

Agent ATO: Visualizing Agent Interaction Timelines from Logs

Takuto Kawamoto*, Yoshiki Higo*, Raula Gaikovina Kula*

*The University of Osaka, Japan

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 illustrate how Agent ATO may help developers inspect and compare agent actions using selected runs from two repair tasks. 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.

Index Terms—AI Coding Agent, Agentic Trajectory, Prompt Debugging

I. 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. Some work examines software engineering agent trajectories through thought-action-result sequences and execution traces in automated program repair agents [2], [3]. Other work analyzes trajectories to identify why LLM agent systems fail [4]. Tool support has also been motivated by the need to debug agent workflows and review long agent histories [5]. In parallel, visual tools for LLMs have compared responses across prompts or trials for prompt improvement [6]–[8]. Recent work on coding agents also shows that comparing agent behavior across trials is a relevant visual analytics problem [9].

However, although console logs record events in temporal order, they do not clearly present interaction types or their

ordering. Developers and researchers must still read long logs to locate searches, reads, edits, executions, and segments requiring 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. Similar problems have arisen in other software-development contexts, where event logs have been aggregated and visualized to support behavioral understanding, such as in debugging activity analysis [10].

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 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 demon-

¹<https://github.com/earendil-works/pi>

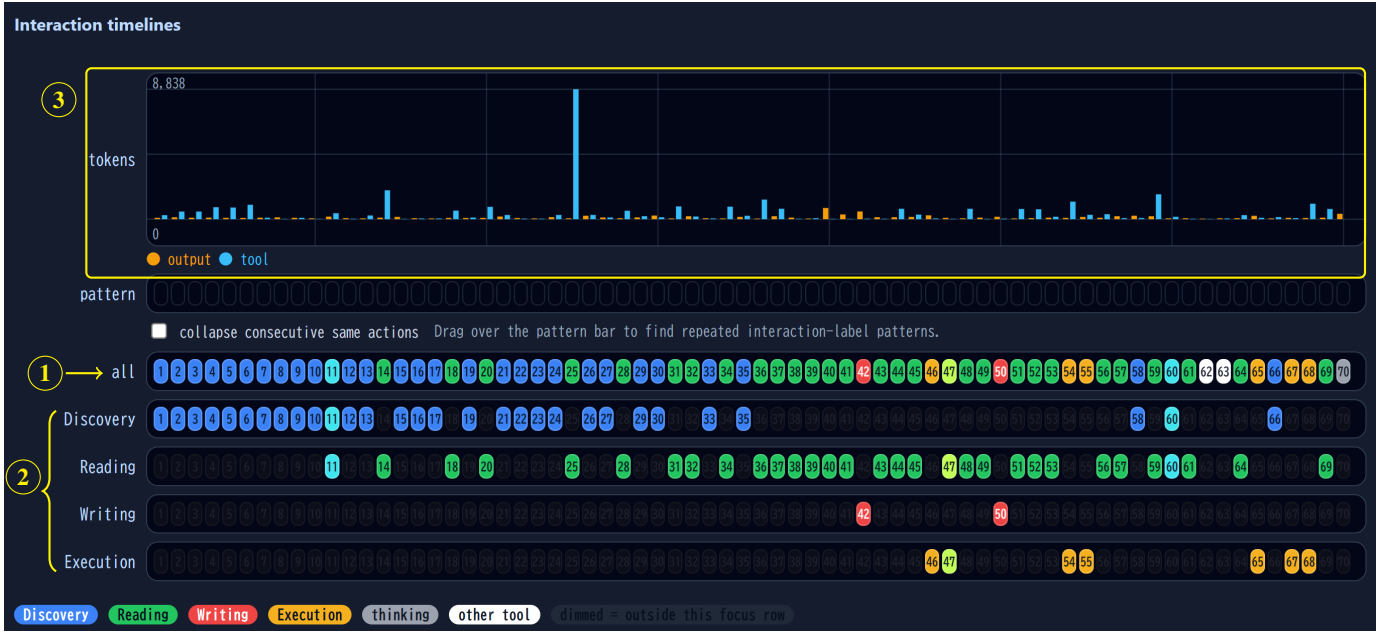


Fig. 1: Overview of our prototype visualization. Each small rectangle represents an interaction reconstructed from the Pi coding agent event log.

strate 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.

II. AGENT ATO (AGENTIC TRAJECTORY OBSERVER)

Fig. 1 shows the Agent ATO visualization. Each rectangle represents one interaction, ordered from left to right. We call this sequential representation a timeline, although it does not represent elapsed time or duration. 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.

A. 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.

B. 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.

C. 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

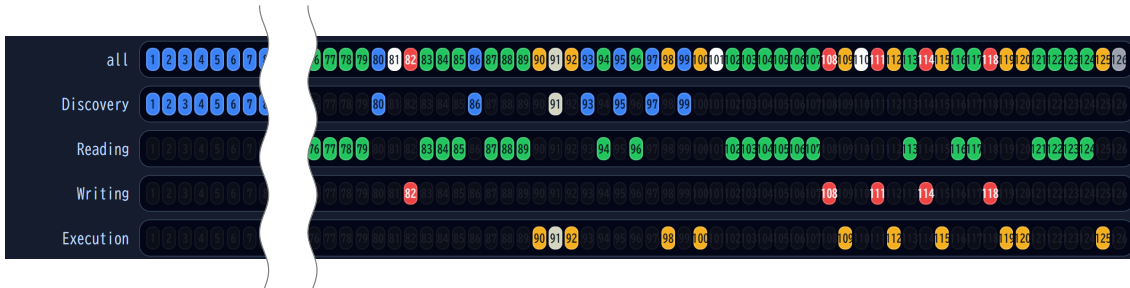


Fig. 2: Case 1, Attempt 1. Writing interactions are followed by Execution interactions, showing repeated edit-validation cycles.

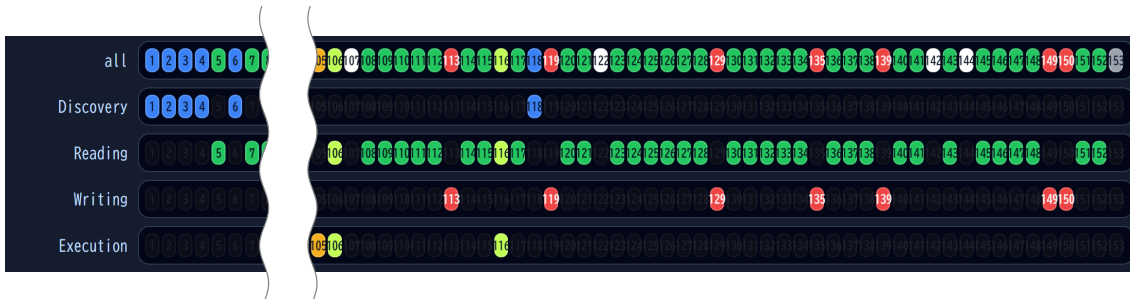


Fig. 3: Case 1, Attempt 2. The run contains several Writing interactions after context gathering, but these edits are not followed by corresponding Execution interactions.

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.

III. TECHNICAL DETAILS

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

A. 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 implementation, 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.

The script also assigns one or more tags—Discovery, Reading, Writing, and Execution—to each interaction based on rule-based matching of commands and tool names. 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.

B. 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.

IV. CASE STUDIES

This section illustrates how Agent ATO compares selected runs under the same task, prompt, and environment.

A. 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

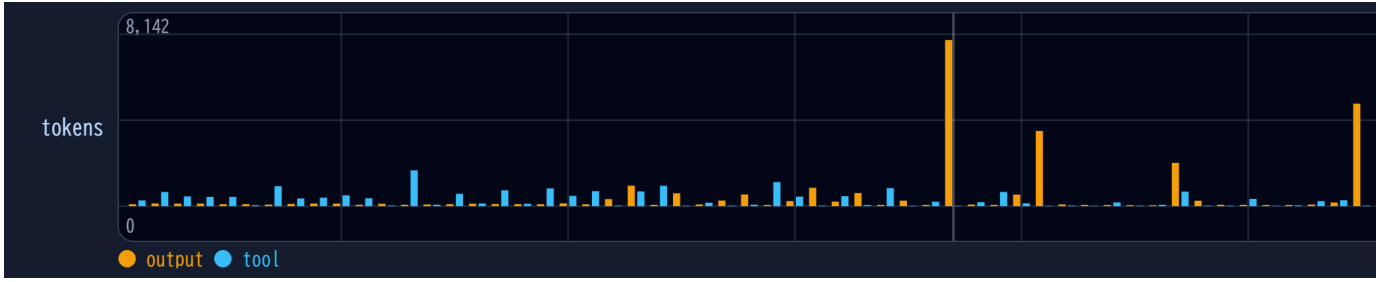


Fig. 4: Case 2, Attempt 1. The largest spikes indicate that high token usage is mainly caused by long LLM output.

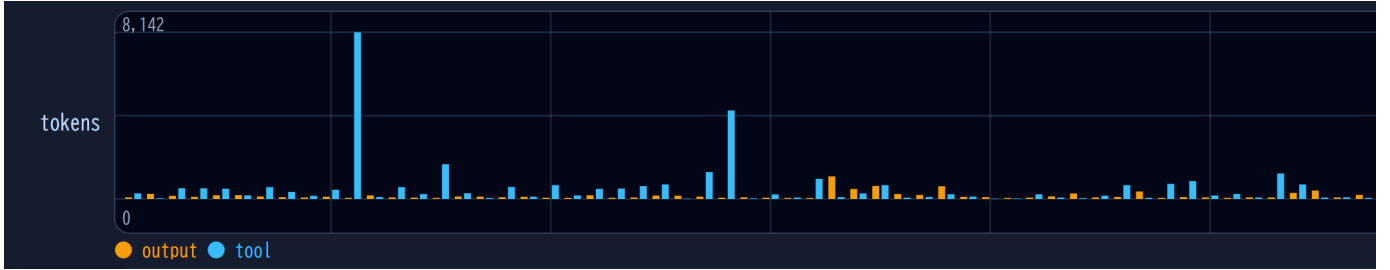


Fig. 5: Case 2, Attempt 2. In contrast to Attempt 1, the largest spikes indicate that high token usage is mainly caused by large tool results.

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 IMMICH², and issue #74530, “Formatting section of Y-axis settings has scroll but it shouldn’t,” in METABASE³. 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 II. We have made the replication package for this experiment publicly available on GitHub.⁴

B. 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.

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 suggests a potential prompt-debugging approach: requiring early test identification and validation after editing.

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.

²<https://github.com/immich-app/immich/issues/28383>

³<https://github.com/metabase/metabase/issues/74530>

⁴<https://anonymous.4open.science/r/VISSOFT2026ReplicationPackage-0E76>

C. 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 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.

V. CONCLUSION AND FUTURE WORK

In this paper, we proposed Agent ATO, a timeline-based visualization for inspecting AI coding agent trajectories reconstructed from console logs. Agent ATO combines an all-interaction timeline, filtered timelines for Discovery, Reading, Writing, and Execution, and an aligned token-usage graph. Its goal is not to infer hidden intentions or automatically diagnose failures, but to help users observe what the agent did, in what order, and where operation-level or token-level differences occur.

Our case studies showed two uses of this representation. First, the timelines exposed differences in validation behavior, distinguishing repeated edit-validation cycles from edits without subsequent test execution. Second, the token-usage graph separated high token usage caused by long LLM output

from high token usage caused by large tool results. These results suggest that trajectory visualization can guide detailed log inspection and support prompt-debugging and agent-design discussions.

Current evidence is limited to one agent, two tasks, selected runs, and unvalidated rule-based tags. Future work will evaluate tagging accuracy, compare Agent ATO with raw-log inspection, and study whether trajectory observations lead to effective prompt or agent revisions.

ACKNOWLEDGMENT

This work was supported by JSPS KAKENHI Grant Number JP24H00692, JP25K03102, JP26K02889, JP26H02500, JP23K28065.

REFERENCES

- [1] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: agent-computer interfaces enable automated software engineering," in *Proceedings of the 38th International Conference on Neural Information Processing Systems*, ser. NIPS '24. Red Hook, NY, USA: Curran Associates Inc., 2024.
- [2] I. Bouzenia and M. Pradel, "Understanding software engineering agents: A study of thought-action-result trajectories," in *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Press, 2025, p. 2846–2857. [Online]. Available: <https://doi.org/10.1109/ASE63991.2025.00234>
- [3] I. Ceka, H. Mitchell, S. Pujar, L. Buratti, S. Ramji, J. Yang, G. Kaiser, and B. Ray, "Understanding automated program repair agents through the lens of traceability: An empirical study," 2026. [Online]. Available: <https://arxiv.org/abs/2506.08311>
- [4] M. Cemri, M. Z. Pan, S. Yang, L. A. Agrawal, B. Chopra, R. Tiwari, K. Keutzer, A. Parameswaran, D. Klein, K. Ramchandran, M. Zaharia, J. E. Gonzalez, and I. Stoica, "Why do multi-agent LLM systems fail?" in *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2026. [Online]. Available: <https://openreview.net/forum?id=fAjbYBmonr>
- [5] W. Epperson, G. Bansal, V. C. Dibia, A. Fourney, J. Gerrits, E. E. Zhu, and S. Amershi, "Interactive debugging and steering of multi-agent ai systems," in *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, ser. CHI '25. New York, NY, USA: Association for Computing Machinery, 2025. [Online]. Available: <https://doi.org/10.1145/3706598.3713581>
- [6] H. Strobelt, A. Webson, V. Sanh, B. Hoover, J. Beyer, H. Pfister, and A. M. Rush, "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, Jan. 2023. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/TVCG.2022.3209479>
- [7] A. Mishra, B. Danzy, U. Soni, A. Arunkumar, J. Huang, B. C. Kwon, and C. Bryan, "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, Oct. 2025. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/TVCG.2025.3535332>
- [8] I. Arawjo, C. Swoopes, P. Vaithilingam, M. Wattenberg, and E. L. Glassman, "Chainforge: A visual toolkit for prompt engineering and llm hypothesis testing," in *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, ser. CHI '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: <https://doi.org/10.1145/3613904.3642016>
- [9] J. Wang, Y. Chen, M. Pan, C.-C. M. Yeh, and M. Das, "Illuminating llm coding agents: Visual analytics for deeper understanding and enhancement," 2025. [Online]. Available: <https://arxiv.org/abs/2508.12555>
- [10] V. Bourcier, A. Bergel, A. Etien, and S. Costiou, "Debugging activity blueprint," in *2024 IEEE Working Conference on Software Visualization (VISOFT)*, 2024, pp. 48–58.