Copy-on-Write Scoring: Application-Specific Agent Evaluations

Joanna Roy [1] **Sven Hölzel** [1]

Abstract

Trustworthy deployment of LLM-based agents in software systems requires evaluating how they perform on application-specific workflows, with enough granularity to localize where they succeed and fail. Yet existing agent evaluation mechanisms are limited: benchmarks have low construct validity for application-specific workflows and environments, and replica evaluation environments are expensive and prone to drift. We propose **Copy-on-Write (CoW) Scoring**[1], a framework that evaluates agent operations directly within application environments using a PostgreSQL-level Copy-on-Write mechanism to isolate agent writes. CoW Scoring produces session- and operation-level scores that highlight where agents' database write operations succeed and fail in a given application environment, enabling inexpensive evaluation and iteration on agent harnesses and tool surfaces. We demonstrate the framework on Plane, an open-source project-management platform, where analysis surfaced specific issues in the tool surface, and corresponding fixes produced measurable improvements on affected models.

1. Introduction

Large Language Model (LLM)-based agents are increasingly deployed in production software systems, from enterprise tooling (e.g., ServiceNow, Salesforce, Notion) to MCP integrations across a wide range of consumer applications.[2] Many of these applications are relied upon by a wide range of organizations and end-users, underpinning core business operations across large segments of the economy. Agent unreliability and errors — such as accidental deletion or overwriting of data, misuse of tooling, or incorrect changes to critical records — could have substantial ripple effects, yet are difficult to identify and prevent in practice, particularly when agent writes are interleaved with the prior database state.

Evaluating agents in software contexts remains an underdeveloped area. Benchmarks are the most common means of assessing performance, but strong benchmark results do not guarantee strong performance in real systems or on real-world tasks (Mohammadi et al., 2025; Raji et al., 2021; Liao & Xiao, 2025; Deng et al., 2023). Existing benchmarks tend to have low *construct validity* (Bean et al., 2025) for the environments in which software agents are deployed.

Some benchmarks have attempted to close this gap by situating agents in replicas of common applications (Drouin et al., 2024; Zhou et al., 2024b; Xu et al., 2025) or in simulated domains with API access (Yao et al., 2024). While these partly address the construct validity gap, two main limitations remain: (a) replicas and simulations are time- and resource-expensive to produce, and can quickly drift from the live state of the application; and (b) evaluating performance on a *specific* software is still not necessarily predictive of performance in others — for example, one with different tooling, system prompts, and workflows. Since no single benchmark can predict agent performance for arbitrary applications, teams need complementary evaluation methods that work directly in their application environment, on representative workflows.

We propose ***Copy-on-Write (CoW) Scoring***, a framework to develop and safely conduct use-case-specific agent evaluations in software applications. The CoW mechanism allows agents to operate directly within the application environment, avoiding the cost and drift of replicas while keeping sessions isolated and setup easy to reuse across runs. This work assumes a PostgreSQL database, although the CoW pattern could similarly be extended to other data stores, which would enable scoring a broader range of agent operations. The scoring framework provides session- and operation-level scores that surface where agent writes succeed and fail when interacting with a given tool surface.

We demonstrate an evaluation loop on Plane,[3] an open-source project-management platform, where scoring surfaces multiple failure modes specific to the deployed agents

[1]trail-ml, Munich, Germany. Correspondence to: Joanna Roy <joanna@trail-ml.com>, Sven Hölzel <sven@trail-ml.com>.

Published at the Second Workshop on Agents in the Wild: Safety, Security, and Beyond (AIWILD) at ICML 2026. Copyright 2026 by the author(s).

[1]Python library: `agent-cow`

[2]Gartner (2025); BCG (2025); Deloitte (2026)

[3]plane.so

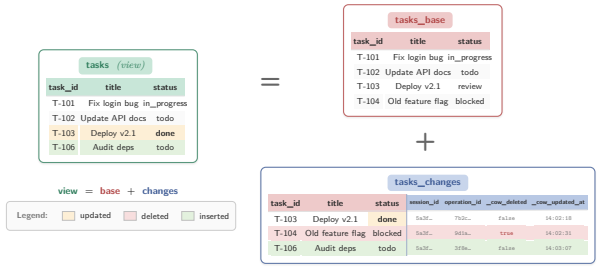

Figure 1. **Copy-on-Write (CoW) mechanism in agent evaluation.** The agent reads and writes through a **view**, which merges the **base table** (production data) with the **changes table** (per-session using the operation/session metadata). Agent writes are intercepted and written to the changes table, leaving base data untouched.

and tool surface, and a corresponding fix produces measurable improvement on the affected models. The same score-diagnose-fix procedure applies to any application using CoW Scoring, enabling low-cost iteration on the model, prompt, retrieval, or system prompt in a given deployment.

2. Copy-on-Write (CoW) Scoring

This section describes the CoW *mechanism*, which isolates agent database changes, and the CoW *Scoring framework*, which compares a ground-truth (GT) execution of a given workflow with a corresponding agent execution at the session and operation level. Both components are implemented in the open-source `agent-cow` library.

2.1. CoW Mechanism

Copy-on-Write (CoW) is a resource-management technique that lets processes share underlying data without creating copies upfront. Rather, reads go to the shared data and writes trigger creation of a copy such that only the writing process observes the change (Silberschatz et al., 2018). Applied to agent operations on databases, CoW prevents agents from writing directly to the underlying production data.

To enable CoW on a database, each original table is split into a **base** table (contains the underlying production data), **changes** table (contains agent writes), and corresponding **view** (defined by a query merging base and changes table data). All application operations are sent to the view, since it takes the name of the original table. View triggers redirect CoW write operations to the changes table, with associated CoW metadata columns (`session_id`, `operation_id`, `_cow_updated_at`, `_cow_deleted`) appended. Actions from a given session can then be queried using the respective `session_id`. The components are visualized in Figure 1, and more details are described in Appendix A.

The `agent-cow` library handles database setup — creating base tables, changes tables, views, and deploying triggers (Algorithm 1, Appendix A), and the application is responsible for passing `session_id` and `operation_id` with each agent operation. For Plane, CoW integration required ∼250 lines of code for CoW functionality, and an additional ∼540 lines specific to our recording infrastructure and portable to a testing harness (Appendix C.1). We have separately integrated CoW Scoring into a closed-source production application, with comparable effort.

2.2. Scoring

Scoring takes two CoW sessions as input: a ground-truth (GT) session and an agent session. The GT session is recorded by a human performing the ideal sequence of actions for a given workflow with CoW enabled. The human also writes a prompt describing the workflow such that they would expect an agent to be able to reproduce their actions. The agent is then given this prompt and executes its own sequence of actions with CoW enabled.

CoW Scoring happens along two dimensions and at two levels of granularity. The two dimensions are structural (which tables, rows, columns, and relationships were written) and content (what values were written). The two levels of granularity are session-level (evaluates the session as a whole) and operation-level (evaluates each operation). Users can supply their own comparators and overall scoring functions; Appendix B.1 describes the library architecture.

2.2.1. Session-level comparison

For session-level scoring, we keep the most recent written row state for each primary key. Changed rows are classified as: *matched* (N_{matched} — rows present in both GT and agent sessions, under UUID mapping), *missing* (N_{missing} — GT rows the agent never produced), or *extra* (N_{extra} — agent rows with no GT counterpart).

Session-level structural score, s^{struct}: The fraction of *matched* rows relative to the total number of rows changed.

$$s^{\text{struct}} = \frac{N_{\text{matched}}}{N_{\text{matched}} + N_{\text{missing}} + N_{\text{extra}}} \tag{1}$$

Session-level content score, s^{content}: The mean per-field similarity over all matched rows.

$$s^{\text{content}} = \frac{1}{N_{\text{matched}}} \sum_{(g,a)\in\text{matched}} \text{sim}(g, a) \tag{2}$$

where $\text{sim}(g, a) \in [0, 1]$ is the similarity calculation per row between g (GT) and a (agent). Each column has a default similarity comparator according to its data type, but custom comparators can be provided as described in Appendix B.1.1.

Notably, CoW Scoring compares final database states rather than action sequences, so the session-level scores are invariant to the path the agent took.

2.2.2. Operation-level comparison

While the session-level scores evaluate whether the agent reached the desired GT world state, the operation-level scores quantify how useful each operation was in getting there. We first sort operations in topological order, then calculate the following for each operation.

Op-level structural utility, $u^{\text{struct}}(o_i)$: The change in the session-level structural score before and after o_i is applied.

$$u^{\text{struct}}(o_i) = s^{\text{struct}}(G_{\leq i}) - s^{\text{struct}}(G_{<i}) \tag{3}$$

Where $G_{<i}$ denotes the world state prior to applying o_i, and $G_{\leq i}$ denotes the state after o_i is applied.

Op-level content utility $u^{\text{content}}(o_i)$: The mean per-field content similarity over operation rows.

$$u^{\text{content}}(o_i) = \frac{1}{|M_i|} \sum_{(g,a)\in M_i} \text{sim}(g, a) \tag{4}$$

where M_i is the subset of matched rows whose agent-session write originated in o_i.

3. Methods

Because the framework targets application-specific evaluation rather than a universal benchmark, our empirical validation will not compare against an external ground truth. The preliminary study below aims to show that (a) the CoW mechanism and scoring framework can be implemented in a PostgreSQL application, and (b) resulting scores surface useful information about agent behaviour, which can inform improvements to the tool surface in subsequent iterations.

3.1. Preliminary Study: Application in Plane

We apply CoW Scoring to **Plane**, an open-source project-management platform, as a concrete demonstration of the evaluation workflow. To produce a diverse set of ground-truth (GT) sessions without spending excessive author effort, Claude Opus 4.6 with access to the Plane codebase was prompted to execute realistic workflows against the seeded data, and each workflow manually reviewed and validated by the authors (details in Appendix C). We developed a testing harness, which embeds Plane's OpenAPI specification into a vector store, and exposes them to the agent through two tools: a `discover` tool for querying relevant endpoints, and an `execute` tool for making the corresponding requests.

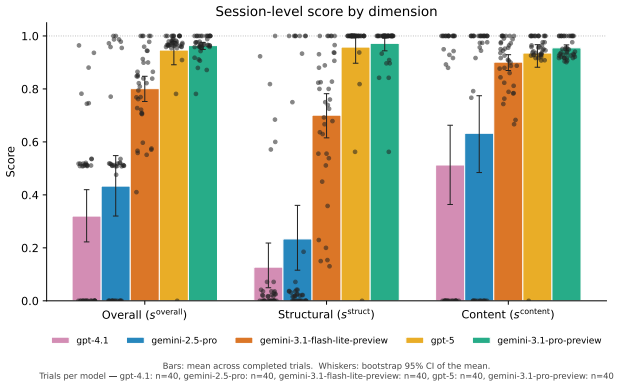

Figure 2. **Score breakdown** by dimension per model, for runs 1 and 2.

Since CoW sessions are isolated, each GT and agent session started with the same application state. For each GT session, an agent was given the associated prompt and access to the Plane API via the harness `execute` and `discover` tools, with a 50-operation limit per session. Five language models were evaluated twice per prompt: GPT-5, GPT-4.1, Gemini-3.1-Pro, Gemini-3.1-Flash-Lite, and Gemini-2.5-Pro. In total, $5 \times 2 \times 20 = 200$ agent sessions were scored against their corresponding GT sessions using CoW Scoring. A third trial per model was run after updating the tool surface (Section 4.3), for a total of 300 sessions.

4. Preliminary Results

4.1. Initial scores across models

Figure 2 shows CoW scores along each dimension, averaged between the two runs per (model, workflow) pair, with per-model summaries in Appendix D (Tables 5 and 6). These are early results from a single application and 20 workflows, with two trials per (model, workflow) pair, so broader claims would benefit from more samples.

Two initial observations serve as a sanity check on the scoring functions: within each model family, the ordering matches public benchmarks (Gemini-3.1-Pro > Gemini-3.1-Flash-Lite, GPT-5 > GPT-4.1), and per-model overall scores differ by at most 0.03 between trials, suggesting the means are representative rather than single-run artefacts.

4.2. Diagnosing failure modes

Vocabulary mismatch. Plane uses "work item" to describe tasks within projects, while the session prompts use the term "issue", which is also commonly used to describe these items (the database tables still use "issues", suggesting an artefact of a previous naming convention), so an agent that searched for "issue" would not immediately find relevant operations. **Gemini-2.5-Pro** repeatedly gave up rather

Table 1. **Run 3 vs. runs 1 & 2 mean overall score.** Run 3 used an updated tool surface based off the analysis of runs 1 and 2.

Model	Avg(1,2)	Run 3	n	Δ
`GPT-4.1`	0.32	0.79	20	**+0.47**
`Gemini 2.5 Pro`	0.43	0.98	20	**+0.54**
`Gemini 3.1 Flash-Lite`	0.80	0.83	20	+0.03
`GPT-5`	0.95	0.97	20	+0.02
`Gemini 3.1 Pro`	0.96	0.98	20	+0.01

than rephrasing or reading endpoint descriptions closely enough to recognise the equivalence (Appendix E, Figure 6). **GPT-4.1** executed several `discover` calls before finding `search_work_items`. It executed one search using an incorrect search string, then wrongly concluded no relevant issues existed when several did (Appendix E, Figure 7). It took **Gemini-3.1-Flash-Lite** fifteen `discover` calls to find the correct operation, causing it to ultimately exceed the 50-operation session limit (Appendix E, Figure 8).

Importantly, this vocabulary mismatch challenge is well-known in information-retrieval (Furnas et al., 1987) and not specific to Plane: any application is likely to have multiple terms in circulation for the same concept.

Hallucination and extra writes. **GPT-4.1** occasionally hallucinated incorrect parameters and operation names, by guessing operations before first calling the `discover` tool to ensure accuracy (Appendix E, Figures 9, 10, 11). In these sessions, the hallucinations were benign: the invalid operations failed and left application state unchanged, but the same failure mode could be damaging if the agent invoked a valid operation with an incorrect name or parameters or unintended side effects. **Gemini-3.1-Flash-Lite** hallucinated extra writes, producing label and assignee updates outside the scope of the prompted task (e.g., 47 extra rows in Campaign Push, 44 in Q3 Feature Prep). This is especially easy to miss outside a CoW session: an agent can satisfy the prompt while modifying unrelated application state in incorrect or harmful ways.

4.3. Improving the tool surface

We updated the tool surface to address the findings, in particular to ensure "issues" was also used to describe operations related to "work items". We ran one additional trial per model; score deltas are reported in Table 1. Models most affected by the vocabulary mismatch saw the largest gains: Gemini-2.5-Pro improved by 54%, and GPT-4.1 by 47%. While Gemini-3.1-Flash-Lite improved by just 3%, it had no failed sessions in run 3 compared to at least one per iteration previously.

This loop — CoW Scoring, localising the points of failure, and iterating or flagging known weak areas to users — can be applied to analyze any tool surface using CoW Scoring.

5. Limitations

GT sessions are created manually. This step takes time, but we believe the effort is comparable to building any application-specific dataset — some investment is needed to address the construct-validity gap that motivates this work.

Evaluation assumes prompts that fully specify the intended outcome. Real users often send underspecified prompts where multiple final states are reasonable, which CoW Scoring would conflate with failure — custom comparators, vocabulary variation, and analyzing structural and content scores separately could partially mitigate this.

Scoring covers only write operations. Agents also issue reads, and the volume and pattern of those reads carries diagnostic signal — failing models in our runs often issued many redundant `discover` calls without affecting structural or content scores, but these calls in fact surfaced the work-item/issue mismatch. Incorporating read-operation signals could help diagnose these failure modes directly.

6. Future Work

Input to optimization loops. The low computational cost of CoW Scoring suggests it could also serve as a signal in optimization loops — for example, to iteratively improve tool interfaces via GEPA (Agrawal et al., 2026), or as a dense reward for model fine-tuning via GRPO-style approaches (Shao et al., 2024). Operation-level scores are particularly well-suited to this: they assign credit to partial progress towards the final world state, so optimization need not depend on the discovery of complete, successful trajectories.

Automated GT generation and coverage. GT sessions are manually generated, which is inherently limited in scale and may not give uniform coverage of the API surface. Automated discovery methods — e.g. MCTS-style (Kocsis & Szepesvári, 2006; Zhou et al., 2024a) exploration with LLM-guided candidate selection — could generate candidate sessions, with human review reserved for auditing.

Impact Statement

CoW Scoring aims to support safer agent deployments by making evaluations possible directly in application environments. The framework risks giving false confidence if GT sessions are unrepresentative of real usage: strong performance on a narrow set of evaluated workflows does not guarantee correct behaviour on others. Though not the focus of this paper, the CoW mechanism can also be used as a general runtime safeguard against unintended agent writes.

References

Agrawal, L. A., Tan, S., Soylu, D., Ziems, N., Khare, R., Opsahl-Ong, K., Singhvi, A., Shandilya, H., Ryan, M. J., Jiang, M., Potts, C., Sen, K., Dimakis, A. G., Stoica, I., Klein, D., Zaharia, M., and Khattab, O. GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning, February 2026.

Bean, A. M., Kearns, R. O., Romanou, A., Hafner, F. S., Mayne, H., Batzner, J., Foroutan, N., Schmitz, C., Korgul, K., Batra, H., Deb, O., Beharry, E., Emde, C., Foster, T., Gausen, A., Grandury, M., Han, S., Hofmann, V., Ibrahim, L., Kim, H., Kirk, H. R., Lin, F., Liu, G. K.-M., Luettgau, L., Magomere, J., Rystrøm, J., Sotnikova, A., Yang, Y., Zhao, Y., Bibi, A., Bosselut, A., Clark, R., Cohan, A., Foerster, J., Gal, Y., Hale, S. A., Raji, I. D., Summerfield, C., Torr, P. H. S., Ududec, C., Rocher, L., and Mahdi, A. Measuring what Matters: Construct Validity in Large Language Model Benchmarks, November 2025.

Deng, C., Zhao, Y., Tang, X., Gerstein, M., and Cohan, A. Investigating Data Contamination in Modern Benchmarks for Large Language Models. https://arxiv.org/abs/2311.09783v2, November 2023.

Drouin, A., Gasse, M., Caccia, M., Laradji, I. H., Verme, M. D., Marty, T., Vazquez, D., Chapados, N., and Lacoste, A. WorkArena: How Capable are Web Agents at Solving Common Knowledge Work Tasks? In *Proceedings of the 41st International Conference on Machine Learning*, pp. 11642–11662. PMLR, July 2024.

Furnas, G. W., Landauer, T. K., Gomez, L. M., and Dumais, S. T. The vocabulary problem in human-system communication. *Commun. ACM*, 30(11):964–971, November 1987. ISSN 0001-0782. doi: 10.1145/32206.32212.

Garcia-Molina, H., Ullman, J. D., and Widom, J. *Database Systems: The Complete Book (2nd Edition)*. Prentice Hall Press, One Lake Street Upper Saddle River, NJ, United States, 2 edition, June 2008. ISBN 978-0-13-187325-4.

Kocsis, L. and Szepesvári, C. Bandit Based Monte-Carlo Planning. In Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J. M., Mattern, F., Mitchell, J. C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M. Y., Weikum, G., Fürnkranz, J., Scheffer, T., and Spiliopoulou, M. (eds.), *Machine Learning: ECML 2006*, volume 4212, pp. 282–293. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-45375-8 978-3-540-46056-5. doi: 10.1007/11871842_29.

Liao, Q. V. and Xiao, Z. Rethinking Model Evaluation as Narrowing the Socio-Technical Gap, January 2025.

Mohammadi, M., Li, Y., Lo, J., and Yip, W. Evaluation and Benchmarking of LLM Agents: A Survey. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, pp. 6129–6139, August 2025. doi: 10.1145/3711896.3736570.

Raji, D., Denton, E., Bender, E. M., Hanna, A., and Paullada, A. AI and the Everything in the Whole Wide World Benchmark. *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, 1, December 2021.

Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Bi, X., Zhang, H., Zhang, M., Li, Y. K., Wu, Y., and Guo, D. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models, April 2024.

Silberschatz, A., Galvin, P. B., and Gagne, G. *Operating System Concepts*. Wiley, 10th edition, 2018. ISBN 978-1-119-32091-3.

Xu, F. F., Song, Y., Li, B., Tang, Y., Jain, K., Bao, M., Wang, Z. Z., Zhou, X., Guo, Z., Cao, M., Yang, M., Lu, H. Y., Martin, A., Su, Z., Maben, L., Mehta, R., Chi, W., Jang, L., Xie, Y., Zhou, S., and Neubig, G. TheAgentCompany: Benchmarking LLM Agents on Consequential Real World Tasks. In *Advances in Neural Information Processing Systems*, 2025. doi: 10.48550/ARXIV.2412.14161.

Yao, S., Shinn, N., Razavi, P., and Narasimhan, K. τ-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains, June 2024.

Zhou, A., Yan, K., Shlapentokh-Rothman, M., Wang, H., and Wang, Y.-X. Language Agent Tree Search Unifies Reasoning Acting and Planning in Language Models, June 2024a.

Zhou, S., Xu, F. F., Zhu, H., Zhou, X., Lo, R., Sridhar, A., Cheng, X., Ou, T., Bisk, Y., Fried, D., Alon, U., and Neubig, G. WebArena: A Realistic Web Environment for Building Autonomous Agents, April 2024b.

A. Copy-on-Write (CoW) Mechanism

This appendix expands on the CoW mechanism summarised in Section 2.1, detailing the four database-level components that together intercept and isolate agent writes without modifying production data:

1. **Base tables**: These tables contain the underlying (base) production data, and follow the schema definitions specified in the application.
2. **Changes tables**: Each base table has a corresponding changes table. Changes tables are structurally identical to base tables (with four additional columns: `session_id`, `operation_id`, `_cow_updated_at`, `_cow_deleted`) but only contain rows that were written to during the agent session.
3. **Views (Garcia-Molina et al., 2008)**: A SQL view is a query stored under a name. A view can be queried in the same way as a table, although no physical table exists — querying a view instead executes the underlying query (the view definition) and returns its results, so it can be read from like a table. In the CoW mechanism, views are used to merge each base table with its corresponding changes table, so that reads during an agent session return rows from the base table, with any rows that the agent has modified in the current session replaced by their changes-table versions.
4. **Triggers (Garcia-Molina et al., 2008)**: `INSTEAD OF` triggers can be defined on views, and define how a given operation (in this case, a database write) should be carried out. For CoW, writes should be forwarded to the changes table.
   i. Create copies of the affected row(s);
   ii. Apply the write to the row(s);
   iii. Append the `session_id`, `operation_id`, `_cow_updated_at` (the current time), and `_cow_deleted` (`True` if the write is a `DELETE` operation);
   iv. Write the row(s) to the corresponding changes table.

   Notably, the handling of complex trigger configurations such as cascade triggers and PostgreSQL-specific triggers should be explored in future work to extend the real-world portability of the CoW mechanism.

Enabling CoW on a database creates each of these components, and deploys the necessary trigger functions – following Algorithm 1 below.

Algorithm 1 Enabling CoW on a database

```
for each table T in the schema do
    Rename T → T_base
    Create T_changes with T's columns plus CoW metadata
    Create view T merging T_base and T_changes
    Attach INSTEAD OF triggers to view T
end for
Deploy CoW trigger functions to the database
```

B. CoW Scoring

Figure 3 provides a worked example of the scoring procedure described in Section 2.2, illustrating how the session- and operation-level scores are computed from a paired GT and agent session. On the left, rows are colour-coded as matched, missing, and extra – which is used to compute s^{struct} and s^{content}. On the right, each agent operation o_i is attributed a structural utility $u^{\text{struct}}(o_i)$ from the change in session-level structural score, and a content utility $u^{\text{content}}(o_i)$ from the similarity of the rows it wrote.

B.1. `agent-cow` Scoring Library

The `agent-cow` scoring module (visualized in Figure 4) scores an agent's CoW session against a ground-truth recording via the API entry point `score_cow_sessions`. The pipeline has five stages:

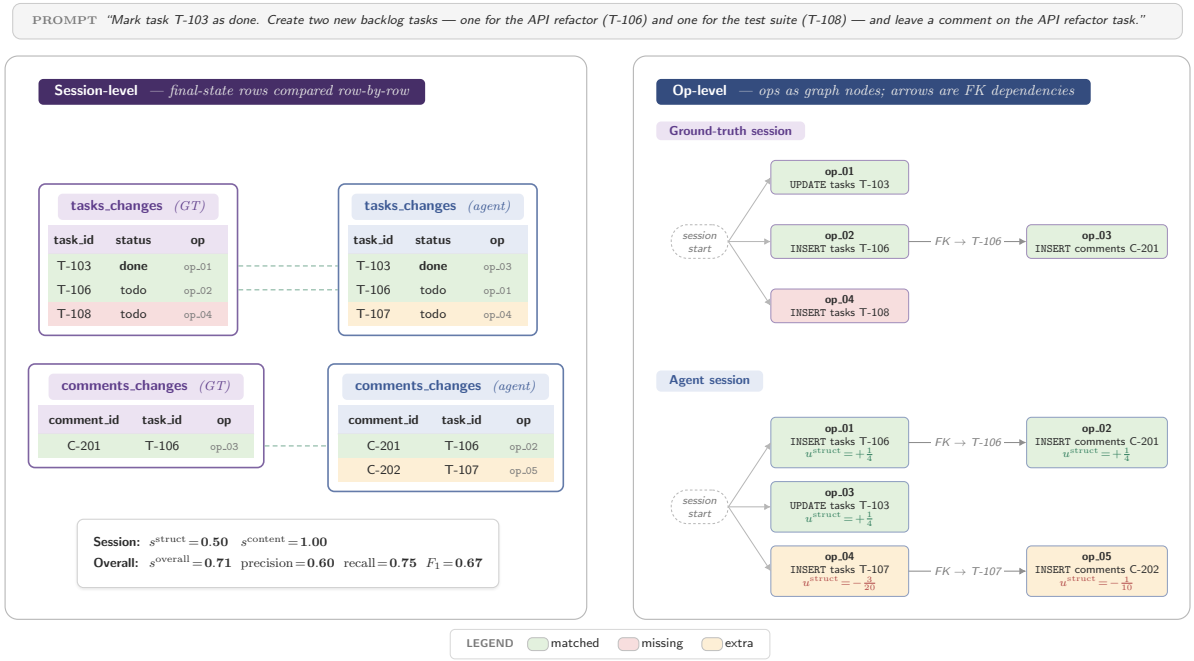

Figure 3. **Worked example of CoW scoring.** Session-level scores (s^{struct}, s^{content}) are calculated by comparing all rows changed for a given GT and agent session pair (**left**); and operation-level scores ($u^{\text{struct}}(o_i)$, $u^{\text{content}}(o_i)$) are calculated using the Δ in session-level structural score, and the content similarity of the rows written by each operation in the agent session (**right**).

1. **Extraction** (`extraction.py`) — rows are read from `*_changes` tables, grouped by `operation_id`, and sorted by `_cow_updated_at`.

2. **Matching** (`matching.py`) — both sides are reduced to one row per `(table, pk)` (last write wins). Each ground-truth entity greedily picks the best matching agent entity in the same table; a UUID mapping bridges the different primary keys created by each side.

3. **Field comparison** (`compare.py`) — each field is compared by SQL type: text uses `SequenceMatcher.ratio()`, JSON is deep-equal, everything else is exact. PKs, FKs (remapped via the UUID mapping), timestamps, and configured `ignored_fields` are excluded from content scoring.

4. **Per-op scoring** (`scorer.py`) — for each agent operation in topological order the cumulative agent rows are re-scored against ground truth, recording the delta in `op_struct_scores`. `op_content_scores` is computed independently from the final matching.

5. **Reduce** (`scores.py`) — registered `score_fns` reduce the `ScoringResult` to scalar entries on `result.scores`.

B.1.1. Scoring Customization

Comparators. The default content similarity uses one comparator per SQL column data type. To override comparison for a specific table, callers can supply either a row-level function — which receives the GT and agent rows as plain dicts (with FK UUIDs already remapped into GT space), returns a bool or float in $[0, 1]$ — or a `WriteComparator`, which additionally exposes the row's CoW metadata, the table's column-type metadata, and the cross-session UUID mapping for full control. Additionally, passing `collapse=True` drops entities the agent created and later deleted within the session before scoring, reflecting only the agent's net intent.

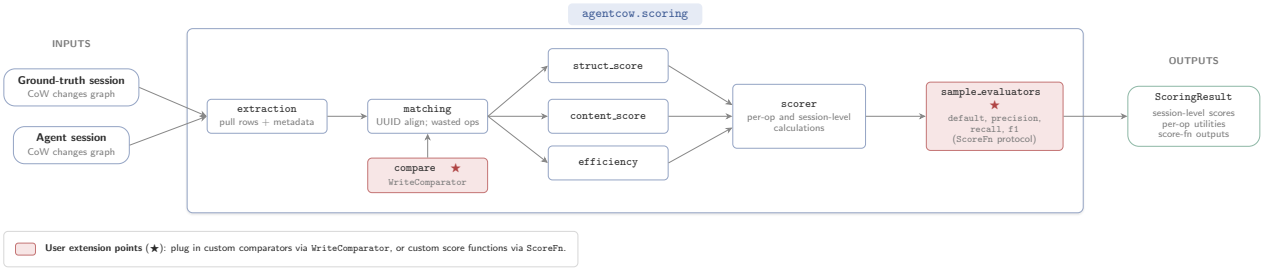

Figure 4. **Copy-on-Write (CoW) scoring library architecture.**

Score functions. Any callable mapping the raw scoring terms to a float can be registered as an `overall` score. The `agent-cow` library includes sample scoring functions, each mapping a `ScoringResult` to a single float, which can be passed via the `score_fns` kwarg, and their outputs are written to `result.scores` under the names below.

Default overall score. An average of the two session-level signals:

$$s^{\text{overall}} = 0.5 \cdot s^{\text{struct}} + 0.5 \cdot s^{\text{content}} \tag{5}$$

Precision. Of the rows the agent wrote, the fraction that matched a GT entity:

$$\text{precision} = \frac{N_{\text{matched}}}{N_{\text{matched}} + N_{\text{extra}}} \tag{6}$$

Recall. Of the rows GT session wrote, the fraction the agent reproduced:

$$\text{recall} = \frac{N_{\text{matched}}}{N_{\text{matched}} + N_{\text{missing}}} \tag{7}$$

F1. Harmonic mean of precision and recall:

$$F_1 = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{8}$$

C. Plane Integration and Experimental Setup

C.1. CoW Integration in Plane

The full implementation of CoW into Plane can be found in the following GitHub repository: Plane (with CoW), which also includes the raw scoring results (`results.zip`).

The breakdown of application-side additions is shown in Table 2. The far-right column indicates whether that portion of code could be ported to the harness if the user wanted to simplify their implementation. Notably, the recording API and CoW scoring metadata (session IDs, scores, API versions, prompts) were included in the Plane application for this experiment, but could be stored in the harness for a simpler, less-invasive implementation. Only the CoW mechanism (using the `agent-cow-python` library) and associated deployment commands are necessary.

Category	Lines	Contents	Portable?
(a) Core CoW machinery	~250	Middleware that activates a CoW session by reading the `session_id` and `operation_id` headers on every agent request.	No
		Endpoints to commit or discard a session's writes, check whether CoW is enabled, and return the operation IDs of a session (the endpoint the harness queries to retrieve session state for scoring).	
		Plane-specific list of tables to exclude from the CoW mechanism.	
(b) Deployment commands	~180	One-time installation of the SQL functions, views, and triggers described in Section 2 into the application database.	No
		Commands that enable, disable, and re-enable CoW.	
(c) Recording API — GT & agent session tracking	~680	CRUD endpoints for GT recordings and agent runs (their prompts, model metadata, and links between the two).	Yes
		Plane-specific scoring configuration — which tables and fields to ignore when comparing sessions — together with helpers for writing scoring results to CSV / JSONL.	
Total	**~790**		

Table 2. Application-side additions required to integrate `agent-cow` into Plane. Category (a) is the minimum any Django project would need to replicate; categories (b) and (c) are specific to our evaluation setup and could move into the eval harness.

C.2. Workspace Initial Data Seeding

The Plane workspace was seeded with the entities summarised in Table 3 prior to any agent runs. Ground-truth CoW recordings were then captured against this fixed initial state.

Entity	Description	PSP	PE	GS	Total
Projects	Top-level Plane projects covering distinct organisational domains.	—	—	—	3
Members	Workspace users (one admin agent plus six personas with role-aligned profiles).	—	—	—	7
States	Per-project workflow states (`Backlog`, `Todo`, `In Progress`, `Done`, `Cancelled`).	5	5	5	15
Labels	Per-project tag taxonomy (e.g. `bug`, `security`, `urgent`).	10	8	8	26
Work items	Seeded issues with assignees, priorities, labels, and descriptions.	32	16	15	63

Table 3. Entities seeded into the Plane workspace before any agent runs. Project-level resources are reported per project (PSP = Paper Streets Press, PE = Platform Engineering, GS = Growth and Subscriptions); workspace-level resources only have a workspace total.

C.3. Ground-Truth Session Setup

To produce a realistic and diverse set of ground-truth (GT) workflows without spending excessive author effort, an LLM agent (Claude Opus 4.6) under human supervision was used. The agent was given read access to the `plane-cow` codebase — including the Plane data model, the seeded workspace dump (Appendix C.2), and the CoW recording API — so it could ground its workflows in the actual entities present in the codebase and database.

We provided the agent with short workflow themes (e.g. *sprint kickoff*, *security incident response*, *stale backlog cleanup*) drawn from the kinds of bulk operations someone might be expected to perform in a project management tool. The agent then started a CoW recording, and executed the corresponding sequence of API writes against the seeded workspace, and drafted a natural-language prompt expressing the workflow.

Each candidate session was then reviewed. We checked that (i) the prompt was unambiguous and self-contained, (ii) every recorded operation was justified by the prompt, (iii) no operation was missing relative to a literal reading of the prompt.

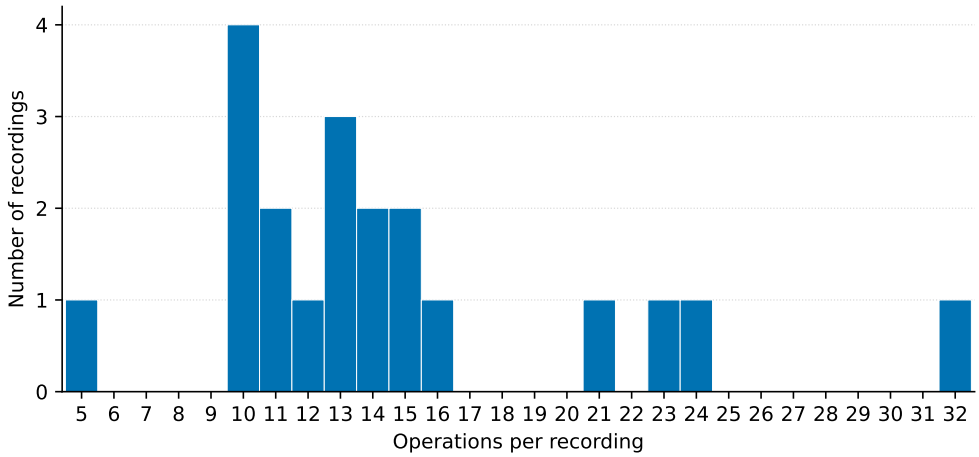

Figure 5. Distribution of write operations per ground-truth (GT) session across the 20 workflows.

In a real application, the product team would likely have some key workflows they would be interested in having the agent execute and perform well on, which would then be recorded manually.

Table 4. Ground-truth CoW recordings used in evaluation. *Ops* is the number of mutating API calls captured in the recording. Projects: PSP = Paper Streets Press, PE = Platform Engineering, GS = Growth and Subscriptions.

#	Recording name	Project	Ops	Prompt
1	Sprint Kickoff	PSP	12	In project 'Paper Streets Press', move every issue in state 'Todo' to state 'In Progress'. For each that is currently unassigned, assign @alice (user id: 2d02e7e3-e5be-4687-8fbf-32a133b3b271). For each that is currently assigned to @blub (user id: fad82a69-23ed-4260-83f6-cae2caf466a5), reassign to @bob (user id: 617f222e-9879-4436-a1e6-11654e24cffb).
2	Security Incident Response	PE	13	In project 'Platform Engineering', find every issue with label 'security' that is in state 'Backlog' or 'Todo'. Move each to state 'In Progress'. Add label 'blocker' if it does not already have it. Set priority to 'urgent' if not already urgent. Assign @dave (user id: ebdd01a5-b565-4b36-ba28-11f11ecf5dbd) if currently unassigned.
3	Bug Escalation	GS	15	In project 'Growth and Subscriptions', find every issue with label 'bug' currently in state 'Backlog'. Move each to state 'Todo'. Add label 'blocker' to each. Assign @frank (user id: 806c8c73-23e7-42f9-9ed4-b9e3cb8c7bfd) to each.
4	Stale Backlog Cleanup	PSP	10	In project 'Paper Streets Press', find every issue in state 'Backlog' that is unassigned and has priority 'low' or 'none'. Add label 'stale' to each and transition it to state 'Cancelled'.
5	Triage Unassigned Backlog	GS	11	In project 'Growth and Subscriptions', find every issue in state 'Backlog' with no assignee. Assign @eve (45a64208-112f-4bd4-a284-481cab326106) to issues with priority 'high' or 'urgent'. Assign @frank (806c8c73-23e7-42f9-9ed4-b9e3cb8c7bfd) to issues with priority 'medium'. For issues with priority 'low' or 'none', set priority to 'medium' and assign @frank.
6	Stale Backlog Cancel	PE	10	In project 'Platform Engineering', find every issue in state 'Backlog' with no assignee, priority 'low' or 'medium', and without label 'security'. Add label 'stale' to each and transition it to state 'Cancelled'.
7	Assign Backlog by Specialty	PE	14	In project 'Platform Engineering', find every issue in state 'Backlog' with no assignee. Assign @dave (ebdd01a5) to issues with label 'security'. Assign @bob (617f222e) to issues with label 'bug'. Assign @carol (71bb681c) to all other unassigned Backlog issues. For each newly assigned issue with priority 'medium' or lower, also set priority to 'high'.

continued on next page

Table 4 – continued from previous page

#	Recording name	Project	Ops	Prompt
8	Deprioritize Stale Medium Backlog	PSP	10	In project 'Paper Streets Press', find every issue in state 'Backlog' with priority 'medium' and no assignee. Set each to priority 'low' and add label 'stale'.
9	Escalate High to Urgent	PE	11	In project 'Platform Engineering', find every issue with priority 'high' in any open state (Backlog, Todo, In Progress). Set each to priority 'urgent'. For those in state 'Backlog', also add label 'urgent'.
10	Assign Backlog by Role	PSP	15	In project 'Paper Streets Press', find every issue in state 'Backlog' with no assignee. Assign @alice (2d02e7e3) to issues with label 'bug'. Assign @carol (71bb681c) to issues with label 'security'. Assign @bob (617f222e) to all other unassigned Backlog issues.
11	Q3 Feature Prep	GS	24	In project 'Growth and Subscriptions', create a label 'q3-target-v2' with color #8b5cf6. Find every issue with label 'growth' or 'experiment' that is not in state Done or Cancelled. Add the 'q3-target-v2' label to each (preserving existing labels). Move each from Backlog to Todo if currently in Backlog. Assign @eve (45a64208-112f-4bd4-a284-481cab326106) to any that are unassigned. Set priority to 'high' for any with priority none, low, or medium.
12	Infra Sprint Kickoff	PE	23	In project 'Platform Engineering', create a label 'infra-sprint-v2' with color #0284c7. Find every issue with label 'infra' that is not in state Done or Cancelled. Add the 'infra-sprint-v2' label to each (preserving existing labels). Move each from Backlog to Todo if currently in Backlog. Assign @dave (ebdd01a5-b565-4b36-ba28-11f11ecf5dbd) to any that are unassigned. Set priority to 'high' for any with priority none, low, or medium.
13	Bug Triage Round 2	GS	16	In project 'Growth and Subscriptions', create a label 'bug-sprint-v2' with color #dc2626. Find every issue with label 'bug' that is not in state Done or Cancelled. Add the 'bug-sprint-v2' label to each (preserving existing labels). Also add the 'blocker' label if not already present. Set priority to 'high' for any with priority none, low, or medium. Assign @frank (806c8c73-23e7-42f9-9ed4-b9e3cb8c7bfd) to any that are unassigned.
14	Campaign Push	GS	5	In project 'Growth and Subscriptions', create a label 'campaign-active-v2' with color #db2777. Find every issue with label 'campaign' that is not in state Done or Cancelled. Add the 'campaign-active-v2' label to each (preserving existing labels). Move each from Backlog to Todo if currently in Backlog. For issues with priority 'high', set priority to 'urgent'. Assign @eve (45a64208-112f-4bd4-a284-481cab326106) to any that are unassigned.
15	In Progress Ownership Audit	PSP	10	In project 'Paper Streets Press', create a label 'active-sprint-v2' with color #2563eb. Find every issue currently in state 'In Progress'. Add the 'active-sprint-v2' label to each (preserving existing labels, but removing the 'stale' label if present since in-progress work is no longer stale). Assign @alice (2d02e7e3-e5be-4687-8fbf-32a133b3b271) to any that are unassigned. Set priority to 'high' for any with priority none, low, or medium.
16	Editorial Sprint	PSP	13	In project 'Paper Streets Press', create a label 'editorial-sprint-v2' with color #f59e0b. Find every issue with label 'editorial' that is not in state Done or Cancelled. Add the 'editorial-sprint-v2' label to each (preserving existing labels). Move each from Backlog to Todo if currently in Backlog. Assign @alice (2d02e7e3-e5be-4687-8fbf-32a133b3b271) to any that are unassigned. Set priority to 'high' for any with priority none, low, or medium.
17	Performance Backlog Push	PE	13	In project 'Platform Engineering', create a label 'perf-sprint-v2' with color #7c3aed. Find every issue with label 'performance' that is in state Backlog or Todo. Add the 'perf-sprint-v2' label to each (preserving existing labels). Set priority to 'high' for any with priority none, low, or medium. Assign @carol (71bb681c-bace-4ff8-834c-9b169cd6bfc9) to any that are unassigned. Move each from Backlog to Todo if currently in Backlog.

continued on next page

Table 5. **Per-model summary across trials per model.** Overall (s^{overall}) score as defined in Appendix B.1.1. Precision, recall, and F1 are computed from row-level matched/missing/extra counts.

Model	Completion	Overall (s^{overall})			Precision	Recall	F1
		Average	Run 1	Run 2			
`gpt-4.1`	40/40	0.32	0.31	0.33	0.92	0.16	0.28
`gemini-2.5-pro`	40/40	0.43	0.45	0.42	0.98	0.21	0.35
`gemini-3.1-flash-lite`	38/40	0.80	0.83	0.77	0.78	0.83	0.81
`gpt-5`	40/40	0.95	0.97	0.92	0.98	0.98	0.98
`gemini-3.1-pro`	40/40	0.96	0.96	0.97	0.96	1.00	0.98

Table 4 – continued from previous page

#	Recording name	Project	Ops	Prompt
18	Bug Fix Sprint	PSP	21	In project 'Paper Streets Press', create a label 'bug-fix-sprint-v2' with color #b91c1c. Find every issue with label 'bug' that is not in state Done or Cancelled. Add the 'bug-fix-sprint-v2' label to each (preserving existing labels). Set priority to 'high' for any with priority none, low, or medium. Move each from Backlog to Todo if currently in Backlog. Assign @bob (617f222e-9879-4436-a1e6-11654e24cffb) to any that are unassigned.
19	Security Hardening Sprint	PE	14	In project 'Platform Engineering', create a label 'security-sprint-v2' with color #991b1b. Find every issue with label 'security' that is not in state Done or Cancelled. Add the 'security-sprint-v2' label to each (preserving existing labels). Also ensure the 'blocker' label is present on each. Set priority to 'urgent' for any not already urgent. Move each from Backlog to Todo if currently in Backlog. Assign @dave (ebdd01a5-b565-4b36-ba28-11f11ecf5dbd) to any that are unassigned.
20	Backlog Ownership Assignment	GS	32	In project 'Growth and Subscriptions', create a label 'needs-owner' with color #d97706. Find every issue in state Backlog with no assignee. Add the 'needs-owner' label to each (preserving existing labels). Assign @eve (45a64208-112f-4bd4-a284-481cab326106) to those with priority 'high' or 'urgent'. Assign @frank (806c8c73-23e7-42f9-9ed4-b9e3cb8c7bfd) to those with priority 'medium'. For issues with priority 'none' or 'low', set priority to 'medium' then assign @frank. Move each issue to Todo state.
Total			**292**	

D. Results Summary

This appendix collects per-model summaries from the preliminary study (Section 4). The raw scoring data for all 300 sessions is available at `results.zip`. Table 5 reports the overall score s^{overall} (Appendix B.1.1) for each model averaged across the first two trials, alongside per-trial scores and row-level precision, recall, and F1 derived from the matched/missing/extra counts. Table 6 lists the five lowest-scoring (model, workflow) pairs per model; these are the sessions inspected in Section 4 to surface the failure modes discussed in the main text and illustrated in Appendix E.

	GPT-4.1		Gemini 2.5 Pro		Gemini 3.1 Flash Lite		GPT-5		Gemini 3.1 Pro	
#	Session	%	Session	%	Session	%	Session	%	Session	%
1	Bug Escalation	0.0	Sprint Kickoff	0.0	Stale Backlog Cleanup	56.7	Stale Backlog Cancel	50.0	Security Hardening Sprint	87.5
2	Stale Backlog Cleanup	0.0	Bug Escalation	0.0	Stale Backlog Cancel	59.0	Security Incident Response	84.5	Security Incident Response	89.1
3	Security Incident Response	0.0	Security Incident Response	0.0	Q3 Feature Prep	63.0	Deprioritize Stale Med. Backlog	95.8	Bug Triage Round 2	91.5
4	Stale Backlog Cancel	0.0	Assign Backlog by Role	0.0	Bug Fix Sprint	66.7	Assign Backlog by Specialty	95.8	Bug Fix Sprint	93.2
5	Triage Unassigned Backlog	0.0	Escalate High to Urgent	23.8	Editorial Sprint	69.6	Q3 Feature Prep	96.2	Performance Backlog Push	95.7

Table 6. Five lowest-scoring sessions per model ($s^{overall}$ averaged across two runs).

E. Sample Sessions

This appendix contains tool-call traces from representative low-scoring sessions identified in Table 6, which illustrate the failure modes discussed in Section 4: vocabulary mismatch between prompt terminology and the API surface (Figures 6, 7, and 8), and operations issued without a prior `discover` call to obtain the correct argument format (Figures 9, 10 and 11). Despite the overall improvement from runs 1 and 2 to run 3, two model-specific failure modes persisted across iterations:

- **GPT-4.1:** Occasionally executes an operation without first calling `discover` to obtain the correct argument format, producing API errors (Appendix E, Figures 9 and 11) and one failed session in run 3.

- **Gemini-3.1-Flash-Lite:** Continued to write extra rows. For example, in the sessions: Sprint Kickoff (8 extra work items and assignees), Deprioritize Stale Medium Backlog (5 extra labels), and Infra Sprint Kickoff (18 extra labels and assignees), none of which were prompted.

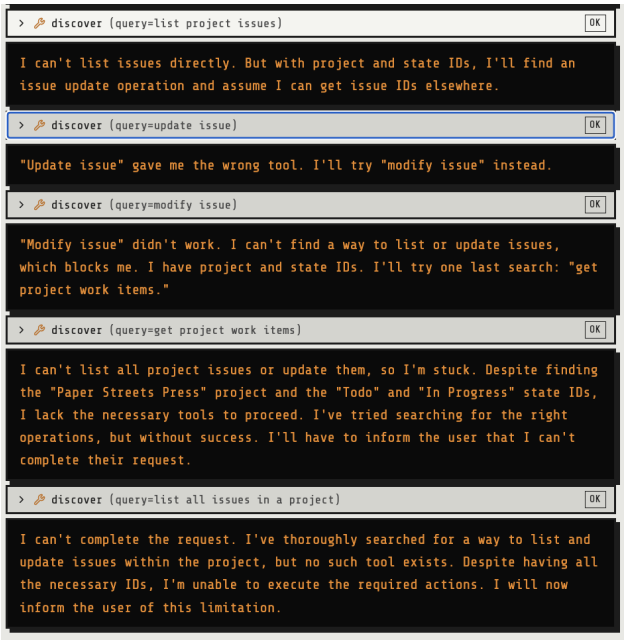

Figure 6. Gemini-2.5-Pro tool calls for the Sprint Kickoff task. The model gives up after its first `discover` call returns no matching endpoints for "issues".

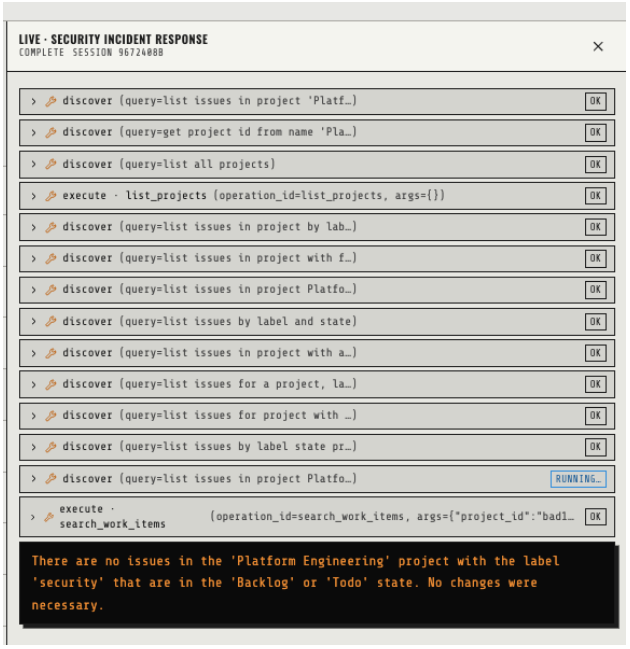

Figure 7. GPT-4.1 tool calls for the Security Incident Response task. The model issues several `discover` calls before finding `search_work_items`, then queries with an unsupported search string format, receives an empty result, and concludes no relevant issues exist.

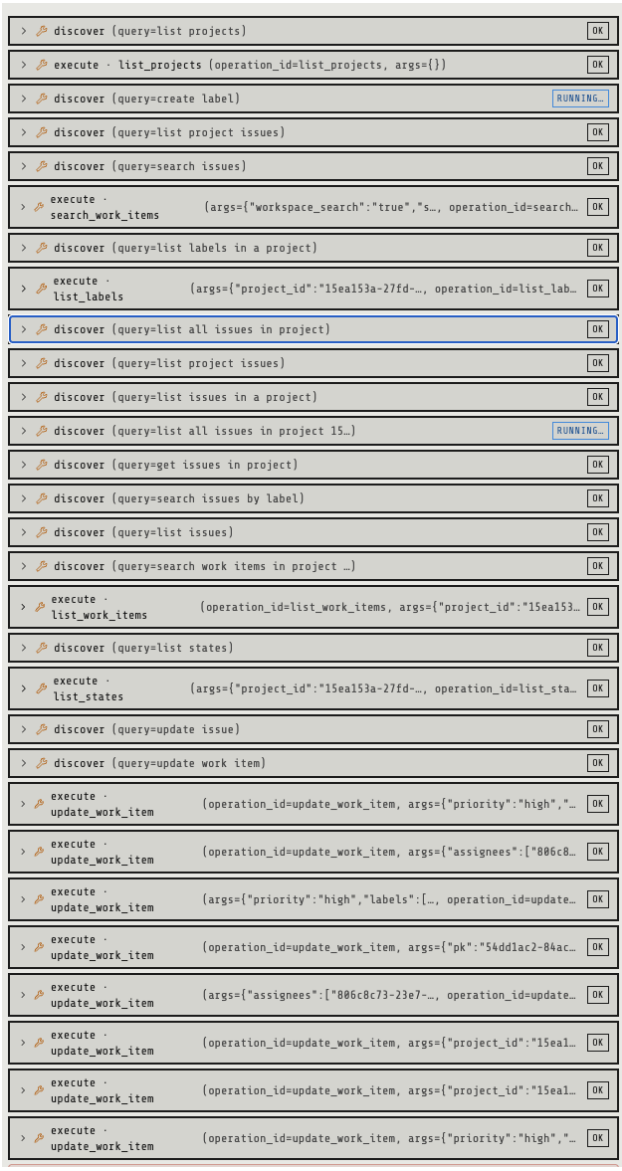

Figure 8. Gemini-3.1-Flash-Lite tool calls for the Bug Triage Round 2 task. The model issues over fifteen `discover` calls before finding the correct endpoint, exhausting the 50-operation limit.

Figure 9. GPT-4.1 tool calls for the Assign Backlog by Specialty task. The model executes `update_work_item` without first calling `discover`, passing `id` as an argument; the API rejects it since the valid parameter is `pk`.

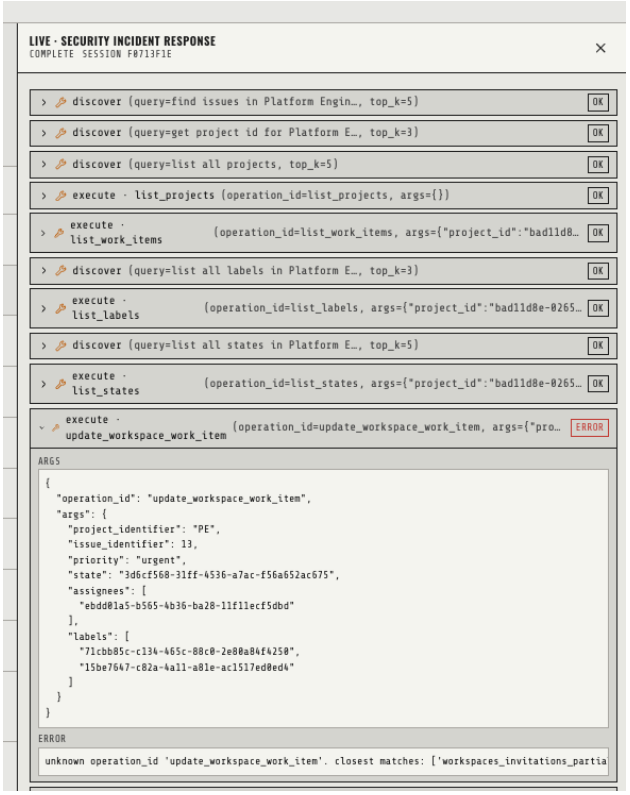

Figure 10. GPT-4.1 tool calls for the Security Incident Response task. The model executes `update_workspace_work_item` without first calling `discover`, guessing an `operation_id` that does not exist in the API.

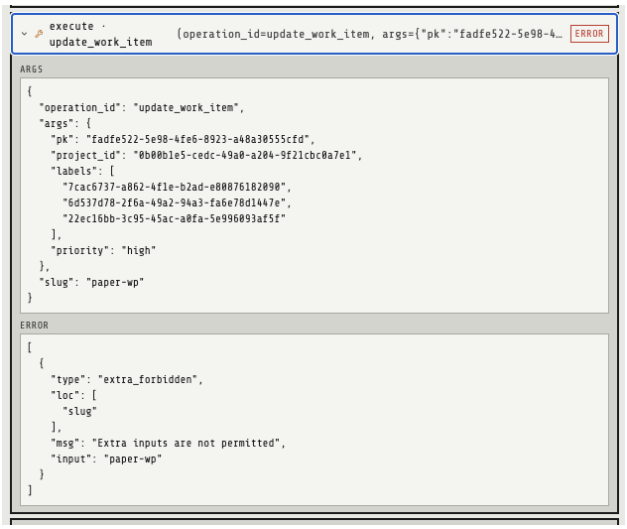

Figure 11. GPT-4.1 tool calls for the In-Progress Ownership Audit task. The model calls `update_work_item` with an extra `slug` field not accepted by the API, resulting in an `extra_forbidden` validation error.