A Time-Based Visualization for Agent-Based Codebase Interactions

Takuto Kawamoto[1,a)] Yoshiki Higo[1,b)] Raula Gaikovina Kula[1,c)]

Abstract: AI coding agents are becoming part of developers' workflows, but their behavior is difficult to understand from final code changes alone. During a task, agents interact with software repositories through sequences of actions such as searching for files, reading code, editing programs, and running tests or build commands. These interactions, together with token usage, are often recorded in console logs, but raw logs are difficult for developers to inspect. In this paper, we propose Agent ATO (Agentic Trajectory Observer), a tool for visualizing AI coding agent interaction timelines from console logs. Agent ATO extracts agent interactions, classifies them by command or tool type, and visualizes them as timelines. In addition to an all-interaction timeline, Agent ATO provides filtered timelines that emphasize file discovery, file reading, file editing, and execution while preserving surrounding context. We apply Agent ATO in case studies to examine how developers can inspect agent actions in software repositories. Future work will apply Agent ATO to more agents, tasks, and development environments, and will evaluate whether it reduces the effort needed to compare trajectories.

1. Introduction

AI coding agents are increasingly used to automate software engineering tasks by interacting with development environments. Unlike one-shot uses of large language models, these agents iteratively search repositories, inspect files, edit code, and run tests through an agent-computer interface [1]. Their work processes are often recorded in console logs, which contain tool calls, commands, file inspections, edits, execution results, and errors. These logs can reveal how the agent proceeded toward final code changes, such as whether it explored relevant files before editing, validated its changes, or returned to exploration after failures.

Prior studies have analyzed agent trajectories and traces to understand agent behavior beyond final outputs. In software engineering, empirical studies have examined thought-action-result trajectories and execution traces to characterize structural properties, recurring action patterns, and decision-making pathways in automated program repair and issue-resolution tasks [2, 3]. More broadly, trace-based failure analysis has been used to identify failure modes in LLM agent systems [4], and large-scale analyses of coding-agent sessions have shown that misalignment can arise across multiple stages of the work process, including project understanding, implementation, command execution, and self-reporting [5]. In addition to empirical analyses, interactive and visual approaches have been explored to help users inspect agent behavior. Some work supports debugging and steering by helping users navigate long histories in multi-agent systems [6], while other work visualizes execution events to analyze the temporal evolution of LLM-based autonomous systems [7]. Visual analytics has also been applied to coding agents to compare agent behavior across trials [8]. In parallel, visual tools for LLMs have supported prompt exploration, response comparison, and iterative prompt improvement [9–11]. Together, these studies demonstrate the value of trace analysis and visual support for understanding LLM and agent behavior.

However, console logs do not provide an easily inspectable view of interaction types or ordering, even when they record events in temporal order. Developers and researchers must still read long logs sequentially to identify where the agent searched, read, edited, or executed commands. They must also find the segments that deserve closer analysis. This makes it difficult to notice process-level cues, such as missing validation after edits or no further exploration after failures. A clearer view of agent execution can help developers inspect past runs and refine future runs.

We propose *Agent ATO*, a timeline visualization that transforms console logs into classified interaction sequences for inspecting and comparing agent trajectories. We focus on interaction histories observable from console logs. In this paper, an interaction is a coherent unit of agent activity initiated by an LLM response and realized through tool use, such as repository navigation, file reading, file editing, or command execution. A sequence of interactions forms an agent trajectory. Agent ATO shows an overview timeline of

[1] The University of Osaka, Suita, Osaka 565–0871, Japan
[a)] tk-kawam@ist.osaka-u.ac.jp
[b)] higo@ist.osaka-u.ac.jp
[c)] raula-k@ist.osaka-u.ac.jp

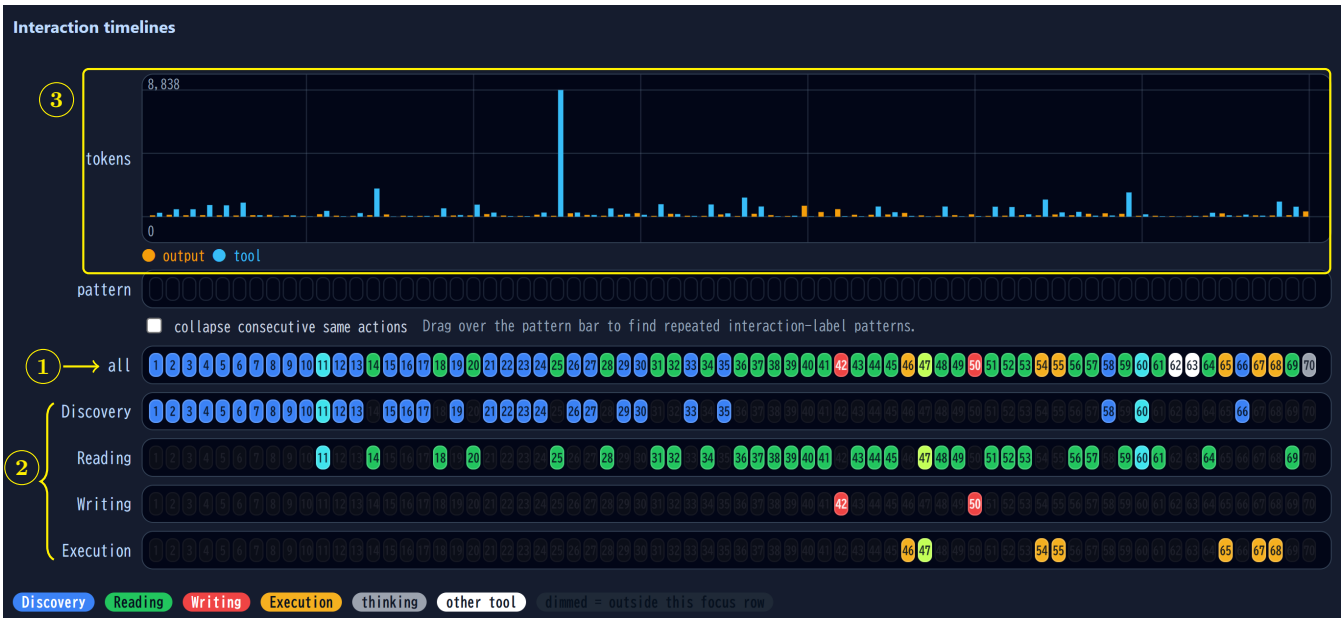

Fig. 1 Overview of our prototype visualization. Each small rectangle represents an interaction reconstructed from the Pi coding agent event log.

all interactions and filtered timelines for four tags: Discovery, Reading, Writing, and Execution. Because a single interaction may belong to multiple tags, the filtered timelines let users focus on one perspective while preserving temporal context. We implemented a prototype for the Pi coding agent [*1].

To demonstrate Agent ATO, we conduct two case studies using repeated executions of the same coding task under the same prompt and environment. The first case study shows that executions with similar repair directions can differ in validation behavior: one attempt formed repeated edit-validation cycles, whereas another edited without corresponding test execution. The second case study shows that high token usage can have different causes: some spikes come from long LLM-output segments that repeatedly reconsider repair policies, while others come from large tool results produced by file-reading operations. Together, these cases show how Agent ATO helps users compare trajectories and identify process-level cues that are difficult to see from final patches or raw logs alone.

This paper makes the following three contributions. First, we define AI coding agent behavior as an interaction trajectory composed of observable software-development operations. Second, we propose a timeline visualization that combines detailed command/tool types with filtered views based on Discovery, Reading, Writing, and Execution. Third, we demonstrate through two case studies how the visualization supports comparison of repeated executions, exposes differences in edit-validation behavior, and distinguishes token usage caused by LLM output from token usage caused by tool results. Rather than automating diagnosis, the visualization provides a basis for observing and comparing AI coding agent trajectories.

[*1] `https://github.com/earendil-works/pi`

2. Our Approach

Fig. 1 shows the Agent ATO visualization. Each small rectangle represents one reconstructed interaction, and time advances from left to right. The row marked ① is the all-interaction timeline, which presents the complete trajectory. The rows marked ② are filtered timelines for Discovery, Reading, Writing, and Execution. These timelines highlight interactions for one analytical perspective while keeping the remaining interactions visible in gray for context. The graph marked ③ shows token usage aligned with the same interaction sequence and separates LLM-output tokens from tool-result tokens. We describe these three parts below.

2.1 ① All-interaction timeline

The all-interaction timeline provides the complete sequence of reconstructed interactions for a task. Agent ATO takes event logs collected during AI coding agent executions as input. These logs include LLM messages, tool-call events, bash commands, tool results, token usage, and timestamps. Agent ATO uses these events to reconstruct the order of commands and tool uses while retaining detailed messages and outputs for drill-down inspection.

Each mark represents one interaction, and the marks are arranged from left to right according to execution order. By scanning the sequence, users can see whether the agent explored and read files before writing, whether writing occurred early, and whether execution followed writing.

2.2 ② Filtered timeline

The filtered timelines reuse the same interaction sequence but emphasize four analytical perspectives:

- **Discovery** highlights operations for finding relevant files or information, such as `ls`, `find`, `rg`, and `grep`

`-l`. These interactions are shown in blue.
- **Reading** highlights operations for inspecting file contents or diffs, such as `read`, `cat`, `grep`, and `git diff`. These interactions are shown in green.
- **Writing** highlights operations that modify files or repository state, such as `edit`, `write`, and `rm`. These interactions are shown in red.
- **Execution** highlights validation or execution operations, such as running tests, builds, linters, type checkers, or scripts. These interactions are shown in yellow.

Interactions that do not match the selected perspective remain visible in gray. This design lets users focus on one view, such as Reading or Execution, without losing the surrounding temporal context. When an interaction belongs to multiple perspectives, Agent ATO shows it using a blended color derived from the corresponding tags.

2.3 ③ Token-usage View

The token-usage graph is aligned with the same interaction sequence as the timelines. For each interaction, it separates token usage into LLM-output tokens and tool-result tokens. This distinction helps users tell whether high token usage came from a long agent response or from a large tool result. Because the graph is aligned with the operation sequence, users can inspect not only where token usage is high, but also what type of activity caused it.

3. Technical Details

In this section, we describe the technical implementation and user interactions behind the visualization.

3.1 Implementation

We implemented a prototype consisting of two components: a TypeScript extension for collecting Pi coding agent events and a Python script for generating the visualization. The TypeScript extension records events during Pi agent execution and saves them as a JSONL file. The recorded events include Pi turn boundaries, LLM messages, tool-call events, bash commands, tool results, token usage, and timestamps.

In Pi, a turn consists of one LLM response and the tool calls triggered by that response. A single user prompt may therefore produce multiple Pi turns. In our implementa-

Table 1 Rules for assigning focus filters to interactions.

Filter	Source	Rule
Discovery	bash	`ack`
	bash	`fd`
	bash	`find`
	bash	`grep -l`
	bash	`grep --files-with-matches`
	bash	`ls`
	bash	`npm root`
	bash	`npm list`
	bash	`npm ls`
	bash	`rg`
	bash	`tree`
	bash	`wc`
	bash	`which`
Reading	tool	`read`
	bash	`cat`
	bash	`git diff`
	bash	`grep` (without `-l` or `--files-with-matches`)
	bash	`less`
	bash	`more`
	bash	`sed -n`
	bash	`sed -i`
	bash	`sed --in-place`
Writing	tool	`edit`
	tool	`write`
	bash	`rm`
	bash	`sed` (without `-n`)
	bash	`sed --in-place`, `sed -i`
	bash	Python inline script with `open(..., 'w')` and `write(...)`
Execution	bash	`bun run`
	bash	`node`
	bash	`npm exec`
	bash	`npm install`, `npm i`
	bash	`npm run`
	bash	`npm test`
	bash	`npx`
	bash	`python3`, `python`, `py`
	bash	`pytest`
	bash	`pnpm`
	bash	`tox`
	bash	`uv run`
	bash	`yarn`

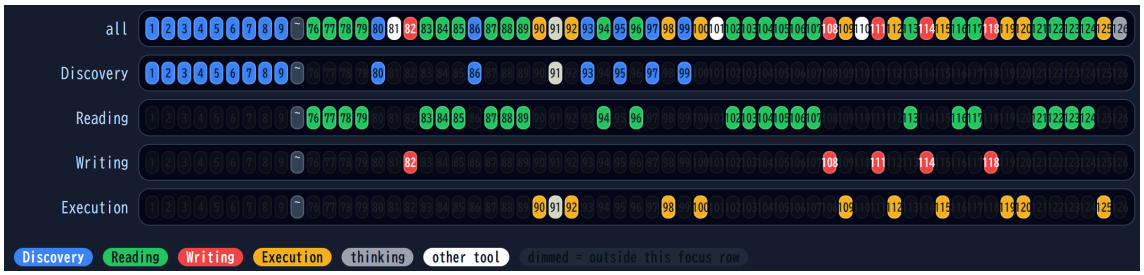

Fig. 2 Case 1, Attempt 1. Writing interactions are followed by Execution interactions, showing repeated edit-validation cycles.

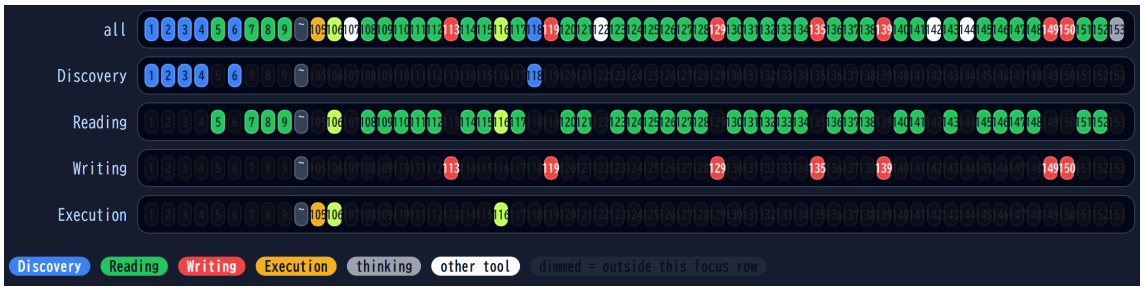

Fig. 3 Case 1, Attempt 2. The run contains several Writing interactions after context gathering, but these edits are not followed by corresponding Execution interactions.

tion, we treat each interval from `turn_start` to `turn_end` as one interaction. For each interaction, the generation script assigns a representative label based on the executed bash command, pipeline, or tool name. For example, it uses labels such as `find | grep | head`, `git diff`, or `npm test` to summarize the concrete operation performed in the interaction.

It then generates a static HTML file that contains the all-interaction timeline, filtered timelines, token-usage graph, and interactive detail views. The generated HTML file can be opened in a web browser, allowing users to inspect trajectories without running a separate visualization server.

3.2 Focus Filter Assignment

To generate the filtered timelines, the script assigns focus-filter tags to each interaction. The current implementation uses rule-based matching over Pi tool use and bash commands. Table 1 summarizes the rules used for assigning the Discovery, Reading, Writing, and Execution filters. These rules were determined through discussion among the authors after inspecting commands observed in several agent executions.

The Discovery filter captures commands that help the agent locate files, directories, dependencies, or matching occurrences, such as `find`, `ls`, `rg`, and `grep -l`. The Reading filter captures interactions that inspect file contents or textual output without directly modifying project files, including the `read` tool, `cat`, `git diff`, and `sed -n`. The Writing filter captures interactions that modify files or remove project artifacts, such as the `edit` and `write` tools, `rm`, in-place `sed` commands, and inline Python scripts that write to files. The Execution filter captures commands that run programs, tests, package managers, or build-related tasks, such as `npm test`, `npm run`, `pytest`, `node`, and `python3`.

An interaction may receive multiple tags when a command matches multiple rules.

3.3 User Interactivity

The visualization allows users to move from the overview to detailed logs. When a user hovers over an interaction, the visualization shows its representative label, such as `find | grep | head`, `git diff`, or `npm test`, and highlights other interactions with the same label. This allows users to identify the concrete command or tool use behind each interaction and to notice repeated operations, such as recurring searches or repeated test executions, without placing all labels directly on the timeline. Clicking an interaction opens a detail panel with the representative label, timing, token usage, tool results, and command outputs. Thus, the visualization does not replace the original log; it helps users find the parts of the log that deserve closer inspection.

Users can also select a consecutive sequence of interactions and search for the same operation pattern elsewhere in the trajectory. This supports visual inspection of recurring interaction patterns.

4. Case Studies

This section shows how Agent ATO can compare trajectories generated by the same AI coding agent under the same task, prompt, and execution environment.

4.1 Study Design

We do not assess whether the task was completed successfully. Instead, we examine process-level differences that are difficult to understand from the final patch alone. These include when the agent makes edits, whether it validates those

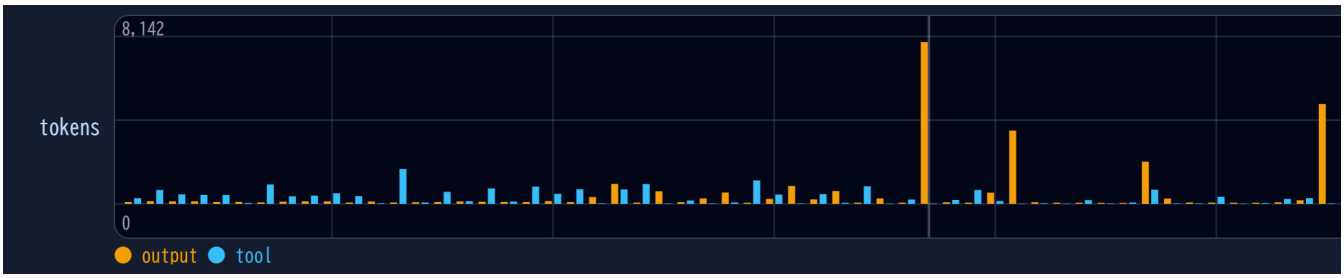

Fig. 4 Case 2, Attempt 1. The largest spikes indicate that high token usage is mainly caused by long LLM output.

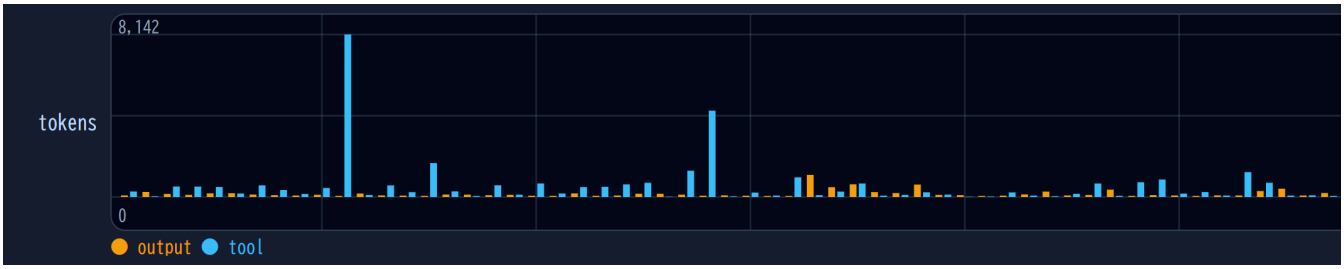

Fig. 5 Case 2, Attempt 2. In contrast to Attempt 1, the largest spikes indicate that high token usage is mainly caused by large tool results.

edits, and which interactions use the most tokens. In each case, we first use the timeline to identify a potential difference. We then inspect the detailed labels or logs to interpret that difference.

We used two real GitHub issues as target tasks: issue `#28383`, "Album ordering resets to stale order after opening asset viewer," in IMMICH [*2], and issue `#74530`, "`Formatting` section of Y-axis settings has scroll but it shouldn't," in METABASE [*3]. For each issue, we executed the same Pi coding agent 10 times with the same prompt and execution environment. During each run, we collected the Pi event log, including turn boundaries, LLM messages, tool calls, bash commands, tool results, token usage, and timestamps. Each log was then reconstructed as an interaction sequence using the method described in Section 2.

4.2 Case 1: Comparing Trajectories for the Same Issue

The first case study targets the IMMICH issue `#28383`, a client-side state management and navigation bug. Specifically, this issue involves a defect where changing the display order of photos in an album does not immediately apply to the UI state. As a result, when a user opens the asset viewer, the album ordering resets to a stale state, requiring a manual page refresh to correct the display.

Across the 10 executions, most attempts reached files related to the issue, but their repair strategies were not uniform. We therefore focus on four attempts that treated the bug mainly as a page-state problem and modified overlapping locations. This subset lets us compare process differences among executions that followed a similar repair direction.

Figs. 2 and 3 compare two representative attempts from this subset. In both attempts, Writing interactions appear after earlier Discovery and Reading interactions, indicating that the agent gathered codebase context before editing. The difference appears in the relationship between the Writing and Execution timelines. In Fig. 2, red Writing interactions are repeatedly followed by yellow Execution interactions, especially in the latter half of the trajectory. In Fig. 3, several Writing interactions appear, but the Execution timeline does not show corresponding validation after these edits.

Inspecting the representative labels of the Execution interactions in Fig. 2 showed that these executions were test commands. Thus, the visualization exposes a process difference that is not captured by the fact that both attempts pursued a similar repair direction: Attempt 1 formed edit-validation cycles, whereas Attempt 2 did not validate its edits within the observed trajectory.

Fig. 2 also shows that the first validation required more intermediate interactions than later validations. After the first Writing interaction, several additional interactions occurred before the first test execution; after later Writing interactions, test execution appeared immediately afterward. This suggests that the agent first had to identify how to run a relevant test and then reused that command in later cycles. This observation illustrates prompt debugging: explicitly requiring validation after editing and asking the agent to identify the relevant test command early in the run.

[*2] `https://github.com/immich-app/immich/issues/28383`
[*3] `https://github.com/metabase/metabase/issues/74530`

Case 1 Finding

Agent ATO exposed that, among attempts with similar repair directions, one repeatedly validated edits with test execution while another edited without subsequent validation.

4.3 Case 2: Token Consumption and Its Causes

The second case study targets the METABASE issue `#74530`, a UI bug in which the `Formatting` section of the Y-axis settings scrolls unnecessarily. This frontend defect causes a scrollbar to appear in the formatting panel even when the content fits within the designated area, creating an unintended visual artifact in the interface. This case focuses on the token-usage graph aligned with the interaction sequence. Among the 10 executions, we selected two attempts that both had high token usage but differed in source. Figs. 4 and 5 show the token-usage graphs for these attempts. Attempt 1 contains large spikes in LLM-output tokens, whereas Attempt 2 contains large spikes in tool-result tokens.

In Fig. 4, the dominant spikes are orange, indicating that the corresponding interactions consumed many LLM-output tokens. Inspecting the detailed log for the largest spike showed a long thinking segment in which the agent repeatedly reconsidered competing repair policies. This does not show that the final modification was wrong; rather, it identifies a token-efficiency issue. From a prompt-debugging perspective, such a pattern suggests instructing the agent to summarize candidate policies, select one, and continue concisely when repeated reconsideration occurs. At the system level, unusually long thinking segments could also trigger interruption, summarization, or user confirmation.

Fig. 5 shows a different pattern: the largest spikes are blue, meaning that they come from tool-result tokens rather than LLM-output tokens. The detailed logs showed that these spikes were caused by large Reading results from file inspection. This is not necessarily an error, because the relevant information may reside in a large file. However, it suggests different interventions: prompts can ask the agent to check file size and read partial ranges, while agent designs can search for relevant identifiers first, read only necessary ranges, or summarize long tool results before continuing.

Case 2 Finding

Agent ATO showed that high token usage can arise from different sources: long LLM-output segments suggest reasoning-control issues, whereas large tool results suggest context-management or partial-reading issues.

5. Discussion

Our analysis of Case Study 2, the METABASE UI polish issue, highlights a critical challenge in evaluating AI coding agents. In software repair tasks, executable tests are commonly used to determine whether a generated patch succeeds or fails. For example, coding-agent benchmarks such as SWE-bench evaluate generated patches by checking whether they pass issue-specific tests and preserve existing behavior [12]. This test-based evaluation is useful because it provides an executable and reproducible success criterion. However, our case study shows that UI-related tasks can expose limitations of evaluating agents only through final outcomes or test results.

5.1 Diverse Approaches

In the 10 executions addressing the UI scroll bug, the AI agent conceptually demonstrated three main approaches. The first approach unconditionally hides the scrollbar. The second approach conditionally removes the scroll capability only under specific conditions. The third approach reduces the internal margins of trailing elements. In contrast, the actual merged pull request solved the issue by providing sufficient vertical space, ensuring that the settings content fits entirely without overflowing and thereby preventing unnecessary scroll behavior.

It is crucial to note that the actual pull request should not be treated as the absolute ground truth. The fact that the AI agent's proposed patches differ from the actual fix does not immediately render them incorrect. In front-end UI adjustments, different implementation strategies—such as modifying container heights, overriding CSS properties, or adjusting internal margins—can resolve the same visual artifact. Thus, several of the agent's attempts could be considered plausible alternative approaches if they remove the user-facing scroll behavior without introducing functional or layout regressions.

5.2 Limitations of Test-Based Evaluation in UI Tasks

The existence of multiple valid solutions poses a challenge for automated evaluation datasets when applied to front-end tasks. Evaluation methods relying on strict patch or diff matching against an actual fix can incorrectly penalize agents that find alternative valid UI solutions.

Execution-based evaluation using tests also has limitations for UI polish issues. Although tests can confirm that a patch does not break covered behavior, they do not necessarily confirm that a user-visible visual defect has been resolved. UI bugs such as unexpected scrollbars, spacing problems, or pixel-level margin misalignments may depend on rendered layout and interactive behavior that are not fully captured by ordinary unit tests. Prior work on GUI design violations and web presentation failures has therefore used visual inspection or computer-vision-based analysis of rendered interfaces to detect UI defects [13, 14]. Consequently, passing the existing test suite should be interpreted as evidence that the agent preserved tested functionality, not as sufficient evidence that the visual issue was fixed or that no new layout regression was introduced.

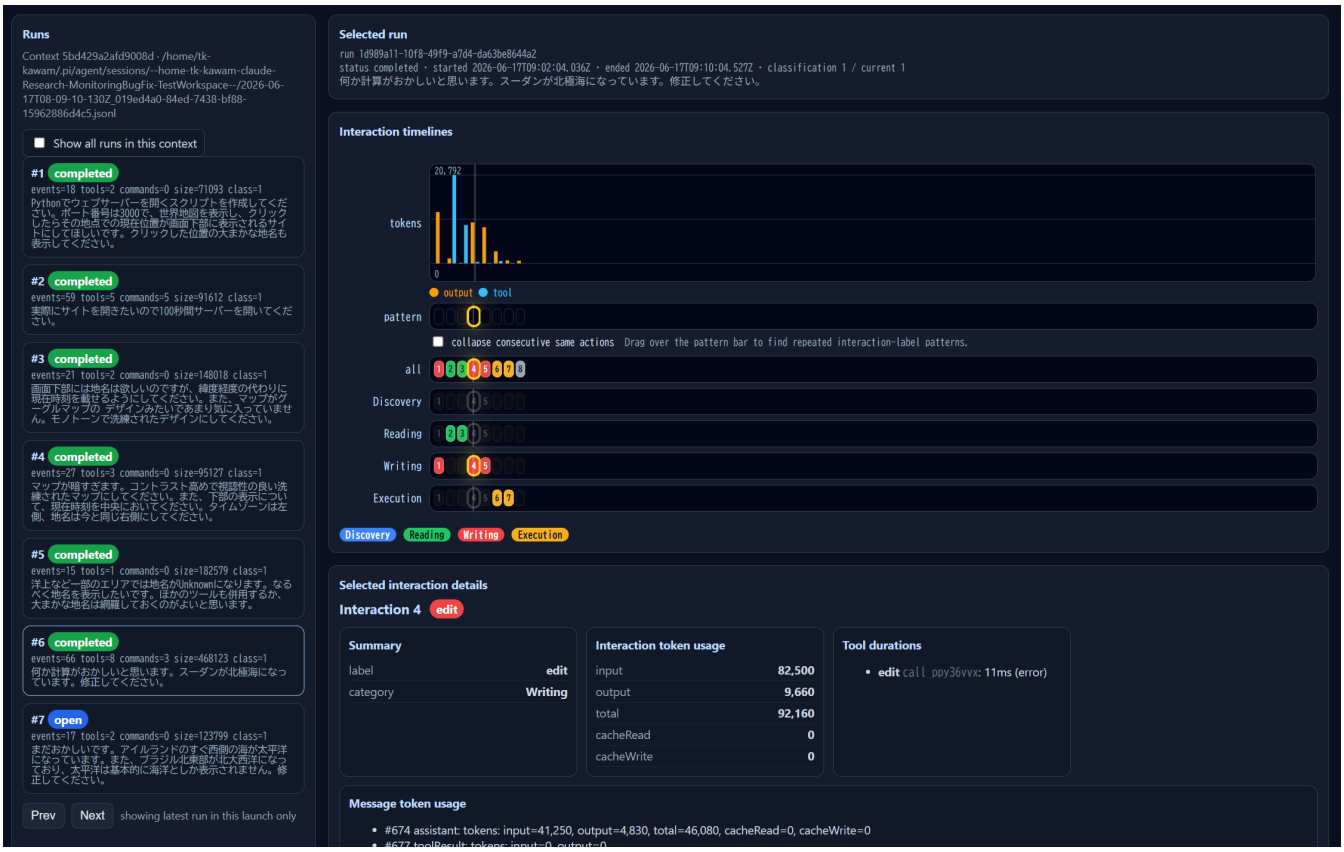

Fig. 6 Latest Agent ATO prototype with a run-selection sidebar. Users can select an individual agent run from the sidebar and inspect the corresponding token-usage graph, interaction timelines, filtered timelines, and interaction details.

5.3 Interaction Observation as a Evaluation Perspective

Given the difficulty of evaluating AI agents based solely on their final output patches or test results, observing the agent's interaction process offers a valuable complementary perspective. Although the agent's final patches in Case Study 2 differed from the actual fix, our timeline visualization revealed that in many executions, the agent navigated to relevant files. In several attempts, the agent even targeted and edited the same file that was modified in the actual pull request.

Interaction visualization provides a detailed timeline of when the agent discovered relevant files, how long it spent reading surrounding context, and when it decided to make edits. Even if the final output differs from the reference fix, confirming that the agent performed fault localization and attempted modifications based on a reasonable hypothesis provides useful evidence about its problem-solving process. In domains such as front-end development, where valid solutions can be diverse and test-based success criteria may be incomplete, analyzing the interaction trajectory can provide an additional evaluation dimension that final patches and test results alone cannot capture.

6. Conclusion and Future Work

6.1 Summary

In this paper, we proposed Agent ATO, a timeline-based visualization for inspecting AI coding agent trajectories reconstructed from console logs. Agent ATO addresses the difficulty of understanding long console logs by reorganizing agent actions into time-aligned views of concrete interactions, focus-filtered operation categories, and token usage. This design helps users observe how an agent explored files, read information, edited code, and validated changes over the course of a trajectory.

6.2 Limitations

The current prototype has several limitations. Agent ATO has so far been applied to one coding agent and a small number of repair tasks. In addition, the focus-filter tags are assigned by rule-based matching over commands and tool names. The current rules were manually defined by the authors based on commands observed in several agent executions, and they do not cover the full diversity of commands used across agents, repositories, and development environments. Moreover, some commands can play different roles depending on their arguments and context.

For example, `python3` commands may run tests or scripts as Execution, but they may also modify files when an inline script opens a file in write mode and writes content to it. The current implementation includes rules for assigning Writing tags to such observed file-writing patterns, but exhaustively covering all argument-dependent behaviors remains difficult. Thus, the current tagging should be re-

garded as an extensible operational classification rather than a complete semantic interpretation of command behavior. Future work should extend the rule set, examine how the rules generalize to other agents and repositories, and explore finer-grained categories, especially within Execution, where tests, builds, package installation, linting, type checking, and program execution may need to be distinguished.

6.3 Future Work

We are extending Agent ATO to support a more realistic prompt-debugging workflow. In software development with AI coding agents, developers often inspect the result after each agent run, enter a follow-up prompt, and continue the task through multiple rounds of prompting and agent work. To make these repeated runs easier to distinguish, the latest version of Agent ATO adds a run-selection sidebar, as shown in Fig. 6. The sidebar lists individual runs in the same launch context, together with their status, event counts, tool-call counts, command counts, log size, and the prompt text. When a user selects a run, Agent ATO updates the selected-run summary, interaction timelines, token-usage graph, filtered timelines, and interaction-detail panel for that run. This design is intended to help users compare how the agent behaved after different prompts, inspect whether a revised prompt changed the timing of exploration, reading, editing, or validation, and relate each trajectory to the developer's iterative prompting process.

Using this extended version, we plan to conduct a user study to evaluate how effectively the visualization supports prompt debugging. While our case studies discuss possible prompt improvements based on our observations, the user study will examine how developers actually use the visualization in an iterative prompting workflow. In particular, we aim to investigate what improvements developers can actually make when using the timeline, filtered views, token-usage graph, and run-selection sidebar to understand agent behavior, identify problematic operation sequences, and revise subsequent prompts. Through this study, we aim to assess the practical usefulness of the proposed visualization approach and continue refining Agent ATO as a tool for inspecting and improving AI coding agent trajectories.

Acknowledgments This work was supported by JSPS KAKENHI Grant Number JP24H00692, JP25K03102, JP26K02889, JP26H02500.

References

[1] Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K. and Press, O.: SWE-agent: agent-computer interfaces enable automated software engineering, *Proceedings of the 38th International Conference on Neural Information Processing Systems*, NIPS '24, Red Hook, NY, USA, Curran Associates Inc. (2024).

[2] Bouzenia, I. and Pradel, M.: Understanding Software Engineering Agents: A Study of Thought-Action-Result Trajectories, *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE Press, p. 2846–2857 (online), DOI: 10.1109/ASE63991.2025.00234 (2025).

[3] Ceka, I., Mitchell, H., Pujar, S., Buratti, L., Ramji, S., Yang, J., Kaiser, G. and Ray, B.: Understanding Automated Program Repair Agents Through the Lens of Traceability: An Empirical Study (2026).

[4] Cemri, M., Pan, M. Z., Yang, S., Agrawal, L. A., Chopra, B., Tiwari, R., Keutzer, K., Parameswaran, A., Klein, D., Ramchandran, K., Zaharia, M., Gonzalez, J. E. and Stoica, I.: Why Do Multi-Agent LLM Systems Fail?, *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, (online), available from ⟨https://openreview.net/forum?id=fAjbYBmonr⟩ (2026).

[5] Tang, N., Chen, C., Xu, G., Shi, Y., Huang, Y., McMillan, C., Dong, T. and Li, T. J.-J.: How Coding Agents Fail Their Users: A Large-Scale Analysis of Developer-Agent Misalignment in 20,574 Real-World Sessions (2026).

[6] Epperson, W., Bansal, G., Dibia, V. C., Fourney, A., Gerrits, J., Zhu, E. E. and Amershi, S.: Interactive Debugging and Steering of Multi-Agent AI Systems, *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, CHI '25, New York, NY, USA, Association for Computing Machinery, (online), DOI: 10.1145/3706598.3713581 (2025).

[7] Lu, J., Pan, B., Chen, J., Feng, Y., Hu, J., Peng, Y. and Chen, W.: AgentLens: Visual Analysis for Agent Behaviors in LLM-Based Autonomous Systems, *IEEE Transactions on Visualization and Computer Graphics*, Vol. 31, No. 8, pp. 4182–4197 (online), DOI: 10.1109/TVCG.2024.3394053 (2025).

[8] Wang, J., Chen, Y., Pan, M., Yeh, C.-C. M. and Das, M.: Illuminating LLM Coding Agents: Visual Analytics for Deeper Understanding and Enhancement (2025).

[9] Strobelt, H., Webson, A., Sanh, V., Hoover, B., Beyer, J., Pfister, H. and Rush, A. M.: Interactive and Visual Prompt Engineering for Ad-hoc Task Adaptation with Large Language Models , *IEEE Transactions on Visualization & Computer Graphics*, Vol. 29, No. 01, pp. 1146–1156 (online), DOI: 10.1109/TVCG.2022.3209479 (2023).

[10] Mishra, A., Danzy, B., Soni, U., Arunkumar, A., Huang, J., Kwon, B. C. and Bryan, C.: PromptAid: Visual Prompt Exploration, Perturbation, Testing and Iteration for Large Language Models , *IEEE Transactions on Visualization & Computer Graphics*, Vol. 31, No. 10, pp. 6946–6962 (online), DOI: 10.1109/TVCG.2025.3535332 (2025).

[11] Arawjo, I., Swoopes, C., Vaithilingam, P., Wattenberg, M. and Glassman, E. L.: ChainForge: A Visual Toolkit for Prompt Engineering and LLM Hypothesis Testing, *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, CHI '24, New York, NY, USA, Association for Computing Machinery, (online), DOI: 10.1145/3613904.3642016 (2024).

[12] Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O. and Narasimhan, K. R.: SWE-bench: Can Language Models Resolve Real-world Github Issues?, *The Twelfth International Conference on Learning Representations*, (online), available from ⟨https://openreview.net/forum?id=VTF8yNQM66⟩ (2024).

[13] Moran, K., Li, B., Bernal-Cárdenas, C., Jelf, D. and Poshyvanyk, D.: Automated reporting of GUI design violations for mobile apps, *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, New York, NY, USA, Association for Computing Machinery, p. 165–175 (online), DOI: 10.1145/3180155.3180246 (2018).

[14] Mahajan, S. and Halfond, W. G. J.: WebSee: A Tool for Debugging HTML Presentation Failures, *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 1–8 (online), DOI: 10.1109/ICST.2015.7102638 (2015).