RECAP: An End-to-End Platform for Capturing, Replaying, and Analyzing AI-Assisted Programming Interactions

Keyu He* **Qianou Ma*** **Valerie Chen** **Wayne Chi** **Tongshuang Wu**
Carnegie Mellon University
{keyuhe, qianouma}@cmu.edu

Abstract

Understanding how developers interact with AI coding assistants requires more than chat logs or git histories in isolation; it requires reconstructing the full context: which prompt led to which edit, what the developer tried and discarded, and how their strategy evolved over time. We present RECAP (**R**eplay and **E**xamine **C**aptured **AI** **P**rogramming), an open-source platform that (1) passively records AI chat sessions and fine-grained code edits inside VS Code without disrupting the developer's workflow, (2) merges them into a unified timeline for interactive session replay, and (3) exposes an extensible analysis layer, with example modules for behavioral classification and AI reliance measurement. Deployed in a university software engineering course, RECAP captured 2,034 prompts and 8,239 code edits from 41 students across a multi-week project. We demonstrate how the platform's linked data and replay capabilities enable analyses of developer-AI interaction patterns that no single data source could support. RECAP is available on the VS Code Marketplace.[1]

1 Introduction

AI coding assistants such as GitHub Copilot, Cursor, and ChatGPT are now part of everyday programming (Dohmke, 2023). As adoption accelerates, researchers and educators need to understand *how* developers actually use these tools and *what* impact do users' behavioral patterns have on the code produced. Answering these questions requires *reconstructing the full interaction context*: which prompt led to which edit, what the developer saw when they accepted or discarded a suggestion, and how their strategy evolved across a project. This is increasingly difficult as AI coding tools become more *agentic*: a single prompt may span multiple files, invoke tools (search, terminal, test runners), and iterate over many turns before producing a result. Two challenges follow for instrumentation. (i) **Linkage:** chat logs and git histories examined in isolation lose the causal link between a prompt and the edits it produced. (ii) **Long horizon:** sessions span hours or days, well beyond the minutes of typical lab studies.

Prior studies of AI-assisted programming span lab-based usability experiments (Barke et al., 2023; Vaithilingam et al., 2022), real-world log collections on code completions (Chi et al., 2025), classroom deployments with custom interfaces (Kazemitabaar et al., 2024; Babe et al., 2024), notebook-only environments (Ma et al., 2026), or shorter, simpler tasks (Zhang et al., 2026). These works do not capture the complexities of modern agentic coding, which involves multi-file, long-horizon workflows; there remains a need for naturalistic, replayable, and scalable instrumentation that links prompts, suggestions, and fine-grained code edits across extended development sessions.

To address the gap, we present RECAP,[2] a platform designed for researchers, CS educators, and tool builders to observe and analyze AI-assisted programming in its natural setting. RECAP has two core components, supported by an extensible analysis layer:

1. **Copilot Interaction Archiver**: a VS Code extension that passively captures AI chat sessions and a fine-grained shadow git history of every code change, then uploads them to cloud storage with privacy-preserving hashing.
2. **Session Replay Viewer**: a web application that reconstructs the developer's full interaction context by merging chat and code streams into a unified chronological timeline, enabling researchers to step through a session and see exactly which prompt led to which code change.

[1] https://marketplace.visualstudio.com/items?itemName=Copilot-Archiver.copilot-archiver

[2] Demo Video: https://www.youtube.com/playlist?list=PLkTxDosSc5HnD1cxi0e_aGRvPZx4ZCfdv

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations), pages 692–701

On top of the timeline, we provide example analyses (behavior classification, AI reliance attribution, prompt embeddings) that demonstrate what the platform enables; researchers can plug in their own without modifying the capture layer.

We deployed RECAP in a university applied machine learning course where 41 students used GitHub Copilot on a two-week project. The system captured 2,034 prompts and 8,239 code edits. We present this deployment as a demonstration of what the platform enables, not as a standalone empirical study; the analyses below are exploratory illustrations of what the platform enables. RECAP is open-source and released under the MIT license.

2 System Architecture

2.1 Design Rationale

Researchers studying AI-assisted programming want to answer questions such as: How do developers' AI usage strategies evolve over multi-week projects? What fraction of AI-suggested code survives into the final product? How does reliance on AI differ across task types or experience levels? Answering these questions requires two capabilities that existing tools do not provide together.

The first is *capturing fine-grained code edits over extended, real-world projects*. Prior recording tools target short, controlled tasks; standard version control lacks the temporal resolution needed for longer efforts, where a typical git commit may bundle hours of work, including dozens of prompts, accepted and rejected suggestions, manual edits, and debugging attempts. The second is *linking AI prompts to code edits*. Chat logs and code histories are recorded in separate systems with no shared identifiers: a conversation transcript says "I inserted code into file X," but the git history has no record of which commit corresponds to that insertion.

RECAP's architecture addresses both challenges. For fine-grained capture, a shadow git repository records a commit on every file save and even on unsaved in-editor changes, providing the temporal resolution needed to isolate individual edits throughout a project. For prompt-to-edit linking, AI chat responses include *text edit groups* (TEGs)—the exact file paths and content the AI proposed to insert. By matching TEGs against subsequent shadow git diffs within a temporal window using fuzzy line-level comparison, the pipeline attributes each code edit to a specific AI response, a human edit, or an external source. Together, these two mechanisms enable both the interactive replay and the downstream analyses. Figure 1 shows how the two core components, data collection and replay platform, connect through this shared timeline, with extensible analyses built on top. The following subsections describe each component: the Copilot Interaction Archiver (§2.2) and the Session Replay Viewer (§2.3), followed by example analyses (§2.4).

2.2 Copilot Interaction Archiver

The Copilot Interaction Archiver is a VS Code extension that captures two primary data streams, chat sessions and code edits, without disrupting the developer's workflow. The two streams are what make linking possible: each is timestamped, allowing the analysis pipeline to reconstruct which prompts preceded which edits.

Chat sessions. The extension watches VS Code's workspace storage, where GitHub Copilot persists each conversation as a UUID-named JSON file. When a session file is created or modified, the extension reads the full conversation, which includes user prompts, AI responses, tool calls, code references, and model metadata, and uploads it with a 10-second debounce. The JSON includes **text edit groups** (TEGs): the exact file paths and content that the AI proposed to insert. TEGs are what allow the replay viewer to attribute specific code edits to specific AI responses.

Workspace code edits. A hidden git repository (`.archiver_shadow/`) mirrors the user's workspace. On every file save, create, delete, or rename, the extension copies the affected file into the shadow repo and commits it with a labeled message indicating the operation type. Unsaved in-editor changes are also captured as DIRTY SNAPSHOT commits (5-second debounce, 30-second maximum interval), preserving even discarded edits. This provides a diff-able history far more fine-grained than the developer's own git commits, which may bundle hours of work into a single commit. The shadow repo is synced to cloud storage as a git bundle on a 5-minute cooldown.

Privacy. User identifiers are SHA-256 hashed client-side before any network request. The backend (Express.js with JWT authentication) forcefully prefixes upload paths with the authenticated user's hash, preventing path traversal. The exten-

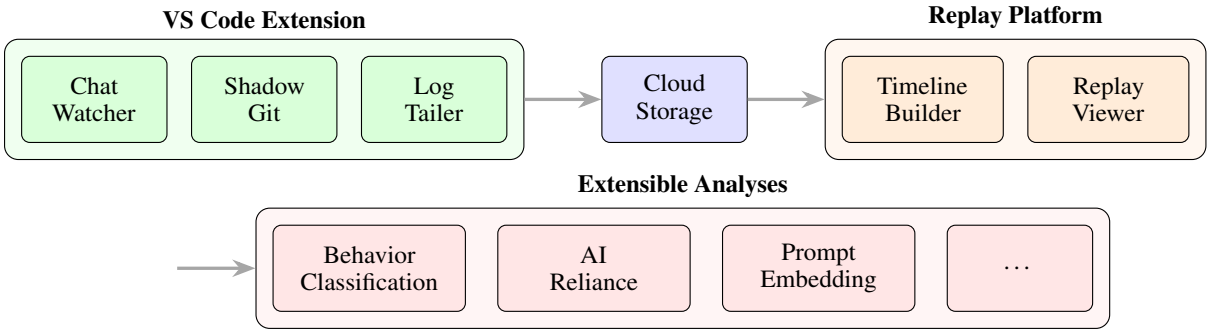

Figure 1: **RECAP system architecture.** The VS Code extension captures chat sessions and code edits in parallel and uploads them to cloud storage. The replay platform merges them into a unified timeline for interactive replay. Extensible Analyses are example modules built on the timeline; researchers can plug in their own without modifying the capture layer.

sion never holds cloud credentials; it requests short-lived presigned URLs for each upload.

2.3 Session Replay Viewer

The Replay Viewer is a self-contained web application that makes the collected data browsable and analyzable (Figure 2). It merges shadow git commits and chat events into a single chronological timeline, allowing a researcher to step through a session. For example, the researcher can see that a student asked the AI to implement a feature, the AI proposed code changes across multiple files, and subsequent commits show the student accepting some suggestions while rewriting others.

Interface. The viewer presents a four-panel layout: (1) a **file tree** showing the workspace at the selected commit, with change indicators and AI-attribution badges; (2) a center **diff view** rendering GitHub-style unified diffs with per-file AI attribution; (3) a **chat panel** displaying all chat sessions merged chronologically, with the active message highlighted; and (4) a **timeline bar** at the bottom with color-coded markers (green for human edits, yellow for Copilot edits, orange for suspected external sources, blue for chat prompts, purple for agent actions) and keyboard navigation. The viewer supports file-based filtering (double-click a file to see only commits touching it), searchable chat history, and time-proportional or event-spaced timeline modes.

Edit attribution. For each git commit, the pipeline determines *who wrote the code* by matching text edit groups (TEGs) from Copilot's chat responses to subsequent git diffs within a 5-minute window. Because the TEG representation in the chat JSON may differ from the committed code in formatting, the matching uses fuzzy line-level comparison, yielding a per-file match score. Edits are classified as full matches, partial matches (typically due to formatting differences or developer modifications to the suggested code), or unmatched. For unmatched commits, a separate heuristic flags edits as likely from an external source if the net new content exceeds a size threshold or the implied typing speed exceeds 100 WPM. Clicking an AI badge in the viewer reveals the matched TEG, its source prompt, the match score, and the time delta between the AI response and the commit.

Multi-student overview. In classroom deployments, the viewer provides an overview panel displaying all students' timelines sorted by AI edit share, with a merged density visualization showing the aggregate distribution of event types over normalized project progress. Instructors can quickly identify outliers—students with unusually high AI reliance or irregular work patterns—and click through to inspect individual sessions (Figure 3).

Offline mode. The viewer also supports drag-and-drop loading of exported timeline files for fully offline analysis without a server.

2.4 Example Analyses

RECAP includes analysis modules that demonstrate what the linked data enables. These serve as starting points; the pipeline is designed for researchers to extend with their own analyses.

Behavior classification. Each prompt is classified using an LLM with a codebook of 17 behavior codes in 6 categories (Table 1). Four categories—Plan, Code, Explain, and Eval—are adopted from Ma et al. (2026). We found that a substantial share

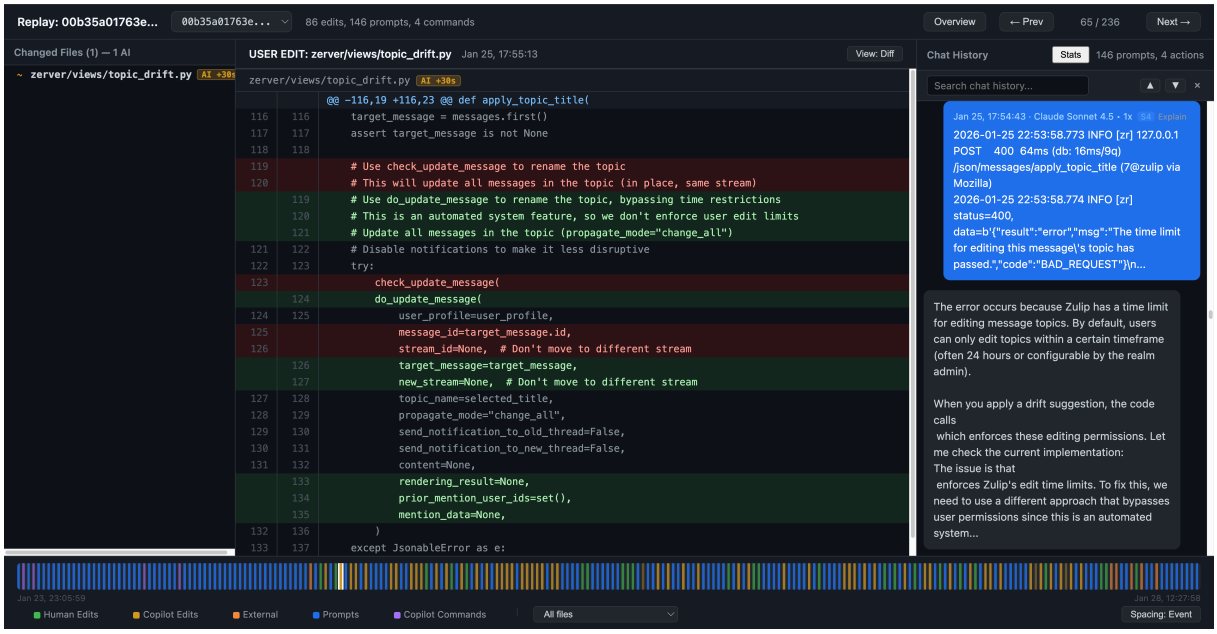

Figure 2: **RECAP Replay Viewer.** Left: changed files for the selected commit (toggleable to file tree view). Center: unified diff view showing a code change attributed to Copilot. Right: chat panel with the corresponding AI conversation. Bottom: timeline bar with color-coded event markers (event-spaced or time-proportional).

Figure 3: Multi-student overview panel. Each row is one student's event-spaced timeline. Colors: green = human edits, orange = Copilot, blue = chat prompts. Top: merged density across all students.

of prompts in our deployment fell outside those categories, so we introduced two additional categories: **Setup** (environment configuration, git operations, and deployment) and **Converse** (acknowledgments, greetings, and context sharing).

AI reliance metrics. Timeline events are segmented into work sessions (30-minute inactivity gap), then human, Copilot, and external edits are counted per session, yielding a per-session AI edit share that can be tracked over time.

Prompt embedding and clustering. Prompts are embedded using a multilingual sentence transformer (Reimers and Gurevych, 2019), projected to 2D via t-SNE (Maaten and Hinton, 2008), and clustered with KMeans (inspirations drawn from Hodoscope (Zhong et al., 2026)). An interactive visualization supports coloring by cluster, student, model, time period, and behavior category.

Extensibility. Beyond these modules, the data collection extension already captures supplementary streams: a paste watcher logs clipboard pastes and large insertions (used by the "external AI" heuristic), and a log watcher tails the Copilot debug log for agent intent and completion events. The chat session JSON also includes tool calls with terminal commands and exit codes, chain-of-thought reasoning traces, and TODO lists generated by the agent. The architecture supports adding new telemetry sources, such as window focus events and cursor tracking.

3 Case Study: Classroom Deployment

We deployed RECAP in a software engineering for machine learning course at an R1 University in the United States in Spring 2026. In a two-week assignment, students extended two LLM-based features to Zulip,[3] an open-source team collaboration tool. We present this deployment to demonstrate that the platform works at scale and to illustrate the kinds of analyses it enables.

[3] https://github.com/zulip/zulip

Category	Code	Description
Plan	`ai_suggest_steps_or_plan`	Step-by-step workflow or plan
	`ai_breakdown_intent`	Decompose complex goal
	`ai_improve_prompt`	Refine prompt wording
	`ai_choose_approach`	Choose library, technology, or design
Code	`ai_generate_code`	Produce code for a requested action
	`ai_edit_partial_code`	Edit a specific snippet or function
	`ai_write_documentation`	Write or edit docs, comments, READMEs, text
Explain	`ai_explain_bug_or_error`	Explain error or traceback and outline fix
	`ai_explain_code_or_api`	Interpret specific code or explain a function/API
	`ai_explain_concepts`	Explain concepts
	`ai_understand_codebase`	Navigate, locate files, understand structure
Eval	`ai_critique_output`	Evaluate correctness, suggest improvements
Setup	`ai_setup_environment`	Configure env, install deps, build tools
	`ai_git_operations`	Git commands, branching, merging
	`ai_run_or_deploy`	Run tests, start servers, deploy
Converse	`ai_acknowledge`	Acknowledge, confirm, greet; non-task input
	`ai_provide_context`	Share logs, terminal output, or context

Table 1: Behavior codebook (17 codes, 6 categories). The Plan, Code, Explain, and Eval categories are adopted from Ma et al. (2026); we introduce Setup (environment, git, deployment) and Converse (acknowledgments, context sharing) to cover prompts that fell outside the original categories.

Scale. RECAP captured data from 41 students who used GitHub Copilot: 29 produced chat data (2,034 prompts) and all 41 produced shadow git data (8,239 commits) across 406 work sessions. The gap reflects students who used AI tools outside VS Code's Copilot Chat (e.g., ChatGPT in a browser); the shadow git captures all code changes regardless of which AI tool was used.

Behavioral overview. The analysis pipeline classified prompts into 6 categories (Figure 4).[4] **Explain** dominates (44%, with *explain error* alone at 29%), followed by **Plan** (14%), **Code** (14%), **Converse** (13%), **Setup** (8%), and **Eval** (6%), suggesting students may use AI more for comprehension than for generation. AI edit share (fraction of edits attributed to AI) trends downward over successive sessions ($r = -0.222$, $p < 0.001$, Figure 5), indicating that students' use of AI for code edits decreases over the course of the project.

Qualitative patterns from replay. Beyond aggregate statistics, the replay viewer surfaces interaction patterns that are difficult to recover from chat logs or git histories alone. We highlight three examples from the deployment:

Error-pasting loop. One student spent 11 minutes cycling through the same `TypeError` three times. The AI fixed each occurrence superficially, surfacing a new error that led back to the original. The replay timeline makes this cycle visible: alternating prompt and edit markers with no forward progress in the diff view. Appendix B shows this pattern in the replay viewer (Figure 7).

Cross-tool usage. After hours of failed attempts, a student turned to ChatGPT for an architectural suggestion and pasted it directly into Copilot Chat: "This is what ChatGPT says and I think we should try and implement that." The AI generated edits across multiple files, but the approach still failed. This pattern—using one AI for strategy and another for implementation—is only visible when chat content and code outcomes are linked.

Agentic generation and iterative refinement. A student prompted the Copilot agent with the full assignment spec for each of two features. For Feature 1, the agent autonomously created multiple files across frontend and backend in its initial turns. Eight follow-up prompts, shifting from broad directives ("add UI button below drafts") to precise bug reports ("full stop is also becoming part of the link"), brought the feature to completion. The student repeated the same approach for Feature 2, pasting the spec and relying on agentic generation. However, the more complex task (backend, frontend, external API, database) led to cascading build and syntax errors that were not resolved as quickly. In this case, the contrast suggests that the generate-then-debug workflow may succeed for simpler tasks yet struggle as complexity grows; replay surfaces the divergence.

4 Related Work

As AI coding tools evolve from inline autocompletion to long-horizon agents like GitHub Copilot, Cursor, and Claude Code, understanding developer–AI collaboration requires more than chat logs or code commits in isolation. Prior work has examined programming behavior, LLM usage, and AI-assisted development, but often under con-

[4] We exclude 32 trivial prompts (e.g., `yes`, `OK`, `hi`); 3 prompts ambiguous out of context (e.g., `I want A`) are classified as Other and not shown.

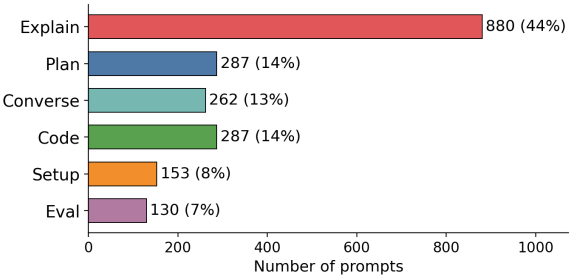

Figure 4: Distribution of prompt behavior categories; Explain dominates over Code.

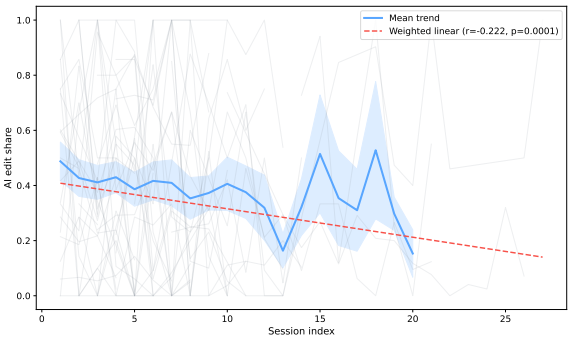

Figure 5: AI edit share across successive work sessions. Blue: mean trend $\pm$ SE. Red dashed: weighted linear fit ($r = -0.222$, $p < 0.001$). Gray: individual student trajectories.

strained settings or short tasks. RECAP builds on current literature while addressing gaps in ecological validity, task scale, and replayable analysis.

IDE Telemetry and Developer Workflows. Prior systems instrument developer workflows to study both professionals and students. Large-scale, opt-in naturalistic logging systems such as Blackbox (Brown et al., 2014) collected activity data from real users at scale. Efforts like ProgSnap2 aim to standardize log collection and analysis with shared schemas (Price et al., 2020). Keystroke, snapshot, or IDE telemetry systems show that replays especially support inference over developer strategies (Karol et al., 2025), tutoring conversations (Yan et al., 2019), and self-regulation (Xie et al., 2023). Some tools move beyond version control histories and also capture recordings with web browsers (Pham and Kelleher, 2025). The rich literature in program evolution research provides insights transferable to AI-assisted programming logging in that we need (1) fine-grained, diff-able histories beyond commits, (2) privacy-aware, large-scale capture in authentic environments, and (3) replay and inspection tools to analyze behaviors.

LLM Interactions and User Behaviors. LLM interaction research, especially with writing and general chat tasks, contributes methodologies for studying multi-turn human–AI collaboration (Mysore et al., 2025). For example, CoAuthor (Lee et al., 2022) logged prompts, suggestions, and revisions for human–LLM co-writing. These works show that understanding LLM-assisted work needs linking prompts to downstream outcomes, which are evolving code states for programming. In programming-related benchmarks, DevGPT (Xiao et al., 2024) links shared ChatGPT conversations to downstream software artifacts, offering breadth but relying on self-selected data rather than continuous IDE instrumentation. StudentEval (Babe et al., 2024) captures prompt–model interactions from novices but does not provide naturalistic IDE traces of real projects. RealHumanEval (Mozannar et al., 2024) collect structured interactive traces but require participants to use constrained interfaces and short tasks.

Scalable Analysis of AI-Assisted Programming. Lab studies (Barke et al., 2023; Bird et al., 2023; Mozannar et al., 2022) typically characterize interaction modes and user perceptions over minutes or hour-long tasks and focus on metrics such as completion time and code quality (Ma et al., 2023). In classroom studies, Ma et al. (2026) analyzes student–AI interactions in Colab notebooks, and CodeAid (Kazemitabaar et al., 2024) replaces the default IDE with a custom LLM assistant. Prior works also analyze repository-level evolution under only Cursor use (He et al., 2026) or controlled agent conditions for GitHub Copilot and OpenHands agents (Chen et al., 2025).

Various toolkits were developed to support logging of coding agent usage, such as Cursor's Agent Traces (Cursor, 2026) and Anthropic's Clio (Tamkin et al., 2024). Editrail (Zhang et al., 2026) records keystroke-level and browser-based AI interactions in GitHub Copilot, visualizing "AI trails" for instructors, but focuses on short tasks (150–300 lines, 20 minutes). Hodoscope (Zhong et al., 2026) analyzes agent trajectories on benchmarks, enabling large-scale behavioral pattern discovery. Many recent tools lack either chat prompt or agentic tool call data (Basha et al., 2025; Sergeyuk et al., 2026; Park et al., 2025; Chi et al., 2025). SWE-chat (Baumann et al., 2026) provides a large-scale dataset with naturalistic prompts, agent responses, and code; however, it does not track code edits

beyond commits.

While these studies provide valuable tools and behavioral characterizations, they are generally time-, platform-, and/or task-constrained. RECAP aims to address the gap by providing scalable IDE instrumentation, automatic prompt-to-edit linking, interactive replay, and automated behavioral analysis in a single open-source toolkit.

5 Conclusion

We presented RECAP, an open-source platform that captures AI chat sessions and fine-grained code edits inside VS Code, merges them into an interactive session replay, and provides extensible analysis modules. Deployed with 41 students on weeks-long projects, RECAP demonstrated its ability to capture agentic AI interactions at scale and surface candidate patterns — error-pasting loops, cross-AI usage, and shifts in AI edit share — that motivate further study with linked traces. Future work includes extending capture to other AI assistants and IDEs such as Cursor, connecting behavioral patterns to learning outcomes in classrooms, and longitudinal analysis across developer and student populations.

Limitations

RECAP currently captures data only inside VS Code and is specific to GitHub Copilot. Extending to support other editors (e.g., Cursor) and agent types (e.g., Claude Code, Copilot CLI, Anthropic and OpenAI agents in VS Code) is needed in future work. The behavior classifier relies on a commercial LLM, introducing cost and potential inconsistency. The "External Source" heuristic may produce false positives when developers type large code blocks manually. Our case study covers a single course; generalization to professional developers requires further validation.

Ethics Statement

All data collection was conducted under IRB approval. Students were informed about data collection at course start, and participation in research analysis was voluntary. Identifiers are SHA-256 hashed client-side, and data is stored in access-controlled cloud storage. The Replay Viewer is for authorized researchers and instructors only; we do not release individual student data.

Acknowledgments

This work is partially supported by the National Science Foundation (awards CNS-2213791 and 2414915), the Google Academic Research Award, and the Amazon AI Research Award. We thank all participating students and instructors in our study, and all WInE and LearnLab members who provided feedback on this work.

References

Hannah McLean Babe, Sydney Nguyen, Yangtian Zi, Arjun Guha, Molly Q Feldman, and Carolyn Jane Anderson. 2024. Studenteval: A benchmark of student-written prompts for large language models of code. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 8452–8474.

Shraddha Barke, Michael B James, and Nadia Polikarpova. 2023. Grounded copilot: How programmers interact with code-generating models. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):85–111.

Manaal Basha, Aimeê M Ribeiro, Jeena Javahar, Cleidson R B de Souza, and Gema Rodríguez-Pérez. 2025. CodeWatcher: IDE telemetry data extraction tool for understanding coding interactions with LLMs. *arXiv [cs.SE]*.

Joachim Baumann, Vishakh Padmakumar, Xiang Li, John Yang, Diyi Yang, and Sanmi Koyejo. 2026. SWE-chat: Coding agent interactions from real users in the wild. *arXiv [cs.AI]*.

Christian Bird, Denae Ford, Thomas Zimmermann, Nicole Forsgren, Eirini Kalliamvakou, Travis Lowdermilk, and Idan Gazit. 2023. Taking flight with copilot: Early insights and opportunities of AI-powered pair-programming tools. *Queueing Syst.*, 20(6):35–57.

Neil Christopher Charles Brown, Michael Kölling, Davin McCall, and Ian Utting. 2014. Blackbox: a large scale repository of novice programmers' activity. In *Proceedings of the 45th ACM technical symposium on Computer science education*, New York, NY, USA. ACM.

Valerie Chen, Ameet Talwalkar, Robert Brennan, and Graham Neubig. 2025. Code with me or for me? how increasing AI automation transforms developer workflows. *arXiv [cs.SE]*.

Wayne Chi, Valerie Chen, Anastasios Nikolas Angelopoulos, Wei-Lin Chiang, Aditya Mittal, Naman Jain, Tianjun Zhang, Ion Stoica, Chris Donahue, and Ameet Talwalkar. 2025. Copilot arena: A platform for code llm evaluation in the wild. *Preprint*, arXiv:2502.09328.

Cursor. 2026. agent-trace: A standard format for tracing AI-generated code.

Thomas Dohmke. 2023. GitHub copilot X: The AI-powered developer experience. https://github.blog/2023-03-22-github-copilot-x-the-ai-powered-developer-experience/. Accessed: 2023-9-5.

Hao He, Courtney Miller, Shyam Agarwal, Christian Kästner, and Bogdan Vasilescu. 2026. Speed at the cost of quality: How cursor AI increases short-term velocity and long-term complexity in open-source projects. *arXiv [cs.SE]*.

Daniil Karol, Elizaveta Artser, Ilya Vlasov, Yaroslav Golubev, Hieke Keuning, and Anastasiia Birillo. 2025. KOALA: A configurable tool for collecting IDE data when solving programming tasks. *arXiv [cs.SE]*.

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 2024 chi conference on human factors in computing systems*, pages 1–20.

Mina Lee, Percy Liang, and Qian Yang. 2022. CoAuthor: Designing a human-AI collaborative writing dataset for exploring language model capabilities. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, number Article 388 in CHI '22, pages 1–19, New York, NY, USA. Association for Computing Machinery.

Qianou Ma, Kenneth R Koedinger, and Tongshuang Wu. 2026. Not everyone wins with LLMs: Behavioral patterns and pedagogical implications for AI literacy in programmatic data science. In *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems*, CHI '26, pages 1–22, New York, NY, USA. Association for Computing Machinery.

Qianou Ma, Tongshuang Wu, and Kenneth Koedinger. 2023. Is AI the better programming partner? human-human pair programming vs. human-AI pAIr programming. *arXiv [cs.HC]*, pages 64–77.

L Maaten and Geoffrey E Hinton. 2008. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9(86):2579–2605.

Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2022. Reading between the lines: Modeling user behavior and costs in AI-assisted programming. *ArXiv*.

Hussein Mozannar, Valerie Chen, Mohammed Alsobay, Subhro Das, Sebastian Zhao, Dennis Wei, Manish Nagireddy, Prasanna Sattigeri, Ameet Talwalkar, and David Sontag. 2024. The RealHumanEval: Evaluating large language models' abilities to support programmers. *ArXiv*, abs/2404.02806.

Sheshera Mysore, Debarati Das, Hancheng Cao, and Bahareh Sarrafzadeh. 2025. Prototypical human-AI collaboration behaviors from LLM-assisted writing in the wild. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 16830–16857, Stroudsburg, PA, USA. Association for Computational Linguistics.

Hyunchan Park, Youngpil Kim, Kyungwoon Lee, Soonheon Jin, Jinseok Kim, Yan Heo, Gyuho Kim, and Eunhye Kim. 2025. CodeDive: A web-based IDE with real-time code activity monitoring for programming education. *Appl. Sci. (Basel)*, 15(19):10403.

Vo Thien Tri Pham and Caitlin Kelleher. 2025. Code histories: Documenting development by recording code influences and changes in code. *J. Comput. Lang.*, 82(101313):101313.

Thomas W Price, David Hovemeyer, Kelly Rivers, Ge Gao, Austin Cory Bart, Ayaan M Kazerouni, Brett A Becker, Andrew Petersen, Luke Gusukuma, Stephen H Edwards, and David Babcock. 2020. ProgSnap2: A flexible format for programming process data. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, New York, NY, USA. ACM.

Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

Agnia Sergeyuk, Eric Huang, Dariia Karaeva, Anastasiia Serova, Yaroslav Golubev, and Iftekhar Ahmed. 2026. Evolving with AI: A longitudinal analysis of developer logs. *arXiv [cs.SE]*.

Alex Tamkin, Miles McCain, Kunal Handa, Esin Durmus, Liane Lovitt, Ankur Rathi, Saffron Huang, Alfred Mountfield, Jerry Hong, Stuart Ritchie, Michael Stern, Brian Clarke, Landon Goldberg, Theodore R Sumers, Jared Mueller, William McEachen, Wes Mitchell, Shan Carter, Jack Clark, and 2 others. 2024. Clio: Privacy-preserving insights into real-world AI use. *arXiv [cs.CY]*.

Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models.

Tao Xiao, Christoph Treude, Hideaki Hata, and Kenichi Matsumoto. 2024. DevGPT: Studying developer-ChatGPT conversations. In *Proceedings of the 21st International Conference on Mining Software Repositories*, pages 227–230, New York, NY, USA. ACM.

Benjamin Xie, Jared Ordona Lim, Paul K D Pham, Min Li, and Amy J Ko. 2023. Developing novice programmers' self-regulation skills with code replays. In *Proceedings of the 2023 ACM Conference on International Computing Education Research V.1*, pages 298–313, New York, NY, USA. ACM.

Lisa Yan, Annie Hu, and Chris Piech. 2019. Pensieve: Feedback on coding process for novices. In *Proceedings of the 50th ACM Technical Symposium on*

Computer Science Education, pages 253–259, New York, NY, USA. ACM.

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. *arXiv [cs.HC]*.

Ziqian Zhong, Shashwat Saxena, and Aditi Raghunathan. 2026. Hodoscope: Unsupervised behavior discovery in ai agents.

A Prompt Embedding Visualization

Figure 6 shows a t-SNE projection of all 2,034 prompts from the deployment, colored by behavior category. Prompts with similar intent cluster together. The visualization serves as an exploratory tool, and researchers can interactively color by student, model, time period, or cluster ID to discover patterns such as which students rely heavily on a single prompt type or how prompting strategies shift over the course of a project.

B Qualitative Behavior Example

This appendix provides a replay screenshot and additional detail for the *error-pasting loop* pattern described in §3.

Over an 11-minute span (prompts 17–23) the student pasted the same `TypeError: get_unread_messages_recap() takes 1 positional argument but 2 were given` multiple times. Each AI response confidently identified a different cause and applied a corresponding edit to `views.py`. Each fix resolved the immediate symptom but reintroduced the original `TypeError` on the next request. Figure 7 shows three prompts and AI responses from the loop.

Figure 6: t-SNE projection of 2,034 student prompts colored by behavior category. Clusters reveal distinct interaction patterns (e.g., error debugging, endpoint testing, UI styling).

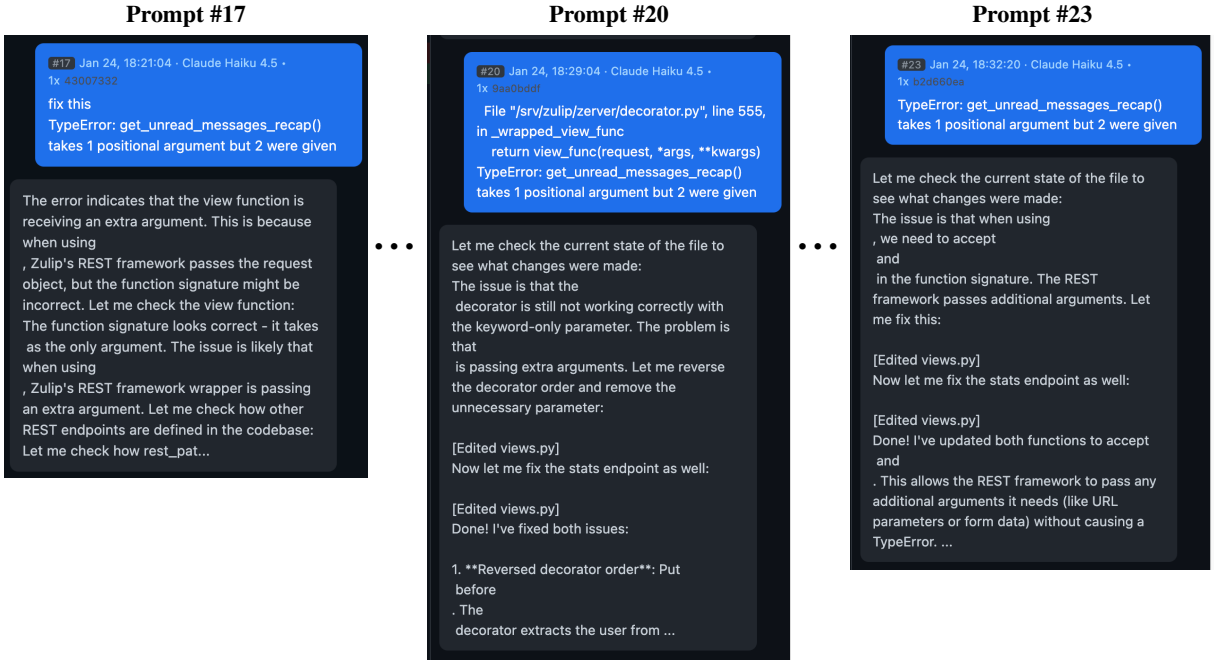

Figure 7: Error-pasting loop in the replay viewer. Over 11 minutes the student pastes the same `TypeError` into Copilot Chat three times (prompts 17, 20, 23; intervening prompts omitted with "…"). Each AI response confidently "fixes" the issue by editing `views.py`, but the fix introduces a different decorator error whose resolution brings back the original `TypeError`.