michigan, USA
gezh@umich.edu

Yan-Ru Jhou
University of Michigan
Ann Arbor, Michigan, USA
yanruj@umich.edu

Yinuo Yang
University of Notre Dame
Notre Dame, Indiana, USA
yinooyang@nd.edu

Shamita Rao
University of Michigan
Ann Arbor, Michigan, USA
shamita@umich.edu

Maryam Arab
University of Michigan
Ann Arbor, Michigan, USA
maryarab@umich.edu

Yan Chen
Virginia Tech
Blacksburg, Virginia, USA
ych@vt.edu

Steve Oney
University of Michigan
Ann Arbor, Michigan, USA
soney@umich.edu

Abstract

Programming instructors have diverse philosophies about integrating generative AI into their classes. Some encourage students to use AI, while others restrict or forbid it. Regardless of their approach, all instructors benefit from understanding how their students actually use AI while writing code [1, 15, 26, 39]. Such insight helps instructors assess whether AI use aligns with their pedagogical goals, enables timely intervention when they find unproductive usage patterns, and establishes effective policies for AI use. However, our survey with programming instructors found that many instructors lack visibility into how students use AI in their code-writing processes. To address this challenge, we introduce Editrail, an interactive system that enables instructors to track students' AI usage, create personalized assessments, and provide timely interventions, all within the workflow of monitoring coding histories. We found that Editrail enables instructors to accurately detect AI use that conflicts with pedagogical goals and to determine when and which students require intervention.

Keywords

Computing Education; Code Visualization; Student-AI Interaction

ACM Reference Format:
Ashley Ge Zhang, Yan-Ru Jhou, Yinuo Yang, Shamita Rao, Maryam Arab, Yan Chen, and Steve Oney. 2026. Editrail: Understanding AI Usage by Visualizing Student-AI Interaction in Code . In *The 39th Annual ACM Symposium on User Interface Software and Technology (UIST '26), November 02–05, 2026, Detroit, MI, USA.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3830398.3830696

UIST '26, November 02–05, 2026, Detroit, MI, USA

ACM ISBN 979-8-4007-2856-3/2026/11
https://doi.org/10.1145/3830398.3830696

1 INTRODUCTION

Recent advances in large language models (LLMs) present both opportunities and challenges in computer science education. While LLMs can boost students' productivity [40] and provide flexible feedback [33], they also raise concerns about over-reliance and reduced engagement in the learning process [10, 37, 40].

However, as we found in a formative study, instructors lack visibility into how their students use AI. This makes it difficult to assess how well students' use of AI agents aligns with pedagogical goals, limits their ability to intervene for students who over-rely on AI, and makes it more difficult to develop nuanced policies on AI use. Prior work has sought to address this challenge by surfacing students' interactions with AI systems, such as visualizing AI-student dialogues [8], but dialogue alone is insufficient to capture how students use and adapt AI-generated content in their learning.

To address this challenge, we propose Editrail, a system for understanding student–AI interactions and facilitating a feedback loop among instructors, students, and AI. Editrail integrates AI usage monitoring into existing code-progress tracking workflows by visualizing AI contributions as trails in students' edit histories. With these visualizations, instructors can use fine-grained contextual evidence to diagnose misunderstandings in AI-usage, deliver in-situ assessments, and provide timely support. For example, instructors can generate targeted quiz questions on concepts where students relied heavily on AI.

To evaluate the efficacy of Editrail in understanding students' AI usage and supporting timely intervention, we collected coding histories and AI interactions from 20 students across two Python programming problems and conducted a within-subject study with 12 instructors. Our findings show that by visualizing AI code contributions as trails in students' edits, Editrail enables instructors to identify AI usage more accurately and efficiently than a baseline system that shows only code and chat logs. Editrail also reveals new opportunities for designing personalized guidance around students' AI use. By bridging the visibility gap between students' interactions with AI and what instructors can observe, Editrail helps instructors

align interventions with pedagogical goals and inform effective classroom AI policies. This paper contributes:

(1) A need finding survey of 27 lead programming instructors at 15 universities, spanning multiple continents and course levels, identifying limited instructor visibility into student-AI interactions, challenges in enforcing AI policies, and design goals for monitoring and guidance tools.
(2) A dataset of 20 students' Python coding activities, capturing 188–319 lines of code, up to 2610 edits, and up to 42 AI chats.
(3) An interactive tool (Editrail) that enables instructors to identify AI usage patterns and deliver in-situ interventions.
(4) Evidence from a comparison study showing that Editrail bridges the visibility gap between students' AI use and instructors' observations, enabling alignment of AI policies and interventions with pedagogical goals.

2 RELATED WORK

2.1 AI in Programming Education

Generative AI has transformed programming education by automating tasks such as diagnosing bugs, generating feedback, producing code examples, and creating learning materials [4, 18, 22, 45]. AI assistants like Codex and CodeHelp have shown promise in helping novices by providing hints, scaffolding, and on-demand support [9, 11]. Studies also highlight cognitive engagement techniques—such as step-by-step prompting, grounded reflection, or worked examples—that can help learners integrate AI suggestions productively [19, 22, 32, 42].

At the same time, these benefits come with risks. AI-generated content can be inaccurate or misleading [5], and over-reliance may encourage novices to copy-paste solutions, bypass validation, or prompt toward full answers without developing problem-solving skills [3, 23, 36]. Classroom deployments reflect this tension, showing both efficiency gains and harms to deeper learning [37]. For example, a study with 69 novices found that learners with stronger prior programming knowledge benefited more from AI code generators, highlighting the need for tools that mitigate over-reliance, support effective prompt construction, and give better visibility into students' AI usage [21]. Preparing students and instructors to use AI responsibly is therefore critical [31].

Researchers have identified practical strategies for students to effectively learn programming with GenAI, including breaking problems down, iteratively refining prompts, and validating AI-generated code [28, 34]. Effective AI use requires substantial metacognitive activity, including planning, monitoring, and critically evaluating AI-generated content [46, 51]. To mitigate the risks of over-reliance, researchers have proposed designs such as teachable agents that students can debug [29], live programming tools that surface runtime behavior to validate AI code [10], and interfaces that promote trust, transparency, and control [7, 23, 35]. Yet, there remains a need for tools that help instructors monitor student–AI interactions, interpret whether usage aligns with pedagogical goals, and provide timely guidance to foster productive habits.

2.2 Student-AI Interaction in Programming

Recent studies have examined generative AI use in programming contexts, highlighting challenges such as over-reliance, insufficient validation, and a lack of iterative refinement [3, 36, 43, 44]. Case studies in introductory programming (CS1) have shown that students sometimes over-rely on AI, for example, by pasting entire task descriptions into ChatGPT without contributing their own effort, or by neglecting to verify AI-generated solutions even when they recognize errors [3]. Similarly, Prather et al. found that students often blindly accept AI suggestions, reinforcing misconceptions and encouraging surface-level completion rather than deeper problem solving [36].

To address these issues, researchers have proposed strategies to encourage more productive engagement. For example, Kazemitabaar et al. deployed CodeAid, an LLM-powered assistant in a semester-long course, and distilled design principles for classroom deployment: exploiting AI's unique advantages, balancing the directness of responses, and supporting trust and transparency [23]. In follow-up work, they examined cognitive engagement techniques, showing that step-by-step problem-solving guidance with interactive prompts before revealing a final solution was particularly effective at fostering learning [22].

In addition to fostering good usage on the students' side, complementary approaches have explored how to make student–AI interactions more visible and interpretable. For instance, Chen et al. introduced StuGPTViz, a visual analytics system that highlights temporal patterns in student use of ChatGPT [8]. Beyond education, work in HCI has proposed methods for interpreting LLM outputs in decision-making contexts [20], suggesting opportunities to adapt similar techniques for programming pedagogy.

However, this work has largely focused on AI outputs, such as what was prompted and what the model generated, without capturing how students engaged with these suggestions, whether they understood them, or how effectively they integrated them into their code. Our work fills this gap by integrating AI outcomes with the processes of adoption and adaptation, giving instructors visibility into how students interpret, modify, or struggle with AI-generated code. This process-oriented view helps instructors distinguish productive from unproductive patterns and provide targeted guidance aligned with pedagogical goals.

2.3 Providing Feedback on Students' Progress

Students benefit more from active engagement in problem-solving activities than from passive content consumption [24, 25, 48]. Within this context, feedback is understood not only as corrective but also as motivational, supporting students in persisting after errors and refining their understanding [49].

Prior work has introduced tools to help instructors understand students' code in order to provide targeted and timely feedback [12, 14, 16, 50, 52, 54]. Some systems cluster code submissions to reveal common approaches and mistakes, enabling reusable feedback at scale [12, 16, 54]. However, these techniques primarily operate on final submissions and overlook the dynamic process of how code is written. CodeOpticon addresses process-level visibility by streaming multiple students' live editors [14], but this requires intensive manual monitoring and does not scale to large classrooms. VizProg reduces this burden by visualizing coding progress as dynamic points on a 2D map [52], yet these tools focus on progress rather than the provenance of code. More recently, tools such as

Meta-Manager visualize metadata about code origins, for example labeling activities like copy–paste or Stack Overflow search [17]. However, such high-level summaries provide only coarse signals and are insufficient for instructors to fully understand how students use AI while programming. Editrail fills this gap by directly visualizing contributions from both students and AI within coding histories, enabling instructors to identify AI usage patterns and deliver timely, targeted support.

3 NEED FINDING STUDY

Prior work has examined AI tools in programming education on a small scale [5, 37], focusing on specific course types, regions, or AI tools. We extend this work through a global survey of programming instructors across levels to understand students' AI usage and the rationale behind classroom policies. To understand instructors' views on AI usage, we conducted a needs-finding survey and applied thematic analysis to code responses, informing the design of tools for understanding and intervening in student–AI interactions.

3.1 Study Method

The survey covers instructors' views on effective and inappropriate AI use, detection methods, classroom AI policies, and their rationales, along with demographic information (e.g., teaching experience, institution, course level).

3.1.1 Participants. To capture diverse experiences with AI integration, we targeted university-level lead programming instructors responsible for course-level AI policies in synchronous classes. We identified 44 institutions (41 highly ranked global universities plus three universities affiliated with the authors), selected relevant coding-heavy courses across undergraduate and graduate levels, and conducted personalized email outreach to the instructors. Of 208 instructors contacted, 27 instructors from 15 universities participated. All were Head or Lead Instructors, most with 3 to 15 years of teaching experience and instructing introductory or intermediate programming courses. Appendix A details our recruiting process.

3.1.2 Data analysis. We conducted an inductive thematic analysis [6], beginning with open coding [41] by two authors, after which six authors reviewed and finalized the themes. In parallel, we conducted a descriptive quantitative analysis of closed-ended and countable responses, using frequency counts (e.g., number of instructors reporting a pattern) to contextualize qualitative findings. The final themes covered: (*i*) perceived good and bad AI usage by students, (*ii*) policies and rationales regarding AI usage, and (*iii*) ways of detecting AI usage and instructors' thoughts on tool incorporation. We further synthesized three key findings: (*i*) the need for a better understanding of students' AI usage, (*ii*) the challenges in tracking AI usage, and (*iii*) unclear classroom AI policies. Based on our findings, we derived two design goals for understanding and guiding student-AI interaction in programming courses.

3.2 Lack of Visibility into Students' AI Usage

When asked about helpful or harmful usage patterns, instructors described isolated behaviors from individual students, lacking broader generalizability. We synthesized nine high-level patterns of inappropriate AI use (Table 1), most commonly involving submitting AI-generated code without understanding, using AI in individual assessments, and producing unusually advanced or course-inconsistent submissions, indicating external generation. Instructors also have limited tool support for detecting or preventing AI misuse. Twelve participants relied on manual checks as indirect indicators, such as *"importing external libraries"*, *"using coding conventions outside the scope of the course"*, or producing unusually polished code, and often conducted *"code reviews for those who achieve a grade that is 80% or higher."* One instructor required students to *"upload chat logs and explain how they use AI as part of their code documentation and report"*. Three others used tools like Turnitin [47], MOSS [2], or Gradescope [13] to detect plagiarism and analyze documents. However, these tools capture limited patterns and focus on final submissions, overlooking the problem-solving process. Overall, instructors rely on ad hoc, experience-driven heuristics, with no consistent or scalable methods for identifying inappropriate AI use across classrooms. Participants found it impossible to prevent or reliably detect such AI use in programming assignments. Instructors primarily care whether students bypass the learning process when using AI. Without direct evidence, they must infer student understanding from the final submissions alone, motivating DG1 (§3.5 on the following page below).

3.3 Tracking Students' AI Usage Is Difficult Due to Behavioral Invisibility and Variation

Instructors report several challenges in tracking students' AI usage. Most courses only collect final submissions, offering little visibility into students' processes, such as how they prompt LLMs, adapt generated code, or how much work is authored by students versus by AI. Instructors instead rely on indirect signals in final code or students' self-reports, both of which are unreliable, as they *"can't have any concrete evidence unless students confess."* Students also vary widely in AI use, from heavy reliance on AI-generated code to constrained or minimal use. Current tools do not support detecting or categorizing these behaviors, leaving instructors to conduct detailed process reviews, which are time-consuming and often impractical. One participant noted that they *"manually look through all the submissions for the AI red flags we've found, and bring in those students to see if they actually understand their code,"* but *"this is a lot of work"*, especially given the limited time instructors and TAs have. These findings suggested the need for intuitive visual signals of students' AI use during coding (DG1, §3.5).

3.4 Unclear Tracking of AI Usage Leads to Inconsistent and Unenforceable AI Policies

Policies varied by personal philosophy and course level, shifting from strict control in introductory courses to greater flexibility in advanced and applied contexts. In introductory courses (e.g., CS1/CS2), 8 instructors prohibited AI use entirely, while 12 allowed it with restrictions; none permitted fully open use. Intermediate courses generally allowed limited use of AI tools, often paired with discussions of LLMs, with only one instructor having a strict prohibition. Advanced courses did not prohibit AI, instead integrating it with disclosure requirements, while applied and specialized courses were the most permissive, restricting AI only in specific assessments. Despite these differences, instructors shared a common goal:

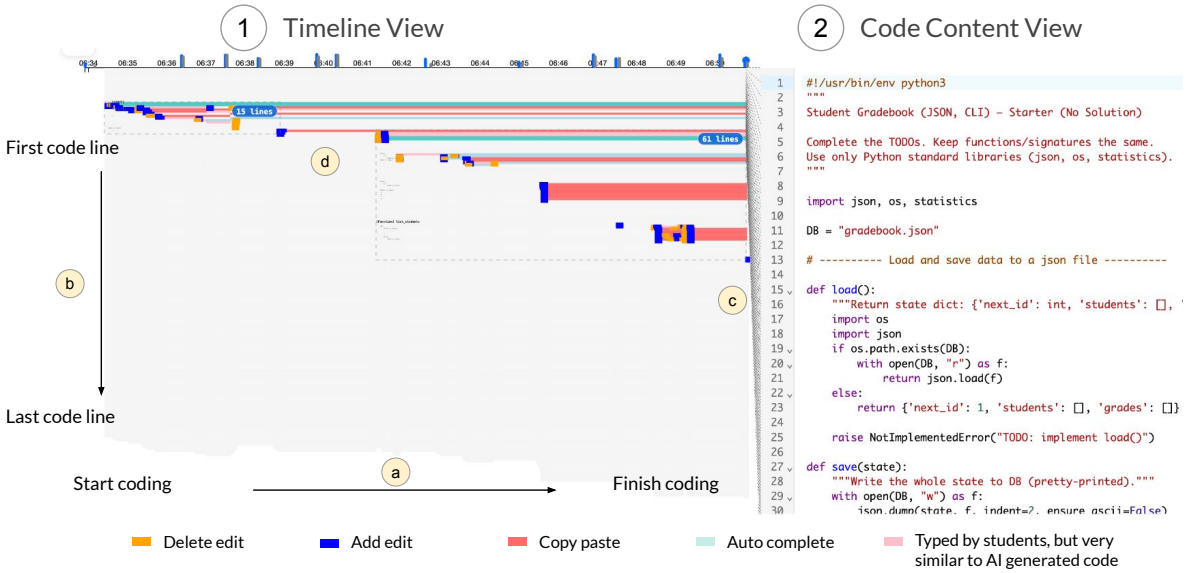

Figure 1: User interface of Editrail. (1) Timeline View: rows represent code lines (b) and time flows horizontally (a). Edits are shown as colored overlays indicating different types of AI involvement (d): red for copy–paste, green for autocomplete, and pink for student-typed code that closely resembles AI-generated output. Indicators also show coding progress from start to finish. (2) Code Content View: aligned program text showing the full source code at the selected point in time (c), enabling instructors to connect timeline annotations with specific code states. When using Editrail, users can follow three steps: (1) get an overview of AI use distribution, (2) identify code regions with potential AI-related concerns, and (3) inspect detailed edits and AI use. The code view shows the content in the student's editor at the selected time, while the timeline view provides a cumulative visualization of the student's progress and AI use. When users zoom in and select an area in the timeline, the code view focuses on the corresponding code region at that time. The code view only shows code content without additional AI color coding beyond syntax highlighting. The AI usage color encoding only appears in the timeline visualization.

ensuring AI use does not undermine learning. They emphasized that students should develop independent problem-solving skills before relying on AI, which may otherwise *"rob them of the processes needed to deepen their understanding."* However, binary policies ('allow' vs. 'ban') are insufficient. Without ways to reliably identify AI misuse, policies were difficult to enforce and sometimes led to unfairness, where compliant students were disadvantaged while others used AI undetected. Even with disclosure requirements, instructors noted that *"students are still not being transparent about their use."* These enforcement challenges make it difficult to interpret suspected AI use without visibility into students' processes. Instructors cannot distinguish among misunderstanding, over-reliance on AI, and a genuine attempt to follow course expectations, motivating DG2 (§3.5 below): enabling targeted, learning-oriented guidance on AI use rather than relying on punitive or policy-driven responses.

3.5 Design Goals

We derived two design goals (DGs) from the need-finding study:

- DG1 **Provide visibility into students' AI usage.** Students' AI usage should be transparent and interpretable to instructors. The system could reveal key aspects of student-AI interaction, such as prompting, adaptation, and authorship balance. This reduces reliance on manual inspection or self-report, giving instructors the awareness they need to understand coding behaviors and intervene when necessary.
- DG2 **Enable targeted, learning-oriented guidance on AI usage.** Visibility alone is not sufficient; instructors also need ways to act on this information. Systems should help instructors connect observed AI usage patterns with context-specific feedback that ensures students learn core concepts rather than bypass them. For example, when instructors see over-reliance on AI suggestions, the system could help them create feedback that addresses misconceptions, scaffolds problem-solving, and encourages productive use of AI.

4 System Design

Guided by our design goals (§3.5), we developed Editrail to help instructors understand students' AI usage by visualizing student-AI interaction in code. Editrail achieves this goal in several ways. First, it allows instructors to monitor AI usage in real-time and identify unproductive usage patterns connected with students' code edits. Second, it enables instructors to create targeted, learning-oriented questions in situ to assess students' understanding. We describe the design of each of these facets of Editrail in more detail below.

4.1 Editrail's UI Overview

Editrail consists of two coordinated views: a Timeline View (Fig. 1.1) and a Code Content View (Fig. 1.2), providing an overview of a student's editing processes and AI involvement. The Timeline View summarizes the student's keystroke and AI usage activity over time, with the x-axis encoding time (Fig. 1.a) and the y-axis representing code line numbers (Fig. 1.b). A light-gray background indicates file length changes as lines are inserted or deleted. Each edit appears as a colored marker positioned by time (x) and affected line (y) (Fig. 1.d). Blue markers indicate insertion edits and orange markers indicate deletion edits. AI-involved edits are overlaid on the timeline (Fig. 1.d): red for copy–paste, green for autocomplete, and pink for student-typed code that closely resembles AI output. The Code Content View shows the program text aligned with the timeline (Fig. 1.2), allowing instructors to inspect code changes at any moment. Selecting a timeline region highlights the corresponding lines in the code view. For longer files, where the timeline must be compressed vertically or the code viewer shows only a subset of lines, Editrail uses a projection indicator to show the visible portion of the code (Fig. 1.c), maintaining alignment between views. These two views reveal students' editing behavior and AI usage patterns, forming the basis for Editrail's interaction techniques described next.

4.2 Monitoring AI Usage in Context

Editrail supports instructors in monitoring AI usage by combining keystroke-level logs, AI interactions, and visual overlays that connect code provenance with code evolution. Below we describe the key interactions and how they support our design goals.

4.2.1 Tracking AI Usage When Students Write Code **DG1**. To increase visibility into AI usage, Editrail first captures students' interactions with AI coding tools and their keystroke-level edits through two lightweight mechanisms: (1) a VS Code extension records keystroke-level edit logs and interactions with AI coding assistants, and (2) a browser extension monitors students' use of ChatGPT. Both forward relevant interaction data to the instructor side. These mechanisms are not intended to be foolproof—motivated and technically knowledgeable students could sidestep detection—but they provide instructors with a practical level of visibility that is otherwise unavailable. The data collected through these mechanisms supported the following interactions.

4.2.2 Viewing Code Edits in Real-Time **DG1**. Editrail extends prior timeline-based visualizations of keystroke edits [53], using data captured by our VS Code extension to show how code evolves over time. As described in §4.1, the timeline maps each edit to its time (x-axis) and file location (y-axis), with insertions and deletions shown as blue and orange markers, respectively (Fig. 2.a). Gray shading represents code length, indicating how the file grows or shrinks over time (Fig. 1.1). As files grow and edits become dense, the timeline compresses; users can zoom in to inspect detailed code changes (Fig. 2.b) [53]. AI-involved edits are overlaid on the timeline (Fig. 1.d), as described next.

4.2.3 Connecting AI Usage with Code Evolution **DG1** **DG2**. Unlike systems that only present raw chat logs or static AI responses, Editrail embeds AI usage directly within the temporal flow of edits (Fig. 1.d) to help instructors identify different AI usage patterns and provide targeted help. At the top of the timeline, Editrail uses colored bars to represent messages sent from the student and AI, where blue represents the student and gray represents the AI (Fig. 3.a). The bar's position on the x-axis represents when the message was sent, and its height represents the message's word count. When hovering over a bar, users can see the actual message content. When a student adopts an AI suggestion, instructors can see exactly when it entered the code (Fig. 3.c), how it evolved, and whether students refined, deleted, or repeatedly reverted it (Fig. 3.d). The code lines edited are shown as colored overlays (Fig. 3.c–d). Users can also locate changes in the code content view by clicking the change indicators; the code content view on the right then jumps to that line of code (Fig. 3.b). This contextualization reveals whether students meaningfully integrated AI code or relied on it uncritically. For example, patterns such as rapid copy–paste followed by minimal adaptation, or cycles of deletion and reinsertion, highlight potential misuse that would otherwise be hidden in the final submission.

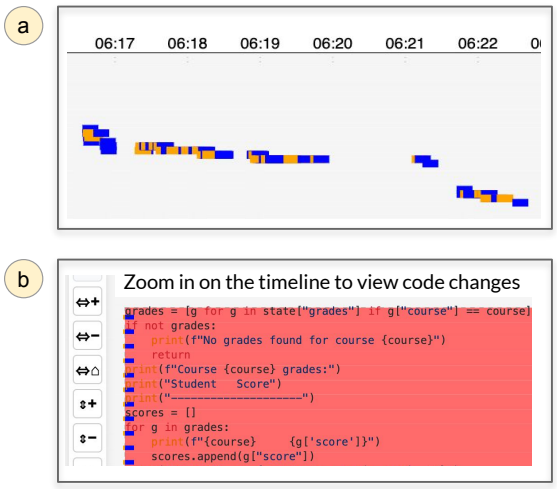

Figure 2: Details of Editrail's timeline interaction. (a) Keystroke-level edit indicators shown along the timeline, where blue ones represent insertion and orange ones represent deletion edits. (b) Zoomed-in timeline view revealing the exact code content associated with individual change indicators for detailed inspection.

4.2.4 Differentiating AI-Generated vs. Student-Written Code **DG1** **DG2**. Editrail distinguishes AI-generated code from student-authored edits, categorizing AI-originated code into three types: (1) copy-pasted from AI suggestions (Fig. 4.a), (2) inserted via autocomplete (e.g., GitHub Copilot) (Fig. 4.b), and (3) typed manually but highly similar to AI output (Fig. 4.c). Each is highlighted in a distinct color—red (copy–paste), green (autocomplete), and pink (AI-like typed code) (Fig. 1.d). AI-originated code appears as trails in the edit history, with subsequent student edits overlaid incrementally, making authorship transparent and enabling instructors to assess AI involvement and student contribution without manually comparing logs or chat histories.

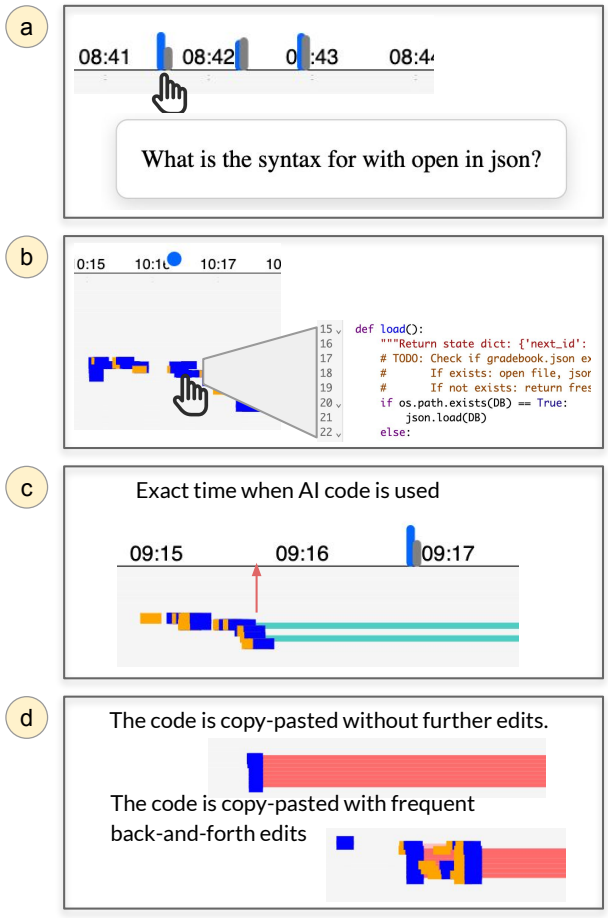

Figure 3: Details of Editrail's interaction of monitoring AI usage. (a) AI–student dialogue shown at the exact point of interaction. (b) Timeline-to-code projection aligning edits with program text. (c) Precise timing of AI code usage. (d) Code pasted from AI without further edits and code pasted from AI with frequent back-and-forth edits by the student.

4.3 Design Probe: Targeted Question Generation

Through this speculative feature, we explored whether instructors might benefit from creating targeted, learning-oriented questions directly on the timeline visualization. When an instructor clicks a region of code edits (e.g., where AI suggestions were introduced and not validated, Fig. 5.a), Editrail shows a pop-up window where they can create context-aware questions to assess students' understanding—either multiple-choice or open-ended (Fig. 5.b). The code area shows students' code at the clicked timestamp, highlighting AI-originated code in yellow (Fig. 5.c). Instructors can optionally adjust constraints for the LLM prompt (Fig. 5.d) to tailor the questions. After clicking 'create question' (Fig. 5.e), Editrail generates an example question (Fig. 5.f) and an expected answer (Fig. 5.g). Instructors can modify these artifacts and send them to students (Fig. 5.h). This lets us observe what kinds of questions they valued and how they imagined aligning them with debugging or conceptual-learning goals. This feature was not fully evaluated; rather, it serves to spark discussion about how question-generation tools might be integrated into timeline-based visualizations.

4.4 Implementation Details

Editrail's implementation contains two key components: a VSCode extension for students and a web application for instructors. Our implementation of Editrail is open-source[1].

4.4.1 VSCode Extension for Students. The VSCode extension streams coding history and AI chat interactions, and detects whether code originates from AI or students. It builds on an existing tool that collects keystroke-level edits from the University of Michigan[2], which we extended to also capture students' GitHub Copilot Chat messages, including the prompt, the AI's responses, and the context sent when prompting LLMs. Every keystroke, edit, and chat is streamed to the instructor's website via WebSocket, and a recording function saves students' edit and chat logs to a JSON file. VSCode also communicates with the instructor's website to forward questions from instructors to students and answers back.

4.4.2 Instructor's Web Application. The instructor web application consists of a React/TypeScript frontend and a Node.js backend. The frontend visualizes students' AI usage and code edits through (i) a timeline view with AI attribution highlighting and (ii) a synchronized code view, and supports interactions such as clicking, playback, and question creation, allowing instructors to generate and send questions to students. The backend is a Node.js server using Socket.IO that manages session state and real-time streaming. The GPT-3.5-turbo model is used for generating questions.

5 Students' Coding Dataset

To get a realistic dataset of students' coding histories, we conducted a study to collect students' coding history data on two Python programming problems. This dataset is used for evaluating Editrail. We also publish the dataset as a contribution to this paper[3].

5.1 Recruitment

We recruited undergraduate and graduate students (18+) from computer science and information science programs with Python experience. To capture diverse coding and AI usage patterns, including those in more complex tasks, we did not restrict participants to novice programmers. Instead, we sought participants with diverse experience so the study could capture a broad spectrum of coding and AI-usage behaviors, including those that arise in more complex codebases. The study included 20 participants (11 men, 8 women, 1 non-binary) with diverse roles and experience levels, ranging from less than 3 months to 9 years (most 1–7 years).

5.2 Method

Each participant completed two Python programming tasks in counterbalanced order, yielding 20 coding histories per task. For each, we collected keystroke coding logs, AI chat messages (GitHub Copilot),

[1]https://github.com/AshleyZG/Editrail
[2]https://github.com/educational-technology-collective/vscode-telemetry
[3]https://github.com/AshleyZG/Editrail-data

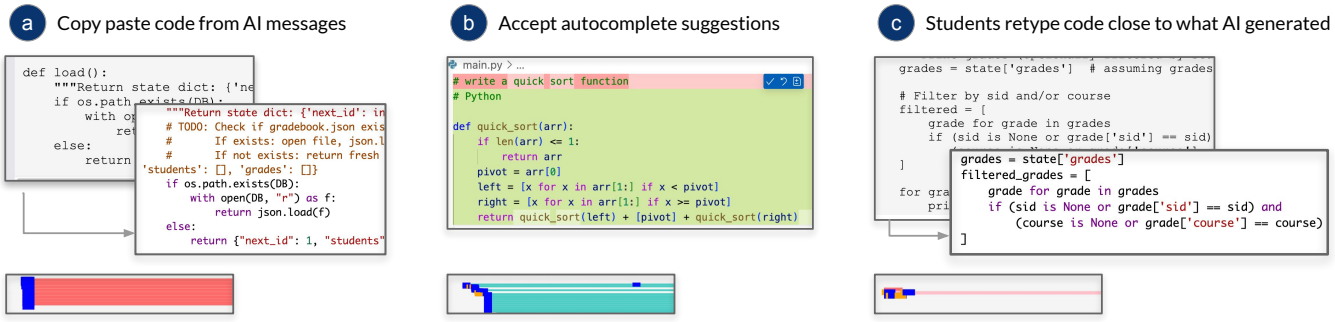

Figure 4: Three cases of code that originates from AI. (a) The user copies and pastes code directly from an AI agent (e.g., ChatGPT or Copilot chat). (b) The user accepts an autocomplete suggestion (e.g., from Copilot's in-editor hints). (c) The user types code that is very similar to something that was part of their conversation with an AI agent. The AI usage color encoding appears only in the timeline visualization. The code view shows code content with syntax highlighting only.

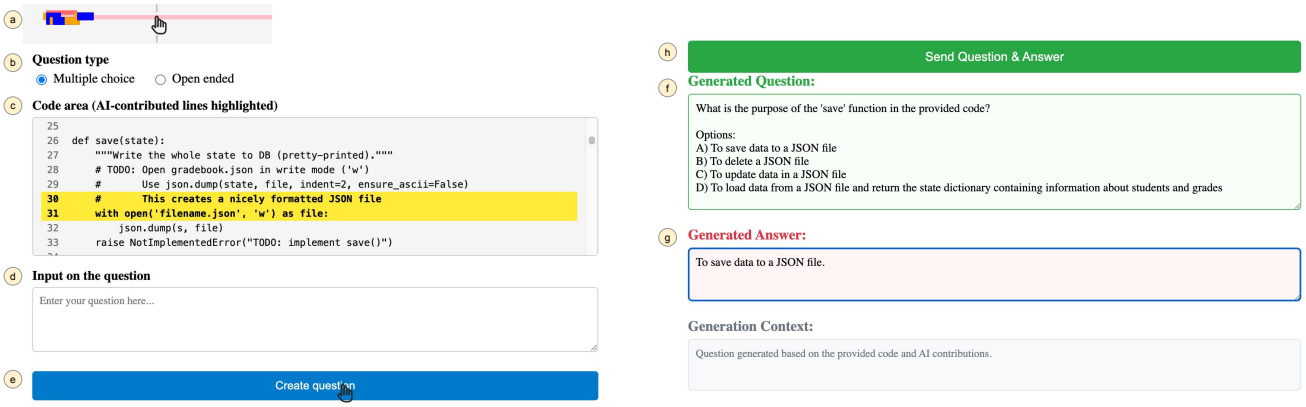

Figure 5: User interface of Editrail's question creation. (a) Instructors select a point on the coding timeline to create questions. (b) Question type options (multiple choice or open-ended). (c) Code area highlighting AI-contributed lines in student's code. (d) Input field for customizing the question. (e) Button to generate the question. (f) Example of a system-generated question based on the selected code. (g) Generated answer for the question. (h) Button to send the question to students.

and test results per run. To ensure diverse coding and AI usage patterns, 10 participants used AI without restrictions, while the other 10 were instructed to use AI only when stuck. These conditions were independent of experience level, as our goal was to elicit varied AI-usage patterns rather than compare groups by background. The study was conducted virtually on Zoom, and each participant received a $15 USD Amazon gift card.

5.3 Data Collection Tool

We built a VSCode[4] extension to collect the coding dataset (Section 4.4.1). The extension captures three types of data: (1) keystroke-level coding histories, (2) chat messages with AI (e.g., GitHub Copilot Chat[5]), and (3) execution results from test files. After each study session, the collected data is stored in JSON format. The VSCode extension was deployed on GitHub Codespaces[6], a cloud-based IDE. Participants used the web-based code editor without installing it on their machines. In addition, for each session, we recorded participants' screens as a redundant source of their coding activities and AI interactions. These recordings served as a supplementary resource to validate and proof-check the coding history logs.

5.4 Tasks

To evaluate Editrail on complex and scalable code, we collected code histories from multi-function Python tasks adapted from common coding problems. Both tasks can be solved within 150 to 300 lines of code. Both tasks were of comparable difficulty, requiring use of lists and dictionaries, JSON processing, statistical computation, and string formatting. Each included starter code with descriptions, TODOs, and a test script for feedback. Tasks were limited to 20 minutes and did not require completion, as the goal was to capture diverse coding behaviors (e.g., iteration, debugging, prompting, and AI-assisted edits). Participants iteratively developed solutions toward passing tests, enabling collection of rich interaction data for evaluating Editrail's visualizations.

[4]https://code.visualstudio.com
[5]https://code.visualstudio.com/docs/copilot/chat/getting-started-chat
[6]https://github.com/features/codespaces

Task 1 (T1) A command-line gradebook system that manages students and grades stored in JSON, supporting add/list operations and generating per-student and per-course statistics.

Task 2 (T2) A hospital appointment system that manages doctors, patients, and appointments with JSON storage, including add/list operations, constraint validation, and conflict-free booking.

5.5 Results

We collected 20 coding histories per Python problem, each lasting 20 minutes. For **T1**, solutions ranged from 188 to 319 lines of code ($M = 229$, $SD = 32.89$), with 51 to 1679 keystroke edits ($M = 788.40$, $SD = 449.78$), 0 to 15 test runs, and 3 to 39 AI chats per participant. For **T2**, solutions ranged from 198 to 306 lines of code ($M = 239.25$, $SD = 31.11$), with 37 to 2610 keystroke edits ($M = 769.42$, $SD = 611.98$), 0 to 15 test runs, and 3 to 42 AI chats per participant. Across all 40 histories, 62.67% of edits were human-authored, 14.91% were generated via GitHub Copilot autocomplete, 7.25% were human edits of AI-generated code, and 1.31% were direct copy-pastes; the rest were file actions (save/open/close). Reliance on AI varied widely across participants: the proportion of AI-contributed edits ranged from 4.47% to 98.70% ($M = 23.22\%$, $SD = 21.56\%$), as expected given the two different AI-usage instructions. Appendix D shows an anonymized example of the collected dataset.

6 Editrail User Study

Editrail is designed to give instructors better visibility into students' AI usage during programming and to support targeted guidance when misuse occurs. We conducted a within-subject study to evaluate Editrail's effectiveness in monitoring AI usage and guiding interventions. To ensure generalizability beyond short examples, the study used the long-code dataset we collected. Since no existing tools support monitoring students' AI usage, we built a baseline system for comparison. The baseline was implemented as a code editor paired with students' chat histories with AI, allowing participants to review both code and messages side by side. Participants used both Editrail and the baseline to answer quiz-style questions about students' AI usage. This design helped us assess how much the visualization improved instructors' ability to detect and reason about AI misuse, as well as its limitations. The study was reviewed and exempted by the Institutional Review Board (IRB).

6.1 Method

6.1.1 Recruitment. Because Editrail's end users are programming instructors, we reached out to students and instructors from the authors' universities, who had experience teaching Python or were at a minimum proficient in Python. During a screening session, participants indicated their prior experience teaching and using Python. In total, we recruited 12 participants (three women and nine men), with Python experience ranging from 4 to 10 years. Most held teaching roles (Graduate Student Instructors, Teaching Assistants, or Tutors), while others included an engineer and a graduate student researcher (Table 3).

6.1.2 Baseline System. Since no widely used tools exist for monitoring students' AI usage in programming, we developed a baseline system (Figure 6) that presents students' final code submissions alongside GitHub Copilot Chat histories. Figure 6 shows a screenshot of the baseline system's interface, using one student's view as an example. It has two parts: the left side shows the student's chat histories with GitHub Copilot Chat, and the right side shows the final code submission. Although prior work has explored understanding student–AI interactions [8], it focuses on conversations with AI, such as topics and prompting patterns, rather than how students use generated code. As this differs from our focus, we did not adopt it as a baseline. We chose this design because chat histories provide direct evidence of how students prompt and revise AI queries, capturing interactions not visible in code alone. In contrast, Editrail emphasizes code edits and AI contributions during the coding process. These views in the baseline system offer complementary perspectives: chat logs reveal dialogue and reasoning, while visualizations show process-level changes. This baseline enables comparison between visualizations and raw chat logs, allowing us to examine their respective strengths, limitations, and interactions, and to inform how future systems can balance these information sources to support instructors.

6.1.3 Study Setup. The study was conducted remotely via Zoom using a within-subjects format where participants used both Editrail and the baseline system. We counterbalanced system and task order (**T1** and **T2**). Each condition included 10–20 minutes of training on the system, tasks, and quiz format, followed by 20 minutes to answer questions about students' AI usage. The quiz questions required participants to analyze coding histories and identify specific AI usage patterns, such as AI-generated or edited functions, levels of reliance, and strategy shifts over time. These questions assess participants' ability to infer AI involvement and authorship from coding histories. After each condition, participants completed a survey about their experience using the system. At the end of each study, we conducted a reflective survey and interview to compare the two systems. We encouraged participants to ask any questions about the usage of both systems. Each study lasted about 70 minutes.

6.1.4 Data Collection and Metrics. During the study, we recorded participants' screens, capturing quiz responses, think-aloud audio, and interactions with both systems. We also collected their answers to the post-study survey and follow-up interviews. Two researchers were present in each session. We evaluated participants' quiz performance using F-score for multiple-choice questions:

$$\frac{2 \times True\ Positive}{2 \times True\ Positive + False\ Positive + False\ Negative}$$

For single-choice questions, accuracy was binary (1 if correct, 0 otherwise). Open-ended quiz responses were coded independently by three authors using a collaboratively developed scheme. We also analyzed screen recordings to examine interaction patterns, including time spent and information-seeking strategies, and conducted thematic analysis on survey and interview data. We used paired t-tests to compare quiz accuracy and workload (time spent answering quiz questions) across the two systems. Normality was verified for both measures (Shapiro–Wilk tests, $p > 0.05$). Figure 7 presents results from the comparison survey.

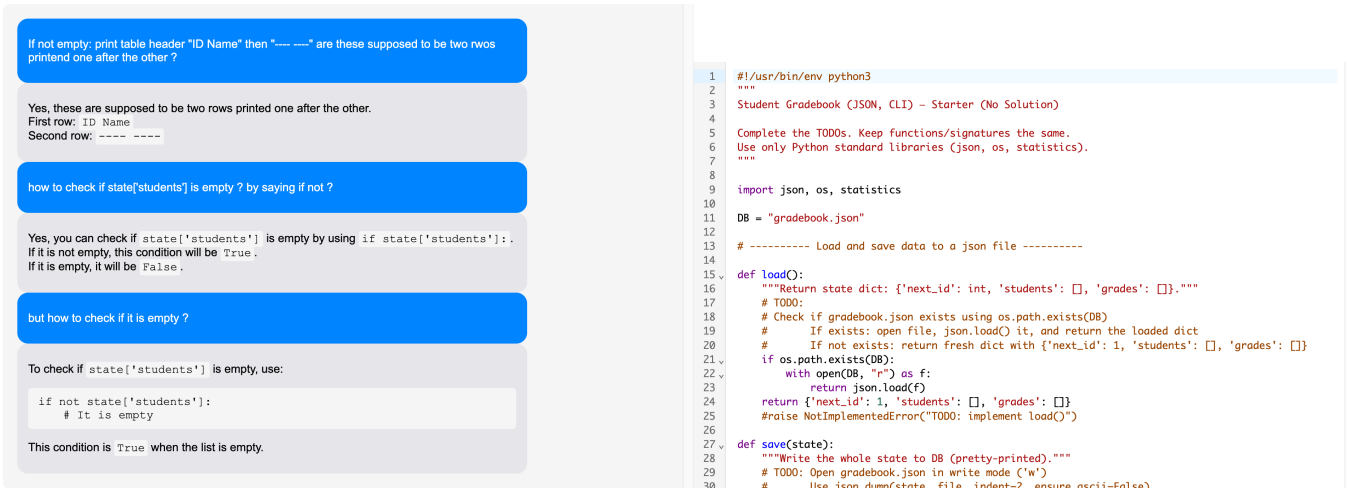

Figure 6: Baseline system for the user study. It represents conversation turns with AI agents (left) and code states (right).

6.2 Results

6.2.1 Participants understand students' AI usage more accurately using Editrail than the baseline, without increasing analysis time. We compared participants' accuracy on quiz questions, which required identifying students' AI usage, how AI-generated code was adapted, and how much of the final code students wrote. Accuracy was computed per participant by averaging F-scores (multiple-choice) and binary accuracy (single-choice), and was significantly higher with Editrail ($M = 0.79$, $SD = 0.13$) than the baseline ($M = 0.31$, $SD = 0.14$, $p < 0.00001$). This aligns with the self-report and comparison survey results (Table 4, Figure 7): participants rated Editrail as more helpful for (1) understanding AI usage patterns, (2) identifying students needing intervention, and (3) detecting unproductive usage conflicting with pedagogical goals. However, this difference was smaller in self-reports than in quiz results: self-reported understanding did not differ significantly across conditions (Table 4), yet objective quiz accuracy improved significantly with Editrail. This suggests raw code and chat histories may create a false sense of confidence, whereas Editrail provides more accurate and actionable insights (see Section 6.3.2).

We also measured time spent on the quiz. Participants took slightly longer with the baseline ($M = 841.75$ seconds, $SD = 223.18$) than with Editrail ($M = 815.75$ seconds, $SD = 221.75$), but the difference was not significant ($p > 0.05$). Two factors may explain this. First, participants noted that Editrail has a learning curve, whereas the baseline's chat logs are immediately interpretable. Second, although the baseline required reading lengthy chat histories, some participants found this tedious and skimmed or skipped content, which reduced the time they spent despite the larger amount of material to review.

6.3 System Usability and Study Insights

6.3.1 Editrail enables instructors to inspect AI usage patterns and apply their own pedagogical criteria when evaluating them. Participants found Editrail's visualizations enabled quick understanding of AI usage, including when and where AI was used, how generated code was adapted, and how students edited it. Timeline views and visual indicators were *"helpful in seeing whether students needed intervention"* (P12). Beyond identifying AI-generated edits, participants could assess engagement with AI output. P12 noted that they examine whether students edit code after AI-generated or copy-pasted content, as such edits indicate continued critical thinking and personal contribution. They also consider the quantity and distribution of AI-influenced lines: widespread edits suggest heavy reliance on AI, whereas localized changes indicate targeted assistance when students are stuck.

However, participants had diverse opinions on what constitutes productive versus unproductive AI use, even when interpreting the same visual patterns. Some viewed heavy reliance on AI (e.g., large AI-generated blocks) as productive, interpreting it as the student assembling a working solution; others saw the same pattern as unproductive, reflecting limited learning. Similarly, back-and-forth edits on AI-generated code were read either as exploration and validation or as struggle and inefficiency. Overall, judgments of productivity varied with how instructors conceptualized AI's role in learning. This aligns with the formative study (Section 3.3): the wide variation in students' AI usage leads to different interpretations of whether it is productive, and highlights the need for mechanisms to quickly assess students' understanding of specific concepts. Editrail provides visibility into fine-grained AI use, enabling instructors to monitor and reason about AI usage patterns and then apply their own pedagogical goals to judge whether it is productive.

6.3.2 When using the baseline, participants believed they had a better understanding of students' AI usage than they actually demonstrated. While participants showed significantly lower quiz accuracy with the baseline, several reported a good understanding of AI usage using the baseline with no statistical significance in their self-report results (Table 4). We identified reasons for this gap between actual and perceived understanding of AI usage. Participants relied on chat histories as direct evidence of prompting behavior, for example, whether students requested explanations or submitted

full problem descriptions. P12 noted that prompting patterns reveal learning quality: copy-pasting instructions to obtain full solutions suggests unproductive use, whereas asking for help with syntax, understanding prompts, or small code segments indicates more productive engagement. While chat history is also provided in Editrail, it was less prominent and rarely used. Participants had two main interaction patterns with the baseline: carefully reading all messages or skimming and guessing. The latter often stemmed from cognitive overload, as text-heavy conversations were difficult to process; P5 found it *"hard to distill insightful points"* from them. Additionally, participants struggled to link chat messages to code changes. P6 noted that without visual cues (e.g., color or code mapping), the baseline requires manual comparison, making it difficult to identify AI-influenced segments and adaptations.

6.3.3 Editrail helps instructors identify when to intervene and which students need intervention. Most participants agreed that Editrail better supports identifying when and whom to intervene. With Editrail, participants could easily see when students copied and pasted code, accepted autocomplete suggestions, or typed in code that closely resembled AI-generated code. P9 noted that visual cues (e.g., colored lines indicating AI use, AI-like code, or copy-paste) reveal where and how extensively AI is used within the code. The visualization also enables rapid intervention: instructors can directly identify AI-contributed segments and respond by giving hints, assessing understanding, or sending quizzes. As P8 noted, this helps identify students using AI unproductively and tailor support based on their level of understanding.

6.4 Preliminary Evaluation on Targeted Question Creation

As an early exploration, we implemented a proof-of-concept question creation feature (described in §4.3) to examine how instructors might use timeline interactions to assess students' understanding. Although not fully evaluated in classrooms, we collected preliminary feedback to inform future design. Participants expressed mixed opinions. Some found the feature useful for assessing understanding of AI-generated content (P2, P5, P8, P11, P12); P2 noted it could help identify gaps and provide targeted resources. Others raised concerns about its reliability and practicality. P12 questioned whether student responses accurately reflect understanding, noting confounds such as misreading, guessing, or differing interpretations. Participants also questioned the quality of generated questions, specifically whether (1) they targeted relevant code regions (P1, P12), (2) their difficulty was appropriate (P6, P9), and (3) they required manageable effort to answer (P4). Additionally, P10 preferred addressing misunderstandings directly through instruction rather than sending extra questions, and doubted such questions could distinguish genuine understanding from AI over-reliance.

7 Discussion

7.1 Implications for Learning to Code with AI

7.1.1 Bridging gaps in visibility. A key contribution of Editrail is closing the visibility gap between students' AI use and instructors' observations. Traditional approaches—such as reviewing final code or relying on self-reports—obscure how students prompt AI, accept or reject suggestions, and iteratively edit. Even with chat histories, our findings show instructors struggle to understand how AI-generated code is adapted. By revealing when and where AI is used and how generated code is integrated, Editrail provides a clearer view of the learning process. This transparency turns a "black box" into a structured narrative, enabling instructors to distinguish productive use (e.g., scaffolding, debugging) from problematic patterns (e.g., blind copy-paste, over-reliance).

7.1.2 Pedagogical value. Our preliminary findings showed that, with greater visibility, instructors can better align their interventions with their pedagogical philosophies. For those who adopt a more cautious strategy towards AI, Editrail provides the evidence needed to intervene when students bypass essential learning steps. For instructors who encourage exploration with AI, the system offers a way to ensure that students still engage meaningfully with concepts rather than deferring all reasoning to the model. In both cases, Editrail demonstrated the potential capability of enabling instructors to personalize feedback, for example, by encouraging a struggling student to slow down and validate AI output, or by supporting advanced students in using AI to explore alternative approaches. Instead of one-size-fits-all advice, instructors can tailor their guidance to individual students' behaviors and goals.

7.1.3 Policy-making support. Another implication is that Editrail can inform policy development at the course or institutional level. As we found in our formative study, many current AI usage policies are binary because instructors lack the means to monitor nuanced interactions (Section 3.4). By providing a fine-grained record of how AI is used in context, Editrail creates the foundation for more flexible and context-aware policies. For instance, policies could distinguish between acceptable uses of AI for targeted help and unacceptable uses for generating entire solutions. Over time, aggregated data from systems like Editrail could reveal broader patterns across cohorts, helping educators refine policies that balance academic integrity with the pedagogical benefits of AI.

Overall, by making AI use more visible, aligning feedback with teaching goals, and informing more nuanced policy considerations, Editrail can contribute to how programming instructors respond to and integrate generative AI in CSEd.

7.2 Editrail's Abstraction Strategy

Editrail combines a timeline view with color coding of AI use to provide a high-level abstraction of students' AI-assisted coding behavior, reflecting several design considerations. First, programming is a dynamic process rather than a static snippet, so how students interact with AI and adapt its code matters more than the final code itself. By linking the timeline with color coding, instructors can quickly see where and when AI is used and how that use changes over time. Second, keystroke-level data is too detailed for instructors to monitor directly. Editrail therefore abstracts it to the line level and ties AI use to each line's content changes, helping instructors make sense of the coding process and AI use in context. This abstraction also supports interaction with targeted code regions, as shown in the QA feature, where instructors can ask questions about a region and assess a student's understanding of specific concepts. The strategy also has limitations. It does not

encode the semantic meaning of chat messages, a promising direction for future work; for example, the system could summarize a student's prompting strategy or categorize the questions they ask. It is also designed for per-student monitoring and does not naturally aggregate information across students.

7.3 Design Scope and Scalability

Scalability is an important and multi-faceted challenge when designing tools for programming education, especially for introductory-level courses. To browse large code files and interaction histories, Editrail uses "semantic zooming" [53] to scale to large files and long timelines. This effectively "zooms out" from detailed edits to higher-level coding patterns as needed, similar to the mini-maps found in many code editors.

Editrail is designed to support understanding individual students' interactions with AI rather than providing aggregate classroom analytics. We see these as complementary layers: aggregate views can help identify which students or assignments warrant closer attention, while Editrail's individual timeline view helps instructors interpret what actually happened in a particular student's coding process. Editrail currently addresses the second layer; future work should explore class-level summaries of AI use before allowing instructors to drill down into individual histories.

Editrail is also deliberately not designed as an enforcement tool. As found in the formative study, students could inaccurately report their AI usage. Instead of automatically interpreting this as intentional obfuscation, this might reflect what recent research has shown: people can have difficulty accurately remembering when and how AI contributed within mixed human-AI workflows, making retrospective reporting challenging [55]. Further, research on nudges and default effects shows that people are substantially more likely to behave in a desired way when doing so is the easy or default option. By capturing fine-grained AI use at the point of interaction rather than relying on students' retroactive self-reports, Editrail may reduce an important source of reporting error.

While preventing intentional obfuscation by determined students is a relevant problem, it is one that we deliberately consider outside Editrail's scope. Editrail is designed to make student-AI interactions easier to capture and understand rather than to serve as an enforcement mechanism. Stronger enforcement approaches, such as proctoring software or controlled environments, already address this complementary challenge, albeit with tradeoffs around student privacy, autonomy, and trust.

7.4 Ethical Implications and Privacy in Editrail

As Editrail tracks students' coding progress and AI interactions (e.g., GitHub Copilot, ChatGPT), it raises privacy concerns, including monitoring beyond coursework. Students may hesitate to share their process, and sensitive data may reinforce bias in teaching [30, 38]. To mitigate this, Editrail allows users to disable edit history and AI tracking in VS Code at any time, supports local deployment via private cloud or LLM APIs, and can restrict access to in-editor AI interactions only. The collected keystroke dataset could also raise privacy concerns. We anonymized identifiable information; prior work further explored ways to modify data so that keystroke-latency-based identification is no longer accurate [27]. This work focuses on instructors' perspectives; future work should examine student experiences through classroom deployments and explore transparency features such as in-editor monitoring notifications.

7.5 Limitations and Future Work

Our work has limitations in both design and evaluation. First, while Editrail aims to help instructors identify unproductive AI usage and provide personalized guidance, we evaluated it only with instructors; future work should examine its use in real classrooms. Our study also included only two Python tasks of similar complexity, limiting generalizability across tasks, difficulty levels, and languages. Preliminary findings on the QA feature suggest opportunities for designs that better assess understanding and encode interpretable metrics of students' reasoning. Our study involved 12 instructors in controlled settings; larger, in-situ deployments may reveal additional challenges, and differences across AI tools (e.g., ChatGPT, Cursor AI, Codex) may also influence student strategies.

Second, Editrail currently visualizes AI contributions to code but does not capture students' understanding. Extending it with mechanisms to assess understanding could reveal how mental models evolve, for example, by showing a shift from heavy AI reliance to increased comprehension and independent code refinement.

Third, instructors may disagree on what constitutes "productive" versus "unproductive" AI use, even with the same visualizations. Future work could explore standardizing these definitions or enabling instructors to customize criteria and metrics.

Finally, Editrail focuses on individual coding histories; future work should support aggregated views across students and assignments to reveal broader patterns (e.g., common AI-supported tasks or misconceptions). Such visualizations could surface recurring AI-generated snippets and intervention points, complementing individual views and enabling class-level instructional decisions. More broadly, future work could explore other aspects of the keystroke coding dataset we provide: in this paper we focused on the quantity and contribution of students' AI use, but other metrics of students' agency in AI-assisted programming—such as how AI-generated code survives through the coding process—remain open.

8 Conclusion

In this work, we conducted a survey study to examine instructors' practices, pain points, and needs around understanding students' AI usage in programming and providing guidance. We introduced Editrail to address a major challenge from the survey: the visibility gap between how students actually interact with AI and what instructors observe from final submissions. Editrail bridges this gap by visualizing AI usage as trails in students' code edits, enabling instructors to distinguish productive from unproductive patterns, create in-situ assessments, and align AI use with pedagogical goals. Our evaluation showed that Editrail let instructors identify AI usage with twice the accuracy of a baseline and more effectively determine when and with whom to intervene. By highlighting how students adapt AI-generated code, the visualization also provides rich meta-information about their problem-solving approaches, reducing the effort to understand text-heavy AI dialogues. Finally, this work offers design insights for facilitating feedback loops among instructors, students, and AI.

References

[1] Rudaiba Adnin, Atharva Pandkar, Bingsheng Yao, Dakuo Wang, and Maitraye Das. 2025. Examining Student and Teacher Perspectives on Undisclosed Use of Generative AI in Academic Work. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. 1–17.

[2] Alex Aiken. 2024. MOSS: A Measure of Software Similarity. https://theory.stanford.edu/~aiken/moss/. Accessed: March 29, 2026. Originally developed 1994.

[3] Matin Amoozadeh, Daye Nam, Daniel Prol, Ali Alfageeh, James Prather, Michael Hilton, Sruti Srinivasa Ragavan, and Amin Alipour. 2024. Student-AI Interaction: A Case Study of CS1 students. In *Proceedings of the 24th Koli Calling International Conference on Computing Education Research*. 1–13.

[4] Rishabh Balse, Viraj Kumar, Prajish Prasad, and Jayakrishnan Madathil Warriem. 2023. Evaluating the quality of llm-generated explanations for logical errors in cs1 student programs. In *Proceedings of the 16th Annual ACM India Compute Conference*. 49–54.

[5] Brett A Becker, Paul Denny, James Finnie-Ansley, Andrew Luxton-Reilly, James Prather, and Eddie Antonio Santos. 2023. Programming is hard-or at least it used to be: Educational opportunities and challenges of ai code generation. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 500–506.

[6] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative research in psychology* 3, 2 (2006), 77–101.

[7] Chaoran Chen, Zhiping Zhang, Zeya Chen, Eryue Xu, Yinuo Yang, Ibrahim Khalilov, Simret A Gebreegziabher, Yanfang Ye, Ziang Xiao, Yaxing Yao, Tianshi Li, and Toby Jia-Jun Li. 2026. Comparing Human Oversight Strategies for Computer-Use Agents. arXiv:2604.04918 [cs.HC] https://arxiv.org/abs/2604.04918

[8] Zixin Chen, Jiachen Wang, Meng Xia, Kento Shigyo, Dingdong Liu, Rong Zhang, and Huamin Qu. 2024. StuGPTViz: A Visual Analytics Approach to Understand Student-ChatGPT Interactions. *IEEE Transactions on Visualization and Computer Graphics* (2024).

[9] Paul Denny, Stephen MacNeil, Jaromir Savelka, Leo Porter, and Andrew Luxton-Reilly. 2024. Desirable characteristics for ai teaching assistants in programming education. In *Proceedings of the 2024 on Innovation and Technology in Computer Science Education V. 1*. 408–414.

[10] Kasra Ferdowsi, Ruanqianqian 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*. 1–8.

[11] James Finnie-Ansley, Paul Denny, Brett A Becker, Andrew Luxton-Reilly, and James Prather. 2022. The robots are coming: Exploring the implications of openai codex on introductory programming. In *Proceedings of the 24th Australasian computing education conference*. 10–19.

[12] Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. 2015. OverCode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)* 22, 2 (2015), 1–35.

[13] Gradescope. 2024. Gradescope: Save Time Grading. https://www.gradescope.com. Accessed: March 29, 2026. Gradescope is a product of Turnitin.

[14] Philip J Guo. 2015. Codeopticon: Real-time, one-to-many human tutoring for computer programming. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. 599–608.

[15] Mohanad Halaweh. 2023. ChatGPT in education: Strategies for responsible implementation. *Contemporary educational technology* 15, 2 (2023).

[16] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. 2017. Writing reusable code feedback at scale with mixed-initiative program synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*. 89–98.

[17] Amber Horvath, Andrew Macvean, and Brad A Myers. 2024. Meta-manager: A tool for collecting and exploring meta information about code. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*. 1–17.

[18] Xinying Hou, Zihan Wu, Xu Wang, and Barbara J Ericson. 2024. Codetailor: Llm-powered personalized parsons puzzles for engaging support while learning programming. In *Proceedings of the Eleventh ACM Conference on Learning@ Scale*. 51–62.

[19] Breanna Jury, Angela Lorusso, Juho Leinonen, Paul Denny, and Andrew Luxton-Reilly. 2024. Evaluating llm-generated worked examples in an introductory programming course. In *Proceedings of the 26th Australasian computing education conference*. 77–86.

[20] Minsuk Kahng, Ian Tenney, Mahima Pushkarna, Michael Xieyang Liu, James Wexler, Emily Reif, Krystal Kallarackal, Minsuk Chang, Michael Terry, and Lucas Dixon. 2024. LLM Comparator: Interactive Analysis of Side-by-Side Evaluation of Large Language Models. *IEEE Transactions on Visualization and Computer Graphics* (2024).

[21] Majeed Kazemitabaar, Justin Chow, Carl Ka To Ma, Barbara J Ericson, David Weintrop, and Tovi Grossman. 2023. Studying the effect of AI code generators on supporting novice learners in introductory programming. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–23.

[22] Majeed Kazemitabaar, Oliver Huang, Sangho Suh, Austin Z Henley, and Tovi Grossman. 2024. Exploring the Design Space of Cognitive Engagement Techniques with AI-Generated Code for Enhanced Learning. *arXiv preprint arXiv:2410.08922* (2024).

[23] Majeed Kazemitabaar, Runlong Ye, Xiaoning Wang, Austin Zachary Henley, Paul Denny, Michelle Craig, and Tovi Grossman. 2024. Codeaid: Evaluating a classroom deployment of an llm-based programming assistant that balances student and educator needs. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–20.

[24] Kenneth R Koedinger, Jihee Kim, Julianna Zhuxin Jia, Elizabeth A McLaughlin, and Norman L Bier. 2015. Learning is not a spectator sport: Doing is better than watching for learning from a MOOC. In *Proceedings of the second (2015) ACM conference on learning@ scale*. 111–120.

[25] Kenneth R Koedinger, Elizabeth A McLaughlin, Julianna Zhuxin Jia, and Norman L Bier. 2016. Is the doer effect a causal relationship? How can we tell and why it's important. In *Proceedings of the sixth international conference on learning analytics & knowledge*. 388–397.

[26] Sam Lau and Philip Guo. 2023. From" Ban it till we understand it" to" Resistance is futile": How university programming instructors plan to adapt as more students use AI code generation and explanation tools such as ChatGPT and GitHub Copilot. In *Proceedings of the 2023 ACM Conference on International Computing Education Research-Volume 1*. 106–121.

[27] Juho Leinonen, Petri Ihantola, and Arto Hellas. 2017. Preventing keystroke based identification in open data sets. In *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*. 101–109.

[28] Qianou Ma, Weirui Peng, Chenyang Yang, Hua Shen, Ken Koedinger, and Tongshuang Wu. 2025. What Should We Engineer in Prompts? Training Humans in Requirement-Driven LLM Use. *ACM Trans. Comput.-Hum. Interact.* 32, 4, Article 41 (Aug. 2025), 27 pages. https://doi.org/10.1145/3731756

[29] Qianou Ma, Hua Shen, Kenneth Koedinger, and Sherry Tongshuang Wu. 2024. How to teach programming in the ai era? using llms as a teachable agent for debugging. In *International Conference on Artificial Intelligence in Education*. Springer, 265–279.

[30] Paola Medel and Vahab Pournaghshband. 2017. Eliminating gender bias in computer science education materials. In *Proceedings of the 2017 ACM SIGCSE technical symposium on computer science education*. 411–416.

[31] Francesc Pedro, Miguel Subosa, Axel Rivas, and Paula Valverde. 2019. Artificial intelligence in education: Challenges and opportunities for sustainable development. (2019).

[32] Weirui Peng, Yinuo Yang, Zheng Zhang, and Toby Jia-Jun Li. 2025. Glitter: An ai-assisted platform for material-grounded asynchronous discussion in flipped learning. In *Proceedings of the 38th Annual ACM Symposium on User Interface Software and Technology*. 1–22.

[33] Tung Phung, Victor-Alexandru Pădurean, Anjali Singh, Christopher Brooks, José Cambronero, Sumit Gulwani, Adish Singla, and Gustavo Soares. 2024. Automating human tutor-style programming feedback: Leveraging gpt-4 tutor model for hint generation and gpt-3.5 student model for hint validation. In *Proceedings of the 14th learning analytics and knowledge conference*. 12–23.

[34] Leo Porter and Daniel Zingaro. 2024. *Learn AI-assisted Python programming: with github copilot and ChatGPT*. Simon and Schuster.

[35] Stanislav Pozdniakov, Jonathan Brazil, Solmaz Abdi, Aneesha Bakharia, Shazia Sadiq, Dragan Gašević, Paul Denny, and Hassan Khosravi. 2024. Large language models meet user interfaces: The case of provisioning feedback. *Computers and Education: Artificial Intelligence* 7 (2024), 100289.

[36] James Prather, Brent N Reeves, Paul Denny, Brett A Becker, Juho Leinonen, Andrew Luxton-Reilly, Garrett Powell, James Finnie-Ansley, and Eddie Antonio Santos. 2023. "It's weird that it knows what i want": Usability and interactions with copilot for novice programmers. *ACM transactions on computer-human interaction* 31, 1 (2023), 1–31.

[37] James Prather, Brent N Reeves, Juho Leinonen, Stephen MacNeil, Arisoa S Randrianasolo, Brett A Becker, Bailey Kimmel, Jared Wright, and Ben Briggs. 2024. The widening gap: The benefits and harms of generative ai for novice programmers. In *Proceedings of the 2024 ACM Conference on International Computing Education Research-Volume 1*. 469–486.

[38] Kevin Robinson, Keyarash Jahanian, and Justin Reich. 2018. Using online practice spaces to investigate challenges in enacting principles of equitable computer science teaching. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. 882–887.

[39] Judy Sheard, Paul Denny, Arto Hellas, Juho Leinonen, Lauri Malmi, and Simon. 2024. Instructor perceptions of ai code generation tools-a multi-institutional interview study. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*. 1223–1229.

[40] Md Istiak Hossain Shihab, Christopher Hundhausen, Ahsun Tariq, Summit Haque, Yunhan Qiao, and Brian Mulanda. 2025. The Effects of GitHub Copilot on Computing Students' Programming Effectiveness, Efficiency, and Processes in Brownfield Programming Tasks. *arXiv preprint arXiv:2506.10051* (2025).

[41] Anselm Strauss and Juliet Corbin. 1994. Grounded theory methodology: An overview. (1994).

[42] Sangho Suh, Meng Chen, Bryan Min, Toby Jia-Jun Li, and Haijun Xia. 2024. Luminate: Structured Generation and Exploration of Design Space with Large Language Models for Human-AI Co-Creation. In *Proceedings of the CHI Conference on Human Factors in Computing Systems (CHI '24)*. ACM, 1–26. https://doi.org/10.1145/3613904.3642400

[43] Ningzhi Tang, Chaoran Chen, Zihan Fang, Gelei Xu, Maria Dhakal, Yiyu Shi, Collin McMillan, Yu Huang, and Toby Jia-Jun Li. 2026. Programming by Chat: A Large-Scale Behavioral Analysis of 11,579 Real-World AI-Assisted IDE Sessions. arXiv:2604.00436 [cs.SE] https://arxiv.org/abs/2604.00436

[44] Ningzhi Tang, Meng Chen, Zheng Ning, Aakash Bansal, Yu Huang, Collin McMillan, and Toby Jia-Jun Li. 2024. Developer Behaviors in Validating and Repairing LLM-Generated Code Using IDE and Eye Tracking. In *2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 40–46. https://doi.org/10.1109/VL/HCC60511.2024.00015

[45] Xiaohang Tang, Sam Wong, Marcus Huynh, Zicheng He, Yalong Yang, and Yan Chen. 2024. SPHERE: Scaling Personalized Feedback in Programming Classrooms with Structured Review of LLM Outputs. *arXiv preprint arXiv:2410.16513* (2024).

[46] Lev Tankelevitch, Viktor Kewenig, Auste Simkute, Ava Elizabeth Scott, Advait Sarkar, Abigail Sellen, and Sean Rintel. 2024. The metacognitive demands and opportunities of generative AI. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*. 1–24.

[47] Turnitin. 2024. Turnitin: Academic Integrity and AI Writing Detection. https://www.turnitin.com. Accessed: March 29, 2026.

[48] Rachel Van Campenhout, Bill Jerome, Jeffrey S Dittel, and Benny G Johnson. 2023. The doer effect at scale: Investigating correlation and causation across seven courses. In *LAK23: 13th International Learning Analytics and Knowledge Conference*. 357–365.

[49] Kurt VanLehn. 2011. The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems. *Educational psychologist* 46, 4 (2011), 197–221.

[50] Yinuo Yang, Ashley Ge Zhang, Steve Oney, and April Yi Wang. 2025. SPARK: Real-Time Monitoring of Multi-Faceted Programming Exercises. In *2025 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 81–92.

[51] Yinuo Yang, Zheng Zhang, Ningzhi Tang, Xu Wang, Alex Ambrose, Nathaniel Myers, Patrick Clauss, and Toby Jia-Jun Li. 2026. Lessons from Real-World Deployment of a Cognition-Preserving Writing Tool: Students Actively Engage with Critical Thinking and Planning Affordances. arXiv:2603.15777 [cs.HC] https://arxiv.org/abs/2603.15777

[52] Ashley Ge Zhang, Yan Chen, and Steve Oney. 2023. Vizprog: Identifying misunderstandings by visualizing students' coding progress. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–16.

[53] Ashley Ge Zhang, Yan-Ru Jhou, Yinuo Yang, Shamita Rao, Maryam Arab, Yan Chen, and Steve Oney. 2026. CodeStream: Augmenting Timelines with Code Annotation for Navigating Large Coding Histories. (2026).

[54] Ashley Ge Zhang, Xiaohang Tang, Steve Oney, and Yan Chen. 2024. CFlow: Supporting Semantic Flow Analysis of Students' Code in Programming Problems at Scale. In *Proceedings of the Eleventh ACM Conference on Learning@ Scale*. 188–199.

[55] Tim Zindulka, Sven Goller, Daniela Fernandes, Robin Welsch, and Daniel Buschek. 2026. The AI Memory Gap: Users Misremember What They Created With AI or Without. In *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems*. 1–22.

A Need Finding Participant Recruitment

Our target population is university-level programming instructors teaching 'traditional' (in-person, synchronous) courses ranging from introductory to advanced topics. We include a wide array of programming courses (e.g., Python programming, web development, and machine learning) to capture diverse experiences with AI integration in the curriculum.

Rather than distributing a broad public poll, we adopted a targeted recruitment strategy where we reached out to instructors directly. Open calls often yield self-selected responses concentrated in a few regions, and, critically, they provide no reliable way to verify whether respondents are, in fact, programming instructors. We also specifically sought *lead instructors*, since they are typically the ones who set course-level policies on AI usage and shape how such tools are integrated into programming curricula. To recruit participants, we followed these steps:

- **Identify universities**. We started by identifying multiple institutions to recruit from. We focused recruitment on (1) the 'top' 50 universities worldwide, using an established ranking (Times Higher Education)[7] and (2) the three institutions that the authors of this paper are affiliated with. Our choice to focus on 'highly-ranked' institutions was not because these universities are inherently more valuable, but because the list provided a tractable way to select institutions that span multiple continents, maintain large programming curricula across course levels, and have the resources and early exposure to AI tools needed to support relevant classroom practices. This approach ensured both diversity and feasibility in recruitment, while avoiding the impracticality of attempting to reach the full global population of programming instructors. Of the 50 universities, we proceeded with 41, primarily because some lacked publicly available instructor information.
- **Identify relevant courses and instructors**. For each selected university, we identified a list of programming-related courses for the current academic year. We focused on "high-resource" programming languages and concepts (i.e., topics where AI may be effective). This included courses in Python, C, algorithms, and data structures, spanning both introductory and advanced courses. We prioritized courses with substantial coding components, a high student-instructor ratio, and assignments that AI tools are more likely to be capable of completing. Both undergraduate- and graduate-level courses were included. We excluded seminars, capstone projects, and theory-based courses without substantial coding components. Short-term workshops or bootcamps outside the standard curriculum were also excluded. We then extracted the names and contact information of the lead instructors.
- **Email outreach**. We sent personalized emails to each instructor, including: (1) a description of the study's goals and relevance, (2) a link to the online survey, and (3) a request for referrals, encouraging instructors to forward the invitation or suggest potential participants.

To encourage participation, participants were entered into a lottery for a $10 USD Amazon gift card and received access to a dataset of anonymized survey responses. Of the 208 instructors we contacted, 27 responded to the survey. The participants were from 15 universities. The respondents represent a range of teaching experience, with the majority having between 3 and 15 years in instructional roles. Three respondents reported less than three years of experience, and others reported more than 15 years, including four participants with over 25 years. In terms of instructional role, all of the respondents identified as Head or Lead Instructors, indicating that the perspectives collected largely reflect those who hold primary responsibility for course design, instruction, and grading. Regarding course levels taught, the data show that most respondents are involved in introductory (21) and intermediate (23) courses, which constitute the foundation of many programming curricula. Some participants also teach advanced (11) and applied/specialized (9) courses. A small number of respondents reported teaching across all four levels, reflecting a broad instructional scope.

B Need Finding Study Responses

We synthesized nine high-level patterns of inappropriate AI usage reported by instructors (Table 1). For each pattern, we include a representative quote to illustrate how instructors observed and interpreted these behaviors.

C Data Collection Participants

To collect a realistic dataset of students' coding histories, we recruited 20 participants with diverse roles and backgrounds. Table 2 summarizes participant demographics.

[7] https://www.timeshighereducation.com

Table 1: Inappropriate AI Usage Patterns Reported by Instructors in the formative study. The table lists recurring behaviors observed by instructors. For each pattern, it lists how many participants mentioned it and a representative quote from participants.

Inappropriate AI Usage Patterns	# participants	Quote
Uncritical use of AI-generated solutions, e.g., pasting without understanding, full AI-written submissions	15	*"Most frequently students copy homework problems into ChatGPT or another LLM, and paste the code results. We often see submissions where the students accidentally copied extra text from the LLM like the next question prompt, or left certain wording in the form the LLM uses."*
AI use on assessments intended to measure individual understanding	7	*"The direct use of AI to trivially solve assigned problems that are explicitly intended to assess students' individual programming learning outcomes."*
Code indicating external generation, e.g., advanced syntax, imported libraries	3	*"When intro students submit solutions with complex syntax not covered in class, I assume they either used AI or a solution directly from Stack Overflow."*
Undisclosed or policy-violating AI use	2	*"In the presence of my policies that allow students to use AI so long as they fully cite all their sources and prompts, students are still not being transparent about their use."*
Shallow comprehension, e.g., failure to build fundamental skills	2	*"[students] don't actually take any time to learn the material or understand the solution"*
Students struggle during office hours or interviews to explain AI-generated code, reflecting a lack of comprehension of concepts.	2	*"Typically, student use of AI for coding is detected during office hours. Such students have code that does not work and cannot express explain how their code works in debugging with an instructor. "*
Misinterpretation of Copilot usage	2	*" When the copilot add some code automatically, they sometimes mistaken that as part of the starter code provided by the instructors."*
AI-generated artifacts showing conceptual misunderstandings, e.g., UI images, design docs	1	*"Some students/groups use AI-generated images in a document to describe a software design, and these designs are always inaccurate."*
Identical AI-generated bugs across students	1	*"I observed a few students fully relying on GenAIs to write their coding homework and having the same bug as ChatGPT's solution. "*

Table 2: Participant Demographics from the coding data collection study.

PID	Gender	Experience	Role
P1	Man	7 years	PhD Student
P2	Non-binary	6 years	Student
P3	Woman	1–2 years	Student
P4	Man	2 years	Student / Software Developer
P5	Man	7 years	PhD Student
P6	Woman	2 years	Student
P7	Man	6 years	Student
P8	Woman	3 years	Data Engineer
P9	Man	3 years	Student
P10	Man	~2 years	Technology Consultant
P11	Man	1–2 years	Student
P12	Man	~5 years	Data Analyst
P13	Woman	15 months	Student
P14	Woman	4 years	Developer
P15	Woman	<3 months	UX Researcher
P16	Man	9 years	PhD Student
P17	Woman	5 years	Software Engineer
P18	Man	~1 year	PhD Student
P19	Man	8 years	PhD Student
P20	Woman	4 years	Software Engineer

D Dataset Example

We provide an anonymized example of the collected dataset. In the dataset, each participant's coding history is stored as a JSON file containing a list of edit events, where each event is represented as a dictionary. The following is an example of an edit event where a line of code containing a return statement is added through auto-completion:


```
{
    "type": "edit",
    "data": {
      "id": "<STUDENT_ID>",
      "edits": {
        "eventName": "DocumentChangeEvent",
        "eventTime": 1756258528922,
        "sessionId": "<SESSION_ID>",
        "machineId": "<MACHINE_ID>",
        "documentUri":
        ↪  "file:///workspaces/LLMTutor/problem-1/solution.py",
        "documentId": 8,
        "operation": "add",
        "value": "return {'next_id': 1, 'students': [],
        ↪  'grades': []}",
        "rangeOffset": "702",
        "rangeLength": "0",
        "rangestart_line": "20",
        "rangestart_character": "4",
        "rangeend_line": "20",
        "rangeend_character": "4"
      },
      "code": "<CODE_CONTENT>",
      "contributor": "autocomplete",
      "context": {
        "autocompleteContent": "return {'next_id': 1,
        ↪  'students': [], 'grades': []}",
        "clipboardContent": "<CLIPBOARD_CONTENT>"
      }
    }
  }
```


E User Study

In this section, we provide additional details about the user study.

E.1 Participant Demographics

In total, 12 participants were recruited, with 4–10 years of Python experience. Table 3 summarizes participant demographics.

E.2 Results

After using each system, participants completed a self-report survey. Table 4 summarizes the results across both conditions.

After using both Editrail and the baseline, participants completed a comparison survey about their experience with the two systems. Figure 7 presents these results, in which participants explicitly compared the two conditions.

Table 3: Participant Demographics from the user study. We report their Python coding experience, courses they teach, and their professions.

ID	Python Experience (Years)	Course / Topic	Profession
1	10	(1) Python — beginner to intermediate, focusing on data science and ML; (2) C for embedded systems, beginner	Teaching Assistant
2	6	Beginner/Intermediate	Python Tutor & Instructional Aide
3	7	Intermediate	Teaching Assistant, Instructor
4	9	None	Lead Virtual Engineer
5	6	None	Graduate Student Researcher
6	9	Use Python to implement basic database system including basic database operations	Teaching Assistant
7	8	Python, beginner	Tutor
8	4	Beginner Python	Teaching Assistant
9	4+	Python, intermediate	Teaching Assistant
10	4	Django	Graduate Student Instructor
11	4	Models of Social Information Processing	Graduate Student Instructor
12	4	Introductory Computer Science	Graduate Student Instructor

Table 4: Participants' self-report results by condition. Compared with the baseline, participants reported that Editrail gave them a better understanding of students' AI usage, of how to provide targeted guidance, and of which students might need intervention.

Measures	Condition	*M*	*SD*	1 (little understanding) to 7 (good understanding)
Students' AI usage patterns	Editrail	5.92	0.90	
	Baseline	4.50	2.20	
Targeted, learning-oriented guidance on AI usage	Editrail	5.67	1.50	
	Baseline	4.67	2.31	
Students that might need intervention	Editrail	6.33	0.78	
	Baseline	5.25	2.22	

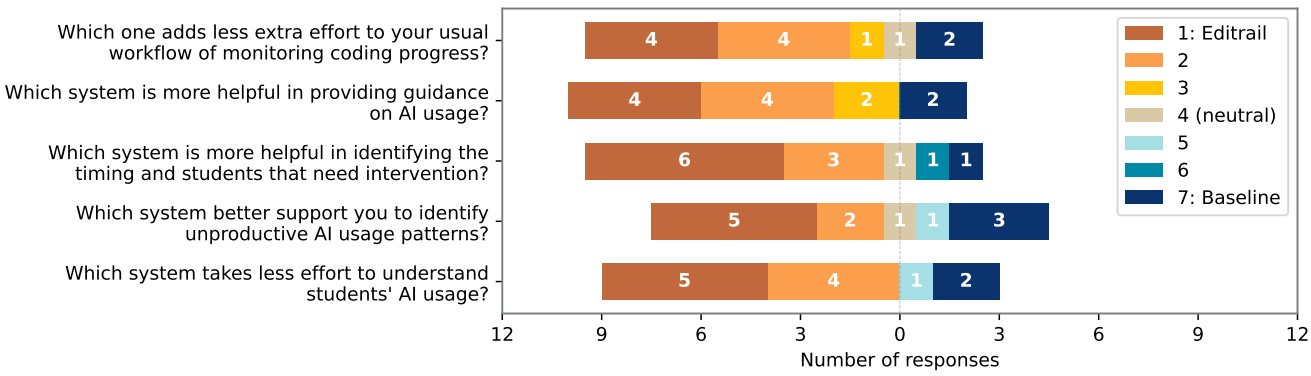

Figure 7: Results of the comparison survey. Participants answered questions (left) on a 7-point Likert scale where 1 represents "Prefer Editrail" and 7 represents "Prefer Baseline".