FlightRecorder: Tracing Fine-Grained Development with AI Assistants

Tobias Dick
Saarland University
Saarbrücken, Germany
tdick@cs.uni-saarland.de

Benedict Bliem
Saarland University
Saarbrücken, Germany
bebl00001@stud.uni-saarland.de

Kallistos Weis
Saarland University
Saarbrücken, Germany
kallistos@cs.uni-saarland.de

Alisa Welter
Saarland University
Saarbrücken, Germany
welter@cs.uni-saarland.de

Sven Apel
Saarland University
Saarbrücken, Germany
apel@cs.uni-saarland.de

Abstract

Software development is rapidly transforming with the widespread adoption of AI coding assistants such as GitHub Copilot. While research has begun to investigate how these tools change the development process, fundamental questions about how code evolves as developers collaborate with AI agents, particularly at fine-grained levels of interaction, remain open. A key limitation of existing research is reliance on telemetry data that cannot distinguish agent-generated changes from inline completions and human edits, nor track when and where modifications occur with sufficient precision. We present FlightRecorder, a lightweight VSCode extension that collects fine-grained code evolution data without disrupting the workflows of developers. Our extension integrates three data sources: snapshots of agent-mode edits, inline completions, and chat history, all timestamped for precise attribution. Researchers can export these data in structured formats alongside anonymization and basic visualization functionality. By making such fine-grained evolution data accessible, FlightRecorder enables researchers to investigate fundamental questions about AI-assisted development. We provide a replication package, including the code for FlightRecorder and the data used throughout our paper.[1]

ACM Reference Format:
Tobias Dick, Benedict Bliem, Kallistos Weis, Alisa Welter, and Sven Apel. 2026. FlightRecorder: Tracing Fine-Grained Development with AI Assistants. In . ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Software development is undergoing a significant shift with the widespread adoption of AI coding assistants such as GitHub Copilot[2]. These tools augment developers through multiple interaction modes, ranging from inline code completions within the editor to agent-driven development, where autonomous agents handle iterative cycles of implementation, testing, and refinement. This range of capability has made AI-assisted development prevalent in industry and has motivated considerable research on LLMs, agents, and coding assistants [3, 5, 11]. Although prior work has examined how AI assistance shapes development processes, human-agent collaboration, code evolution, software quality, and long-term engineering practice remain poorly understood [7].

Most existing studies on how developers use AI coding assistants rely on behavioral telemetry—tracking IDE focus, clipboard activity, and pasting patterns—rather than analyzing code evolution directly [2, 4, 8]. A key obstacle is the absence of fine-grained observational data. The crux is that existing tools cannot reliably distinguish agent-generated edits from inline completions and human edits, nor can they track when and where these occur without disrupting development workflows.

To address this gap, we present FlightRecorder, a VSCode[3] extension that collects fine-grained code attribution data when developers use GitHub Copilot, without disrupting their workflows. Beyond data collection, FlightRecorder includes anonymization and visualization functionality (see Figure 2) to support ethical research practices and enable initial exploration of the collected data. Our extension captures three complementary data sources: (1) snapshots of code edits made by GitHub Copilot's agent mode, (2) accepted inline code completions, and (3) the agent's chat history.

Each data source is timestamped allowing researchers to correlate modifications across all three sources and reconstruct the

```
207     def handle_create(self, name: str, dish_type: Optional[str] = None,
        dietary_restriction: Optional[Any] = None) -> Dict[str, Any]:
208         """Create a new empty meal."""
209         if name in self.meals:
210             return {"status": "error", "message": "Meal already exists."}
211         dietary_restrictions = self._normalize_dietary_restrictions
            (dietary_restriction)
212         self.meals[name] = {
213             "ingredients": [],
214             "dishType": dish_type or self.config.get("dishType"),
215             "dietaryRestriction": dietary_restrictions
216         }
217         return {"status": "ok"}
```

Figure 1: Edit history visualization with code colored by the edit that introduced it (oldest → newest).

[1]We further provide a screencast demo of our tool at https://youtu.be/bZR6aPROVMU.
[2]https://github.com/features/copilot
[3]https://code.visualstudio.com

Conference'17, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

complete picture of code evolution. Since agents can perform many code edits in rapid succession, FlightRecorder does not capture every individual edit. Instead, it accumulates edits until the agent has not made code edits within a user-configurable debounce window. At these pauses, FlightRecorder creates snapshots of the accumulated agent edits. These snapshots, combined with inline completion logs and chat history, are then exported in a structured format that researchers can analyze.

By collecting these data with precise timestamps, researchers can reconstruct and attribute code evolution, identifying which lines were added or modified, when they were edited, and which tool or interaction triggered each edit, and thus can finally investigate the concrete questions that matter to understand AI-assisted development. How do agents and humans work together to structure source code? Do agents make edits linearly, moving sequentially through files, or do they follow conceptual logic and control flow? How much effort do developers invest in refactoring or building upon agent suggestions? Examining these patterns reveals how code evolves when humans and agents work together.

We evaluate FlightRecorder in a controlled study with 18 participants using GitHub Copilot working on an open-ended development task either alone (human–AI) or in pairs (human–human–AI). Our study demonstrates that FlightRecorder reliably captures meaningful data about code evolution and developer-agent interaction. We further show how to analyze the collected data through FlightRecorder's built-in edit history visualization and discuss potential applications for research and practice.

2 Background and Related Work

AI coding assistants, such as GitHub Copilot, have become widely adopted in software engineering practice. Initially, these tools aimed to increase developer productivity by reducing manual coding effort and accelerating routine tasks, for example through real-time code suggestions within an IDE. Recent advances have enabled AI coding assistants to function as autonomous agents capable of executing multi-step software engineering tasks with little to no human intervention. Consequently, a growing body of work has begun to investigate how developers interact with these agents, as well as how such systems influence software development practices and developer behavior [6, 10, 11].

To track how developers interact with AI coding assistants, prior work often relies on screen recordings and manual annotation of coding sessions [1, 10]. Going further, Guglielmi et al. [4] track developer behavior using an IDE plugin, called Tako, screen recordings, and survey data to compare tasks performed with and without GitHub Copilot, investigating how code suggestions affect developer productivity. Tako captures inline code completions—the primary interaction mode with GitHub Copilot at the time of the study—and provides a dedicated analytics view summarizing usage statistics. However, as GitHub Copilot has evolved, its inline completion tracking appears to no longer function, and edits made by the agent are not supported. Tang et al. [8] present CodeGRITS, a JetBrains plugin that combines IDE interaction tracking with eye gaze data to study developers' cognitive processes, and apply it to examine how developers validate and repair LLM-generated code [9], but do not track AI-generated edits directly. Closer to our work, Basha et al. present CodeWatcher [2], a client-server system with a VSCode plugin that captures fine-grained developer interaction events such as insertions, deletions, copy-paste actions and window focus. Javahar et al. [6] use CodeWatcher to study developers using an AI coding assistant, specifically Amazon's CodeWhisperer, to uncover behavioral patterns during development. Crucially, CodeWatcher only has access to IDE telemetry and not to the coding assistant itself, and must therefore infer AI-generated edits through heuristics such as the size of an insertion.

3 FlightRecorder

FlightRecorder is implemented as a lightweight VSCode extension available on the VSCode Marketplace[4] and is open source[5]. It runs locally on the developer's machine, integrating seamlessly into their existing workflow with Git as its only dependency.

Recording AI Interactions. FlightRecorder records AI-related events within explicitly delimited recording sessions that the developer can start and stop on demand. It works by continuously monitoring GitHub Copilot's log output for a selected workspace, detecting events that correspond to either accepted *inline completions* or applied *agent edits*. For each detected event, FlightRecorder records a timestamp and the set of affected files, then immediately captures a snapshot of the current file state. Additional event-specific metadata, including the event type, timestamp, and a list of affected files are stored alongside each snapshot in a structured JSON format. Snapshots are persisted as commits in a Git repository that FlightRecorder maintains automatically in the background, storing the recorded history in a well-established, portable format and allowing later analyses to build on existing Git tooling. To avoid inflating the snapshot history during rapid successive edits, events that fall within a configurable debounce window are collapsed into a single commit, with all individual event-metadata kept intact. Beyond event-based snapshots, FlightRecorder exports two complementary artifacts either on request or when a recording session ends: (1) a *chat session export*, which captures the conversation files that VSCode stores locally for the chat interface, and (2) the raw *log file* that GitHub Copilot writes to disk during the session. Both artifacts are stored verbatim in the repository, with the only transformations applied being the path and identity anonymizations described below.

Anonymization. For FlightRecorder to be usable in research settings, the collected repositories must not leak personal information such as the developer's identity or local file system structure. To this end, FlightRecorder provides a command that produces an anonymized clone of the collected repository by rewriting its full Git history using git-filter-repo[6]. Author and committer identities are replaced with stable, deterministic placeholders, ensuring that co-author relationships are preserved across the full history. Absolute paths in file contents and commit messages are replaced as well. Users can configure whether to replace only paths that identify the developer's machine or all absolute paths.

[4] https://marketplace.visualstudio.com/items?itemName=SE-SIC.flight-recorder
[5] https://github.com/se-sic/flight-recorder
[6] https://github.com/newren/git-filter-repo

Edit History Visualization. To support qualitative analysis of AI-assisted development data, FlightRecorder provides an interactive edit history visualization that attributes each character of code to the commit that introduced it. Characters are color-coded by their originating commit, and hovering over any region surfaces the contextual metadata recorded at that point, including the event type, timestamp, and affected files. The color scheme can be configured in three modes: two timeline views that position commits relative to either the full repository history or the current file, and an event-type view that distinguishes between inline completions and agent edits. Together, these views allow researchers to trace how AI-generated code evolves across a session, identifying which regions survive intact, which are iteratively revised, and which are discarded shortly after being introduced.

Scope & Limitations. FlightRecorder is designed to capture GitHub Copilot activity during active development sessions within a single VSCode workspace, encompassing both GitHub Copilot-induced code changes and chat sessions. All recording is strictly scoped to that workspace; no data from other projects or workspaces is collected, and nothing is written outside the local repository. Since changes are recorded via Git commits, only accepted inline completions are captured cleanly, as rejected suggestions leave no trace. Agent edits present a subtler challenge: because acceptance is a VSCode feature rather than a GitHub Copilot one, FlightRecorder cannot intercept it, and instead captures the full edit as soon as it is applied, before the developer has had a chance to review it. A subsequent rejection can therefore only be inferred from later snapshots rather than being attributed to GitHub Copilot directly. While FlightRecorder does not establish a direct link between agent edits and user prompts, each agent-edit snapshot includes the corresponding request identifier, enabling downstream analyses to correlate it with the prompt that triggered the edit.

4 Showcase

To evaluate FlightRecorder, we conduct a controlled empirical study to demonstrate the data it captures and analyses it enables. We report the collected data and key insights enabled by our tool.

Study Setup. We recruited 18 computer science students (undergraduate and graduate) from a seminar to participate in our study. Participants were given a specification for a small coding project with two parts: a guided first part with detailed specifications, and an open-ended second part offering extension ideas. Participants either worked individually with GitHub Copilot (human–AI, $n = 8$) or in pairs (human–human–AI, $n = 10$). All participants implemented the project in Python over a 60-minute session, using FlightRecorder to record their activity on their own machines. Prior to the study, all participants provided informed consent.

Refining our study protocol, we conducted a pilot study with several students, which allowed us to improve the project specification and address robustness issues in FlightRecorder, particularly its handling of Git misconfigurations and compatibility across log formats from different GitHub Copilot versions. Following these refinements, we collected valid data from 16 of 18 participants; one used the command line GitHub Copilot rather than VSCode's integrated version, and another encountered Git configuration issues mid-session; both were excluded from our analysis.

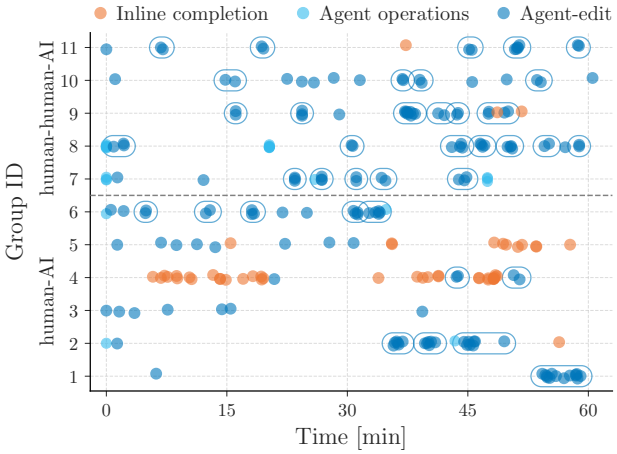

Figure 2: Edit timeline across 6 human–AI and 5 human–human–AI groups. Circled clusters denote multiple agent-edits within the same chat response.

Workflow Insights. FlightRecorder enables us to reconstruct a detailed timeline of AI-related activity for each group, as displayed in Figure 2, with the 60-minute observation window beginning at their first request to GitHub Copilot. Recorded events include *inline completions* and *agent operations*, where agent operations encompass file creation, string replacement, and the primary interaction mode *agent edits*. The timeline reveals distinct usage patterns: while agent-edits predominate across most sessions, groups differ notably in their reliance on inline completions and the duration of idle periods between agent interactions.

Interestingly, in the human–human–AI setting, manual editing (i.e., direct code modification or inline completions without invoking the agent) is nearly absent. A manual analysis confirms that human–human–AI pairs worked almost exclusively through GitHub Copilot's agent, directing it and interpreting its output rather than writing code themselves. The human–AI condition exhibits greater heterogeneity. Group 6 mirrors the human–human–AI pattern, relying almost exclusively on the agent, while the remaining groups fall into two patterns: groups 4 and 5 alternated between agent use and inline completions, whereas groups 1 and 3 manually edited code without relying on completions. Group 2 represents a technical outlier: although they used the agent between minutes 5 and 35, it operated via the command-line tool rather than VSCode's edit-tool, creating a gap in recorded data and preventing the capture of individual change acceptances or rejections[7].

File-level Inspection. FlightRecorder 's edit history visualization allows us to inspect how code evolves at the file level for individual participants. Figure 1 shows a function written by group 8, one of the human–human–AI pairs that worked exclusively through the agent. In this function, an initial commit (blue) establishes the core implementation, before later edits (yellow, red) extend the function with additional optional parameters. This progression suggests that the group first created this function when directing the agent to

[7] Group 2 used the weakest available LLM at the time, which likely explains this atypical behavior, especially the failure to follow GitHub Copilot's tool-use instructions.

satisfy the guided specification. They later returned to extend it with two separate features from the open-ended part of the project. While this interpretation requires manual inspection to confirm, it illustrates how the edit history visualization can surface structure that is not visible in the session-level timeline: the correspondence between edit events and distinct phases of the task.

5 Discussion

The data collected by FlightRecorder open several avenues for both research and practice. In what follows, we discuss how the collected snapshots can be leveraged to understand agent-assisted development workflows, support developers during interactive coding sessions, and improve code review practices when working with agents. We conclude by outlining promising directions for future extensions of FlightRecorder.

Understanding AI-Assisted Development Workflows. Code evolution is typically tracked through version control systems or direct observation of developers. AI-assisted software development introduces a qualitatively different challenge: agents can generate substantial amounts of code and perform numerous modifications in a short span of time, making it difficult to understand the evolution of the codebase through conventional means. FlightRecorder addresses this by capturing atomic snapshots of agent edits during a development session, providing fine-grained visibility into how code evolves when developers interact with AI agents. This enables researchers to study AI-assisted development practices in controlled settings and identify recurring interaction patterns, such as which code regions undergo repeated modifications versus which are created once and left intact. Such insights may help understanding how developers leverage agents effectively and which interaction strategies prove most productive.

Supporting Interactive Development with Agents. Beyond research applications, FlightRecorder offers potential to enhance the developer experience during agent-assisted coding. When developers request substantial changes—such as adding features or fixing bugs—agents often generate large changesets that may necessitate iterative refinements or backtracking if the changes prove unsuitable. The snapshot mechanism that FlightRecorder provides enables developers to easily revert to previous states within a session, offering a form of fine-grained undo that accounts for the scale of agent-generated changes. However, translating this capability to production environments introduces practical challenges. The atomic-level granularity of FlightRecorder's snapshots, while valuable for detailed analysis, becomes unwieldy in large collaborative codebases, where hundreds of agent edits would generate correspondingly many commits and clutter the project's version control history. To make this approach viable, developers would need to selectively collapse related changes into coherent commits before merging, or work within isolated branches during AI-assisted sessions. Nevertheless, the underlying idea of leveraging fine-grained interaction history to support iterative development merits further exploration for real-world deployment.

Improving Code Review for Large Changesets. A critical challenge in AI-assisted development is conducting effective code review when agents introduce hundreds or thousands of lines in a single session. The edit history captured by FlightRecorder could inform code review by surfacing heavily-modified code regions as high-priority for more careful review. Distinguishing between code that evolved through iteration (indicating developer uncertainty or agent overcorrection) and code that remained stable from creation gives reviewers additional information about which sections may need increased attention. Such contextual information, derived from the interaction history rather than static analysis, could complement existing review practices.

6 Conclusion

We introduce FlightRecorder, a lightweight VSCode extension that captures fine-grained code evolution data during AI-assisted development. We show that our tool reliably surfaces behavioral patterns that telemetry-based approaches cannot distinguish, validated through a controlled study with 18 participants.

FlightRecorder is open source, and we plan to extend it in several directions. Continuously staging human-made edits would cleanly separate human and AI contributions in the repository history. Capturing a wider range of agent tool uses—such as terminal commands or Web searches—would yield more complete records of how agents navigate complex development tasks. We also plan to extend support to additional AI coding assistants such as Claude Code. We welcome community contributions.

7 Data Availability

For reproducibility, we provide a replication package, including the code for FlightRecorder and the data used throughout our paper at https://doi.org/10.5281/zenodo.20134366.

Acknowledgements

This work has been supported by the European Union as part of the ERC Advanced Grant Brains On Code (101052182) and by the DFG as part of the TRR 248 CPEC (389792660).

References

[1] 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 (2023), 85–111. doi:10.1145/3586030

[2] Manaal Basha, Aimeê M Ribeiro, Jeena Javahar, Cleidson RB De Souza, and Gema Rodríguez-Pérez. 2025. Codewatcher: IDE Telemetry Data Extraction Tool for Understanding Coding Interactions with LLMs. In *2025 IEEE International Conference on Software Maintenance and Evolution*. 935–939. doi:10.1109/icsme64153.2025.00110

[3] Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. 2025. A Survey on Code Generation with LLM-based Agents. *arXiv preprint arXiv:2508.00083* (2025). doi:10.48550/arXiv.2508.00083

[4] Emanuela Guglielmi, Venera Arnoudova, Gabriele Bavota, Rocco Oliveto, and Simone Scalabrino. 2025. How do Copilot Suggestions Impact Developers' Frustration and Productivity? *arXiv preprint arXiv:2504.06808* (2025). doi:10.48550/arXiv.2504.06808

[5] Burak Gülmez. 2026. Code Generation with Large Language Models: A Survey from Neural Program Synthesis to Autonomous Software Development. *Applied Intelligence* 56, 6 (2026), 200. doi:10.1007/s10489-026-07230-0

[6] Jeena Javahar, Tanya Budhrani, Manaal Basha, RB Cleidson de Souza, Ivan Beschastnikh, and Gema Rodríguez-Pérez. 2025. Cracking CodeWhisperer: Analyzing Developers' Interactions and Patterns During Programming Tasks. In *2025 IEEE Symposium on Visual Languages and Human-Centric Computing*. 391–399. doi:10.1109/vl-hcc65237.2025.00051

[7] Agnia Sergeyuk, Ilya Zakharov, Ekaterina Koshchenko, and Maliheh Izadi. 2026. Human-AI Experience in Integrated Development Environments: A Systematic

Literature Review. *Empirical Software Engineering* 31, 3 (2026), 55. doi:10.1007/s10664-025-10793-0

[8] Ningzhi Tang, Junwen An, Meng Chen, Aakash Bansal, Yu Huang, Collin McMillan, and Toby Jia-Jun Li. 2024. CodeGRITS: A Research Toolkit for Developer Behavior and Eye Tracking in IDE. In *Proceedings of the 2024 IEEE/ACM 46th international conference on software engineering: Companion proceedings*. 119–123. doi:10.1145/3639478.3640037

[9] 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*. 40–46. doi:10.1109/vl/hcc60511.2024.00015

[10] Alisa Welter, Niklas Schneider, Tobias Dick, Kallistos Weis, Christof Tinnes, Marvin Wyrich, and Sven Apel. 2025. An Empirical Study of Knowledge Transfer in AI Pair Programming. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering*. 166–177. doi:10.1109/ase63991.2025.00022

[11] Xin Zhang, Peng Zhang, Yuan Shen, Min Liu, Qiong Wang, Dragan Gašević, and Yizhou Fan. 2024. A Systematic Lterature Review of Empirical Research on Applying Generative Artificial Intelligence in Education. *Frontiers of Digital Education* 1, 3 (2024), 223–245. doi:10.1007/s44366-024-0028-5