TraceView: Interactive Visualization of Agentic Program Repair Trajectories

Amirali Sajadi*, Tu Nguyen*, Kimmie Huynh*, Esteban Parra†, Preetha Chatterjee*
*Drexel University, Philadelphia, PA, USA
{amirali.sajadi, jn866, kh3472, preetha.chatterjee}@drexel.edu
†Belmont University, Nashville, TN, USA
esteban.parrarodriguez@belmont.edu

***Abstract*—LLM-based automated program repair (APR) agents generate patches to fix software bugs with minimal human intervention. These agents often produce long trajectories of reasoning, tool use, and feedback to produce candidate patches. Final patch outcomes show whether a repair attempt succeeded or failed, but they do not show how the agent reached that outcome, or where the process became repetitive or misaligned with the task. This makes agentic repair failures difficult to diagnose, reproduce, and prevent. To help developers address these challenges, we present *TraceView*, an interactive tool for labeling and visualizing repair trajectories from APR systems. TraceView organizes raw and pre-labeled agentic runs with *Thought*, *Action*, and *Result* components to support semantic relation labeling and diagnosis, and renders the resulting trajectory as graph views. Furthermore, TraceView provides relation filters, patch outcome summaries, metrics, and node-level evidence panels to help users inspect how *reasoning, actions*, and *feedback* connect across the various steps of an agentic repair attempt. We evaluate TraceView with five researchers through a survey-based user study. Participants reported that TraceView made trajectories easier to scan and that its overview-to-detail workflow helped them better understand repair behavior. The TraceView source code is available at https://github.com/SOAR-Lab/agent-traj-visualization. A screencast of TraceView is available at https://youtu.be/9ZCh7Ifj2AQ.**

***Index Terms*—automated program repair, trajectory analysis, visualization, interface design**

I. Introduction

LLM-based software engineering agents are increasingly used for APR, issue resolution, and test-driven patch generation [1]–[4]. Systems such as AutoCodeRover and SWE-agent show that agents can search repositories, inspect code, edit files, and run tests while attempting to generate patches that could fix bugs in software systems [5], [6]. However, beyond the functionality of the final output, a passing patch does not explain which evidence the agent used, and a failing patch does not show whether the agent localized the bug, ignored feedback, repeated mistakes, or diverged from the task.

Trajectory analysis provides a way to inspect the repair process taken by an agentic APR system [7]. APR agent logs often consist of recurring *Thought*, *Action*, and *Result* (TAR) components [7]. A *thought* records the agent's reasoning or plan, an *action* records an operation such as searching files, inspecting code, editing code, or running tests, and a *result* records environmental feedback from the repository, tools, or tests. Beyond these components, recent work has shown the value of labeling the relations in agent trajectories [8]–[10]. These relation labels describe how one element connects to another, for example, whether a thought leads to an action, a result supports a later decision, an action repeats an earlier one, or feedback has no clear influence on the next step. Such labels are useful because they show where the agent's reasoning, actions, and feedback remain aligned, and where the repair process starts to loop, or diverge from task focus. They can also provide targeted course-correction guidance to nudge the agent back to a productive path [11].

Prior work also shows that producing these relation labels for agent trajectories is costly. For instance, Bouzenia and Pradel manually labeled 14K pairs of TAR components across 120 APR agent trajectories, with the initial annotation requiring months of work [7]. This effort illustrates both the value of relational trajectory analysis and the need for tool support. Without an interface for creating, filtering, and inspecting relation labels, researchers must move between raw logs and spreadsheets, making it difficult to reuse annotations or connect them back to the evidence in the original trace. This challenge is also consistent with broader principles for interactive analysis [12], [13]. Users should be able to start with an overview of the repair workflow, narrow the graph to relation types such as repetition, divergence, or no influence, and then inspect the exact thought, action, result, and source text behind each labeled connection.

Recent agent workflow analysis tools demonstrate the value of structuring long execution or reasoning traces before visualization. For example, Anteater visualizes program execution values in context [14]; AgentLens organizes agent events into hierarchical behaviors and sub behaviors [15]; Agent Trajectory Explorer supports visualization and feedback over agent trajectories [16]; ReTrace summarizes long reasoning traces through complementary phase based views [17]; InconLens focuses on repeated run diagnosis through semantically meaningful milestones [18]; and SeaView and related visual analytics work compare software engineering agent runs across models, tasks, and workflow settings [19], [20].

Together, these systems show the growing need for tools that structure, summarize, and support interactive analysis of long agent traces. However, no existing tool supports the fine-grained analysis of APR trajectories, where users label action categories and semantic relations among TAR elements,

then inspect how those labels reveal alignment, repetition, divergence, ignored feedback, and other process-level patterns within a repair attempt.

We present *TraceView*, an interactive tool for labeling and visualizing software engineering agent trajectories. TraceView represents each run as a directed graph over *Thought*, *Action*, and *Result* nodes. Users can upload raw agent trajectories, assign action categories, label semantic relations, export the labels into a reusable viewer format, and inspect the resulting graph. They can also analyze pre-labeled trajectories from the bundled dataset. TraceView provides two coordinated graph modes. The iteration view summarizes the run at the step level, while the detailed view expands each step into separate TAR nodes. Furthermore, TraceView provides relation filters, legends, relationship metrics, patch outcome summaries, and an inspector panel that helps users isolate productive flow, repeated behavior, negative relations, or no influence edges for each node.

We evaluated TraceView through a survey-based user study with five researchers, each with more than three years of experience. After using the tool, participants answered Likert-scale and open-ended questions about its perceived usefulness, clarity, evidence availability, and whether moving from trajectory overviews to detailed inspection helps them better understand APR agent behavior.

This paper makes the following contributions:

- We present *TraceView*, an interactive tool for labeling and visualizing APR agent trajectories as structured *Thought*, *Action*, and *Result* graphs.
- We provide a relation labeling and analysis workflow with semantic edges, relation filters, patch results, relationship metrics, and node-level evidence inspection.
- We report a preliminary user study showing that users found TraceView useful for scanning trajectories and understanding repair workflows, while identifying concrete improvements for graph readability and navigation.

II. TraceView

TraceView is a tool designed for researchers and developers who need to inspect and annotate APR agent behavior beyond final patch success or failure. TraceView makes the TAR structure of raw logs visible, highlights semantic relationships between trajectory elements, and lets users move between overview, filtering, and details on demand during trajectory analysis [13].

A. Workflow and Architecture

TraceView is implemented as a Streamlit application with three user-facing sections. The *Labeling* section ingests uploaded trajectories, presents action labeling and relation labeling tasks, and exports labels into the same CSV-based structure used by the viewer. The *Overview* section lists available bundled and user-exported runs. The *Analysis* section renders the selected run as an interactive graph and provides legends, relation metrics, patch result summaries, graph controls, and an inspector for the selected node.

TABLE I
Action categories and relation labels supported by TraceView.

Label type	**Labels**
Action categories	*Explore*, *Locate*, *Search*, *Reproduce*, *Generate fix*, *Run tests*, *Refactor*, *Explain*
Relation labels	*Alignment*, *Follow-up*, *Refinement*, *Redundancy*, *Repetition*, *Divergence*, *Contradiction*, *Informative*, *Triggering*, *No influence*, *Misalignment*, *Misinterpretation*

Internally, TraceView keeps raw trajectory content, user labels, and graph state separate. Action labels are stored at the step level, while semantic relation labels are stored at the edge level, allowing labeled runs to be reopened, analyzed, and reused without reprocessing the original trace.

B. Input Artifacts and Label Vocabulary

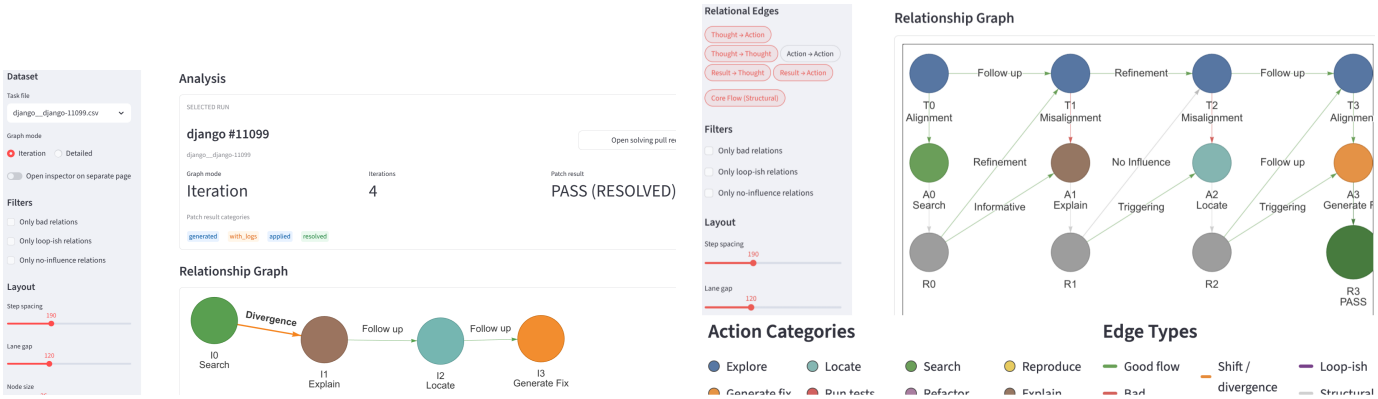
TraceView accepts both raw and pre-labeled inputs. Raw inputs include SWE agent style JSON trajectories [6], JSONL command traces, plain text or log files with *Thought*, *Action*, and *Result* sections, and zip archives containing multiple supported trajectory files. For bundled data, TraceView loads AutoCodeRover style APR trajectories [5], reconstructed text logs, action category CSV files, relation label CSV files, and patch outcome metadata. The action and relation labels in the bundled dataset were produced in prior work and are consumed by TraceView for visualization and analysis; TraceView does not generate these labels automatically.

Supported inputs are normalized into ordered *Thought*, *Action*, and *Result* steps before labeling or analysis. These normalized steps give both workflows the same underlying representation, whether the run comes from an uploaded raw trace or a bundled pre-labeled trajectory.

Following prior work on TAR trajectory analysis of software engineering agents [7], TraceView supports the action categories and semantic relation labels shown in Table I. Action categories describe what the agent does at each step, while relation labels describe how one TAR component relates to another. Relation labels define which types of TAR components are being connected, such as a thought to an action within the same step or a result to a consecutive thought. By attaching a semantic label to each source–target TAR relation, TraceView lets users characterize how the repair process evolves. For example, if a test result reports a failing assertion and the next thought uses that failure to plan a fix, the Result→Thought edge can be labeled *Informative*; if the next thought instead draws an incorrect conclusion from the same feedback, that edge can be labeled *Misinterpretation*.

C. Labeling Workflow

Figure 2 shows the labeling workspace. The labeling workflow follows the dependency structure of the data. Users label actions first because action categories are needed to gather iteration-level context. They then label semantic relations between source and target components. Before relation labeling, TraceView creates relation candidates from fixed source–target



(a) Iteration view. (b) Detailed TAR view.

Fig. 1. TraceView provides two graph views for APR trajectory analysis. The iteration view summarizes each step by action category, while the detailed view expands each step into *Thought*, *Action*, and *Result* nodes with semantic relation edges.

templates. A candidate at step i may connect nodes within the same step or across adjacent steps, depending on the relation template.

For each relation candidate, the interface shows the relation to be labeled, the relevant steps, the evidence text from both TAR components, and the available labels. For example, when labeling whether a test result influences the agent's next thought, the user can inspect the result text and the following thought before selecting a label such as *Informative*, *No influence*, or *Misinterpretation*. Longer evidence can be opened in a popover, reducing table clutter while preserving access to the original text.

Completed labels can be exported as JSON or viewer-compatible CSV files for graph analysis. When a user sends a labeled trajectory to the viewer, TraceView writes the action categories, relation files, reconstructed TAR log, and metadata for the Overview and Analysis sections.

D. Graph Views

Figure 1 shows the two graph views supported by TraceView. The iteration view provides a compact entry point for scanning the run at the step level, while the detailed view expands each step into separate *Thought*, *Action*, and *Result* nodes.

TraceView transforms raw agent logs into a graph-based intermediate representation before rendering [14]–[16]. In detailed mode, each source step i is expanded into three nodes, T_i for the thought, A_i for the action, and R_i for the result. Thought and result nodes use fixed role colors, while action nodes are colored by their action category. For bundled trajectories, patch outcome metadata is reduced to a primary status such as *RESOLVED*, *APPLIED*, *GENERATED*, or *UNKNOWN*. This status determines the terminal result node annotation as *PASS*, *FAIL*, or *UNSCORED*.

The semantic edges are constructed from five relation families. These families connect *Thought*→*Action* nodes within the same step, *Thought*→*Thought* nodes across adjacent steps, *Action*→*Action* nodes across adjacent steps, *Result*→*Thought* nodes across adjacent steps, and *Result*→*Action* nodes across adjacent steps. TraceView can also render structural flow edges to preserve the local TAR sequence and chronological progression when semantic filters are disabled.

In iteration mode, TraceView renders one node per source iteration, labeled by iteration number and action category, with optional context derived from the reconstructed log. Action-to-action relations are displayed between iteration nodes, and structural chronological edges are added when no relation filter is active. Users can switch to detailed mode when they need the full TAR decomposition.

E. Filtering, Inspection, and Metrics

TraceView maps relation labels into visual families. Productive continuation labels such as *Alignment*, *Follow-up*, *Refinement*, *Informative*, and *Triggering* are rendered as flow edges. *Divergence* is shown as a direction shift, *Redundancy* and *Repetition* as iterative loops, *Misalignment*, *Misinterpretation*, and *Contradiction* as negative relations, and *No influence* is treated as a neutral relation. This grouping allows users to filter the graph by relation types, making it easier to isolate productive or unproductive flows, consistent with interactive behavioral analysis tools that help users isolate meaningful subsets rather than inspect a single undifferentiated view [21].

Selecting a node opens an inspector with the underlying thought, action, or result text and the labeled relations connected to that node. This interaction follows details on demand principles and aligns with explanatory and interactive debugging work that emphasizes user guided diagnosis over passive display [13], [22]–[24]. TraceView also reports aggregate relation counts by label and edge family, giving users a compact summary of the dominant patterns in the selected trajectory.

III. Preliminary Evaluation

We conducted a preliminary user study to assess whether TraceView is understandable and useful for researchers or developers inspecting APR agent trajectories. Because the goal of the study was design feedback rather than statistical validation, we focused on perceived clarity, evidence availability, workflow usefulness, and open-ended comments about confusing or missing interface behavior.

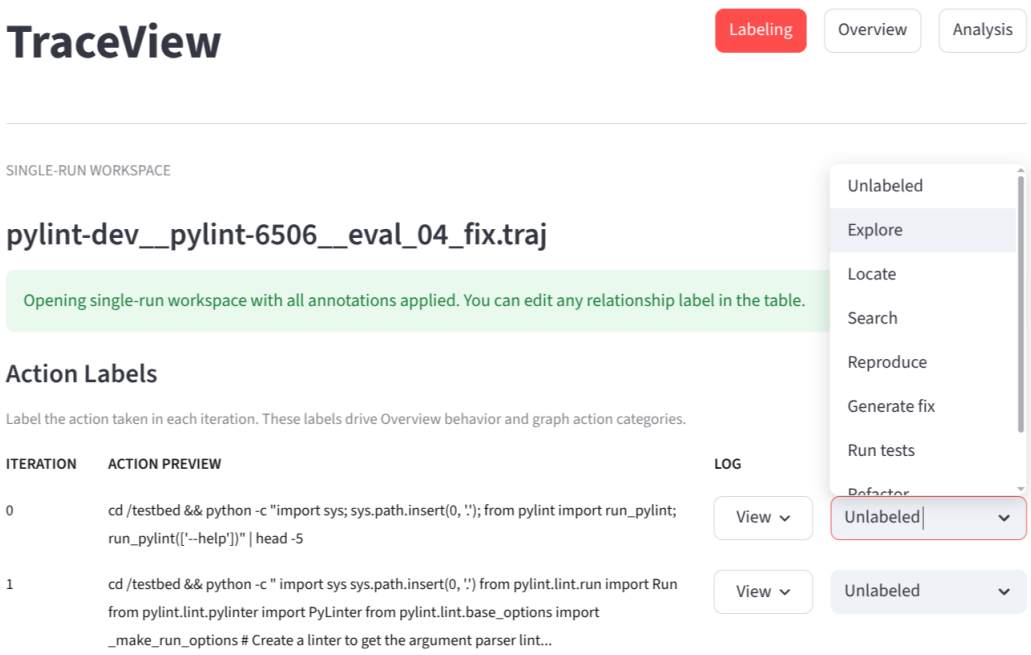


Fig. 2. TraceView's labeling workspace lets users assign action categories and inspect source evidence before labeling semantic relations.

A. Participants and Procedure

We recruited five computer science researchers to participate in the user study. Participants first used TraceView with an APR trajectory sample and followed the workflow from labeling to graph analysis. The task asked them to inspect the repair process, use the available views, and identify confusing or problematic elements of the UI when possible. The participants completed a survey that included both five-point Likert items and open-ended questions.

The survey covered three parts of the interface: 1) the *labeling workflow*, including whether the main sections and accepted input formats were clear, whether action previews were long enough for labeling decisions, whether longer-log popovers were useful, and whether the labeling table was dense without being overwhelming; 2) the *relationship evidence and graph interpretation*, including whether row-level popovers provided enough source and target evidence and whether the graph could be interpreted without guessing; and 3) the *analysis workflow*, asking whether TraceView was more efficient than raw logs, whether the graph made trajectories easier to scan, and whether the overview/detail hierarchy supported analysis and orientation, with each part also including open-ended questions about confusing renderings, missing information, or interface suggestions. Participants reported spending approximately 20 to 90 minutes completing the study tasks and survey, including time spent becoming familiar with the task and interface. The survey form and study responses are included in our replication package.

B. Survey Results

Table II summarizes the survey responses across the three parts of the study. Overall, participants rated TraceView positively across the labeling, relationship-evidence, and analysis workflows. For the labeling workflow, participants rated the clarity of the main sections (median = 4) and rated the accepted input formats and longer log popovers highly (median = 5). For relationship evidence and graph interpretation, row-level evidence popovers received a median of 5, while graph interpretability received a median of 4. For the analysis workflow, participants rated TraceView as more efficient than raw logs, the graph easier to scan, and the overview/detail view useful for analysis, all with medians of 5.

TABLE II
SELECTED LIKERT RESULTS FROM THE FORMATIVE EVALUATION ($n = 5$).

Survey item	Min	Median	Max
Labeling workflow			
Main sections are easy to understand	3	4	5
Accepted input formats are clear	4	5	5
Action previews are long enough for labeling	1	4	5
View popover is useful for longer logs	4	5	5
Relationship evidence and graph interpretation			
Row-level popovers provide enough evidence	3	5	5
Graph is interpretable without guessing	1	4	5
Analysis workflow			
TraceView is more efficient than raw logs	4	5	5
Graph makes trajectory easier to scan	4	5	5
Overview/detail view is useful for analysis	3	5	5

The lower minimum scores for action-preview length and graph interpretability show that some participants needed more concise and informative cues. Open-ended comments suggested that short previews sometimes lacked enough context, while longer previews could be difficult to scan.

C. Feedback and Suggestions for Future Improvement

Open-ended feedback identified several concrete strengths and limitations. Participants found the graph useful for understanding the repair process, but wanted node labels to carry short summaries rather than only identifiers e.g., $T0$

or $A0$. Some comments requested hover previews or concise summaries for TAR nodes. Participants also noted cases where diagonal edges or source–target relations were difficult to interpret without additional explanation, especially when source previews were empty or when relation labels such as *Misalignment* were not immediately obvious from the graph.

These findings are useful as they point to changes that can be made without altering the core data model. Addition of short node summaries and hover previews would make the current graph easier to interpret while preserving the TAR representation and semantic relation workflow.

Since this was a preliminary study with five participants and one APR trajectory sample, the results reflect perceived usefulness rather than numeric improvements in efficiency, accuracy, or diagnostic performance.

IV. Conclusions and Future Work

TraceView can help researchers and developers inspect APR agent behavior as a process rather than as only a final patch outcome. It parses trajectories into TAR components, supports action and relation labeling, and renders the resulting trace as an interactive graph with filters, metrics, patch-status summaries, and node-level evidence. Together, these features aim to make it easier to study where an agent follows useful feedback, repeats the same actions, diverges from the task, or misinterprets the repair context.

Our preliminary user study suggests that the participants found TraceView to improve trajectory scanning compared with raw logs and that the overview-to-detail workflow is useful for understanding repair runs. The study also identified clear next steps, including the addition of shorter node summaries and hover previews. In our future work, we plan to incorporate these improvements and conduct a larger user study with longer and more diverse tool trajectories to measure the tool's usefulness for identifying common success and failure patterns in agentic APR workflows.

References

[1] R. Ehsani, S. Pathak, E. Parra, S. Haiduc, and P. Chatterjee, "What characteristics make chatgpt effective for software issue resolution? an empirical study of task, project, and conversational signals in github issues," *Empirical software engineering*, 2026.

[2] R. Ehsani, E. Parra, S. Haiduc, and P. Chatterjee, "Hierarchical knowledge injection for improving llm-based program repair," in *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE'25)*, 2025.

[3] C. S. Xia and L. Zhang, "Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 819–831. [Online]. Available: https://doi.org/10.1145/3650212.3680323

[4] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. IEEE Press, 2023, p. 1469–1481. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00128

[5] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1592–1604.

[6] 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," *Advances in Neural Information Processing Systems*, vol. 37, pp. 50 528–50 652, 2024.

[7] I. Bouzenia and M. Pradel, "Understanding software engineering agents: A study of thought-action-result trajectories," 2025. [Online]. Available: https://arxiv.org/abs/2506.18824

[8] S. Liu, Y. Chen, R. Krishna, S. Sinha, J. Ganhotra, and R. Jabbarvand, "Process-centric analysis of agentic software systems," *Proceedings of the ACM on Programming Languages*, vol. 10, no. OOPSLA1, p. 1961–1988, Apr. 2026.

[9] M. Abdollahi, K. R. Tasnia, S. K. Saha, J. Yang, S. Wang, and H. Hemmati, "Demystifying errors in llm reasoning traces: An empirical study of code execution simulation," 2025. [Online]. Available: https://arxiv.org/abs/2512.00215

[10] I. Bouzenia, P. Devanbu, and M. Pradel, "Repairagent: An autonomous, llm-based agent for program repair," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 2025, pp. 2188–2200.

[11] R. Nanda, C. Maddila, S. Jha, E. M. Khan, M. Paltenghi, and S. Chandra, "Wink: Recovering from misbehaviors in coding agents," 2026. [Online]. Available: https://arxiv.org/abs/2602.17037

[12] T. Munzner, "A nested model for visualization design and validation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 921–928, 2009.

[13] B. Shneiderman, "The eyes have it: a task by data type taxonomy for information visualizations," in *Proceedings 1996 IEEE Symposium on Visual Languages*, 1996, pp. 336–343.

[14] R. Faust, K. Isaacs, W. Z. Bernstein, M. Sharp, and C. Scheidegger, "Anteater: Interactive visualization of program execution values in context," 2024. [Online]. Available: https://arxiv.org/abs/1907.02872

[15] J. Lu, B. Pan, J. Chen, Y. Feng, J. Hu, Y. Peng, and W. Chen, "Agentlens: Visual analysis for agent behaviors in llm-based autonomous systems," 2024. [Online]. Available: https://arxiv.org/abs/2402.08995

[16] M. Desmond, J. Y. Lee, I. Ibrahim, J. M. Johnson, A. Sil, J. MacNair, and R. Puri, "Agent trajectory explorer: visualizing and providing feedback on agent trajectories," in *Proceedings of the Thirty-Ninth AAAI Conference on Artificial Intelligence and Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence and Fifteenth Symposium on Educational Advances in Artificial Intelligence*, ser. AAAI'25/IAAI'25/EAAI'25. AAAI Press, 2025.

[17] L. Felder, J. Miller, M. Wallinger, S. Kobourov, and C. Chen, "Retrace: Interactive visualizations for reasoning traces of large reasoning models," 2025. [Online]. Available: https://arxiv.org/abs/2511.11187

[18] S. Yan, X. Wen, S. Ruan, Y. Zhang, J. Mi, Y. Sun, H. Qu, and R. Sheng, "Inconlens: Interactive visual diagnosis of behavioral inconsistencies in llm-based agentic systems," 2026. [Online]. Available: https://arxiv.org/abs/2603.28106

[19] T. Bula, S. Pujar, L. Buratti, M. Bornea, and A. Sil, "Seaview: Software engineering agent visual interface for enhanced workflow," 2025. [Online]. Available: https://arxiv.org/abs/2504.08696

[20] 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

[21] A. A. Cabrera, E. Fu, D. Bertucci, K. Holstein, A. Talwalkar, J. I. Hong, and A. Perer, "Zeno: An interactive framework for behavioral evaluation of machine learning," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, ser. CHI '23. ACM, 2023, pp. 1–14. [Online]. Available: https://doi.org/10.1145/3544548.3581268

[22] T. Kulesza, M. Burnett, W.-K. Wong, and S. Stumpf, "Principles of explanatory debugging to personalize interactive machine learning," in *Proceedings of the 20th International Conference on Intelligent User Interfaces*, ser. IUI '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 126–137. [Online]. Available: https://doi.org/10.1145/2678025.2701399

[23] 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. ACM, Apr. 2025, p. 1–15. [Online]. Available: http://dx.doi.org/10.1145/3706598.3713581

[24] R. Hutter and M. Pradel, "Agentstepper: Interactive debugging of software development agents," 2026. [Online]. Available: https://arxiv.org/abs/2602.06593