PROGRAMDISTILL: FROM INTERACTIVE WEB APPS TO VERIFIABLE REFERENCE-GUIDED SWE TASKS

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents are usually evaluated on tasks whose desired behavior is stated in an issue, instruction, or test. In web development, however, the specification often exists only as working software, such as a prototype or a comparable product. We study reference-guided software engineering, in which an agent discovers missing behavior by interacting with a live reference whose source is hidden and implements it in an incomplete codebase. We introduce **ProgramDistill**, a benchmark in which a recorded browser workflow of a running application serves as both a task and its verifier. Its automated task generation pipeline, **mine–craft–patch**, deletes the code behind each workflow and replays the workflow to check the repair, so no issues or human-written tests are needed. Masking several dependent behaviors controls *restoration depth*, yielding 4,063 tasks from 26 applications. Across nine frontier agents, performance drops sharply with depth. GPT-6 Astra falls from 100% at depth 1 to 64.0% at depth 8, and every other agent retains less than half of its depth-1 performance. When rebuilding entire applications, the best agent passes only 49.2% of end-to-end workflows. Most unrecovered behaviors (59.2%) were never observed in the reference, suggesting that exploration and validation, not only code generation, are key bottlenecks.

1 INTRODUCTION

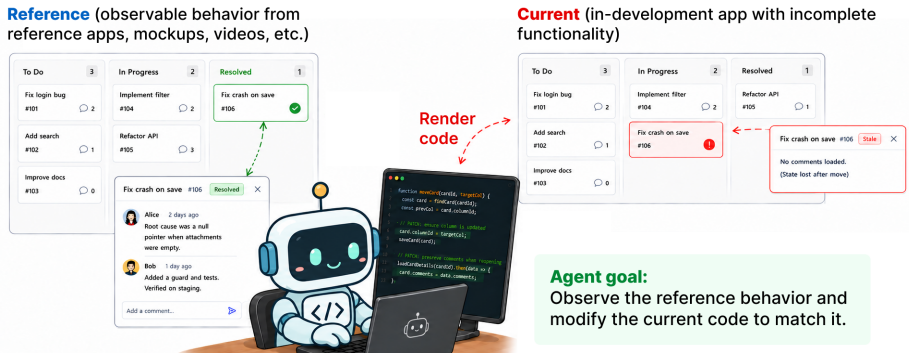


Figure 1: **Reference-guided software engineering.** An agent interacts with a working reference whose source is hidden, modifies an editable current application with missing functionality, and validates the recovered behavior against the reference.

Large language models (LLMs) increasingly act as coding agents that inspect repositories and modify source code (Jimenez et al., 2024; Yang et al., 2024; Sonwane et al., 2025; Jain et al., 2025; Xu et al., 2026). Their benchmarks usually state the desired behavior in an issue, instruction, or test. In web development, however, the specification often exists only as working software, such as an earlier product version, a prototype, or a comparable application. The agent must observe the reference, implement the missing behavior, and validate the result against it (Figure 1). Even dragging a card may require recovering both its visible effect and the state that persists afterward.

Our key observation is that record-and-replay, a long-standing technique in GUI testing (Gomez et al., 2013; Leotta et al., 2013), can turn a running application into both tasks and their verifiers. We record

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

a user-level behavior on the reference, such as renaming a list on a board, as a browser workflow. Deleting the code behind it makes the replay fail, and restoring the behavior makes it pass again, so the same recording defines a task and grades it. This reverses the usual order of SWE benchmarks, which start from a task, such as an issue to resolve, a bug to repair, or a program to rebuild, and then collect tests to grade it. Because we start from the behavior, the unit being restored can be the unit being graded, and no existing issues or tests are needed.

Unlike CLI programs, whose behaviors can be checked as independent input and output pairs, web applications have stateful behaviors that depend on each other. A user logs in before creating a board and creates it before editing it. These *prerequisite lineages* let us mask several behaviors along one chain, turning a single-behavior *atomic* task into a harder *cumulative* task. The number of masked behaviors defines *restoration depth*. We call this transformation *application-to-task factorization*.

The resulting tasks evaluate *reference-guided reconstruction*, in which an agent given a working reference and an incomplete implementation must discover missing behavior through interaction and reproduce it through source changes. Together, application-to-task factorization and reference-guided reconstruction define **ProgramDistill**, reflecting two levels of distillation, from applications into tasks and verifiers and from reference behavior into code that reproduces it. We implement it with an automated **mine–craft–patch** pipeline. Mining discovers replayable behaviors and their lineages, crafting masks the responsible implementation, and patching evaluates agents in two settings, *partial reconstruction* of masked behaviors in an otherwise intact codebase and *full reconstruction* from a minimal scaffold. Across 26 applications, the pipeline mines 1,975 behaviors and constructs 4,063 tasks without human intervention.

Among nine frontier coding agents, GPT-6 Astra (OpenAI, 2026) and Claude Opus 5 (Anthropic, 2026) recover 49.2% and 28.8% of cumulative workflows in full-application reconstruction. In partial reconstruction, their success falls from 100% to 64.0% and from 96% to 32%, respectively, as restoration depth increases from 1 to 8. As depth increases, reconstruction burden grows while observation per required behavior falls sharply. Astra achieves the strongest repair performance while making the most observations and the fewest edit/write steps. These results suggest that effort allocation across observation, validation, and editing is an important dimension of reference-guided software engineering alongside implementation capability.

Our contributions are (1) ProgramDistill, a benchmark and formulation for reference-guided software engineering spanning atomic, cumulative, and full-application reconstruction, (2) a behavior-first design in which each mined behavior defines a task, serves as its verifier, and acts as the unit of composition, enabling per-behavior diagnosis, (3) restoration depth as a controlled difficulty axis, along which all nine evaluated agents degrade substantially, and (4) an LLM-assisted failure analysis indicating that most unrecovered behaviors were never observed in the reference and that agents often do not retest the workflows their edits affect.

2 Related Work

Prior benchmarks differ from ours in the target they measure, the specification, and the verifier. SWE-bench-style benchmarks and task generators (Jimenez et al., 2024; Jain et al., 2025; Yang et al., 2026a) measure whether agents resolve repository issues or bugs, specify behavior through issues, and verify patches with tests.

Design2Code (Si et al., 2025) and Interaction2Code (Xiao et al., 2025) measure whether agents implement given designs as web pages and compare them against fixed visual or interaction artifacts, while WebArena (Zhou et al., 2024) measures task completion on existing sites without modifying them.

ProgramBench (Yang et al., 2026b) and MirrorCode (Adamczewski et al., 2026) ask agents to reimplement an entire CLI program by probing it from a terminal, and ProgramBench grades submissions with separate hidden tests over stdout, exit codes, and files. ProgramDistill instead measures whether agents discover unstated behavior by operating a running web application and restore it in code. The specification is the application itself, and the verifier replays browser workflows mined from it, checking UI, URL, element, and persisted state. To our knowledge, ProgramDistill is the first benchmark in which workflows recorded on a reference web application serve both to construct coding tasks and to verify solutions. Appendix A gives an extended discussion.

3 APPLICATION-TO-TASK FACTORIZATION VIA MINE–CRAFT–PATCH

Figure 2 summarizes our automated multi-agent LLM pipeline for constructing verifiable tasks without human intervention. We use GPT-5.6 Sol (OpenAI, 2026b) for all LLM roles in benchmark construction.

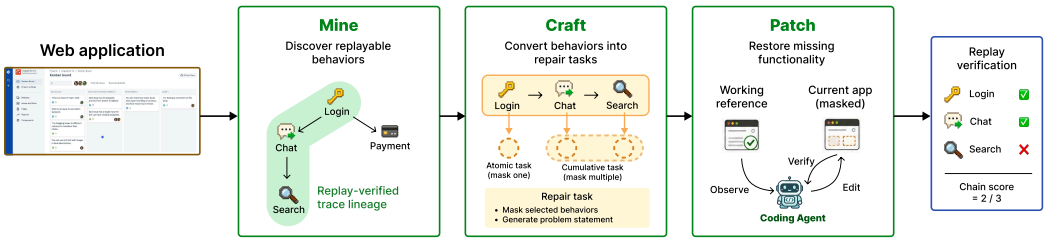


Figure 2: **Mine–craft–patch.** Mining discovers replayable behaviors and their prerequisites. Crafting masks individual behaviors and composes masks along prerequisite lineages. Patching restores missing behavior from a working reference and replays the lineage to measure recovery.

3.1 DETERMINISTIC AND OBSERVABLE APPLICATION INSTANCES

We use 26 self-contained web applications drawn from the OSWorld suite (Xie et al., 2024), real-world open-source projects, and SaaS clones. Appendix B gives the corpus and source revisions. Each application has a fixed production-build *reference* and an *editable* development instance with hot reload. We reset application state before collection and replay, and use a shared deterministic clock across the database, backend, and frontend. Appendix B.2 details deterministic execution.

LLM agents interact with both the working reference and editable current application through a shared Playwright-based browser helper (Microsoft, 2024), which executes actions on targets identified by stable observable attributes and waits for application state to settle before returning visible text, accessibility information, and interactive elements. Appendix C gives details.

3.2 MINING: DISCOVERING AND VERIFYING INTERACTIVE BEHAVIORS

Mining starts from behavior goals grounded in source and UI evidence. For each goal, the pipeline resets the application, replays any required prerequisite traces, and lets an LLM agent explore the new behavior from live observations. The resulting browser actions and selected success signals form an exploratory trace $\tilde{\tau}_d$, which is stored with an optional parent link.

Because exploration may contain unnecessary detours, each successful behavior is re-collected from a clean state using $\tilde{\tau}_d$ only as route context, similar to privileged-context self-distillation (Kim et al., 2026). The agent still chooses actions from live observations and selects expected signals exposed by the environment. The resulting trace τ_d records replay-stable actions, expected signals, and an optional parent. Only replay-verified traces are admitted and then relabeled according to the behavior actually achieved. Goals guide exploration, but tasks are defined by verified behavior. Parent links organize them into prerequisite lineages $L_d = (\tau_1, \ldots, \tau_d)$.

Figure 3 shows this structure in Trello, where account setup enables board/card creation, which in turn enables card editing.

Mining is implemented by four specialized LLM roles: the Planner proposes grounded behavior goals, the Collector executes and records them, the Relabeler describes the behavior actually achieved, and the Reflector guides subsequent exploration rounds. Appendix E.1 details their orchestration, while Appendix E.6 shows an example admitted trace.

Replay verification. Following record-and-replay GUI testing (Gomez et al., 2013; Leotta et al., 2013), each trace records user actions on replay-stable selectors and the resulting signals, such as visible text, accessible names, URLs, and persisted state. For application A, the verifier $V(A, L_d) \in \{0, 1\}$ resets application and browser state, replays $\tau_1, \ldots, \tau_d$ without LLM intervention, and returns

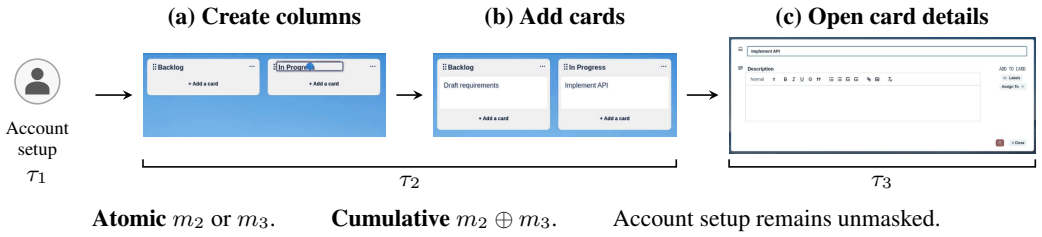
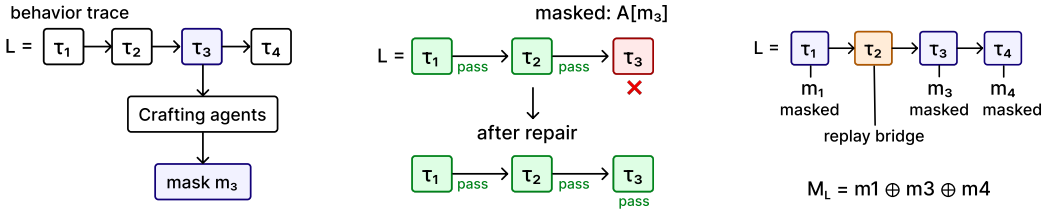


Figure 3: **Behavioral dependencies in `Trello (Knowankit)`.** Account setup (τ_1) precedes board/card creation (τ_2), which precedes editing (τ_3). Crafting masks behaviors while preserving prerequisites, and can compose masks into cumulative tasks.



(a) Trace-conditioned mask (b) Atomic mask (c) Cumulative mask

Figure 4: **Crafting repair tasks.** Atomic validation preserves prerequisites while breaking the target behavior. Cumulative tasks compose validated masks while retaining unmasked replay bridges. Gold-patch replay verifies recovery of the complete lineage.

1 only if all recorded actions complete and all expected signals hold. Unlike conventional record-and-replay, which tests only the application it was recorded on, the same recording both builds and grades the benchmark, admitting traces during mining, validating masks during crafting, and grading repairs of a different implementation during evaluation.

3.3 Crafting: From Verified Behaviors to Repair Tasks

Crafting converts each verified behavior into a repair task by masking its source implementation while preserving its prerequisites. A *logic-only* mask removes the implementation behind an existing UI, while a *logic-and-UI* mask also removes the UI owned by the feature. For target τ_d, crafting agents propose a mask m_d and a problem statement describing the feature's user-facing purpose and workflow. The original implementation, repair procedure, and exact verification signals remain hidden. Appendix F gives the crafting prompts.

Atomic task construction. For each verified target behavior τ_d, crafting proposes a mask m_d that removes its implementation while preserving the behaviors required to reach it. Since $V(A, L_d) = 1$ before masking, for $d > 1$ we require

$$\text{Build}(A[m_d]) = 1, \qquad V(A[m_d], L_{d-1}) = 1, \qquad V(A[m_d], L_d) = 0.$$

Thus, the masked application must still build, its prerequisite lineage must remain valid, and the target behavior must fail. For a root trace, only the build and target-failure conditions apply. These correspond to fail-to-pass (F2P) restoration and pass-to-pass (P2P) prerequisite preservation (Jimenez et al., 2024). A separate mask-depth critic rejects superficial changes that leave substantive implementation intact. If no valid mask is found for a behavior, it is not used as a repair target, but its trace can still serve as a prerequisite for later behaviors.

Cumulative task construction and gold patches. A cumulative task combines validated masks of dependent behaviors along the same lineage. For L_d, let $j_1 < \cdots < j_{r_L} = d$ index the repair targets. Their masks compose as $M_L = m_{j_1} \oplus \cdots \oplus m_{j_{r_L}}$, producing $A[M_L]$. The *restoration depth* r_L counts masked repair targets, excluding prerequisite traces that are replayed but not masked. Thus, $r_L \leq d$, with $r_L = 1$ for atomic tasks. In Figure 3, $m_2 \oplus m_3$ masks board/card creation and editing while leaving account setup intact.

Because each atomic mask is a source-code diff against the same intact baseline, multiple masks can be merged like source-code changes to form a cumulative mask. Non-conflicting changes merge automatically, while conflicts are checked with a `git` three-way merge and resolved by an LLM merge agent. Appendix F.3 gives details.

The merged mask defines the final masked application for the cumulative task. Reversing this masking diff yields a *gold patch*, which must restore replay of the complete lineage. This provides an end-to-end positive control alongside the target-failure and prerequisite-preservation checks.

3.4 Patching: Restoring Behavior from a Working Reference

Partial-application reconstruction. We evaluate reference-guided reconstruction by giving an agent a masked repository, a behavioral problem statement, and access to the working reference. The agent edits the hot-reloading current application to recover the missing behavior. The reference source, masking diff, gold patch, and grading traces remain hidden, and public network egress is disabled. Appendix D.1 details runtime isolation, while Appendix G gives the repair-agent instructions.

Evaluation. For submitted patch Δ, let $A[M_L, \Delta]$ denote the repaired application. Evaluation reuses the recorded actions, selectors, and expected signals unchanged, and both repair targets and replay bridges must pass. The implementation need not match the gold patch, but must reproduce the recorded outcomes, including labels and accessible names, which are part of the specification. We define

$$\text{BinaryScore}(L_d, \Delta) = V(A[M_L, \Delta], L_d), \quad \text{ChainScore}(L_d, \Delta) = \frac{1}{r_L} \sum_{k=1}^{r_L} V(A[M_L, \Delta], L_{j_k}). \tag{1}$$

Binary score requires the complete lineage to pass, while chain score credits the recovered prefix. Because each term replays the full prefix up to its target, a failure also fails every later term, so chain score equals the fraction of targets recovered before the first failure. In the Trello example in Figure 3, the cumulative task masks τ_2 and τ_3 while leaving the prerequisite τ_1 intact. If τ_2 passes but τ_3 fails, the chain score is $1/2$ and the binary score is 0.

Full-application reconstruction. We also evaluate a harder setting in which the existing implementation is removed entirely. The agent starts from a minimal executable scaffold and rebuilds the application by interacting with the working reference. Mined traces serve directly as behavioral tests. Atomic recovery evaluates one target after replaying its prerequisites, while cumulative recovery evaluates complete prerequisite lineages with chain score crediting recovered prefixes. Here, prerequisite replay tests workflow correctness rather than preservation of an existing implementation.

4 Experiments

4.1 Benchmark Generation

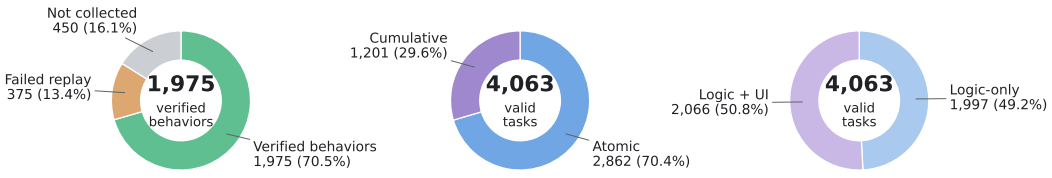
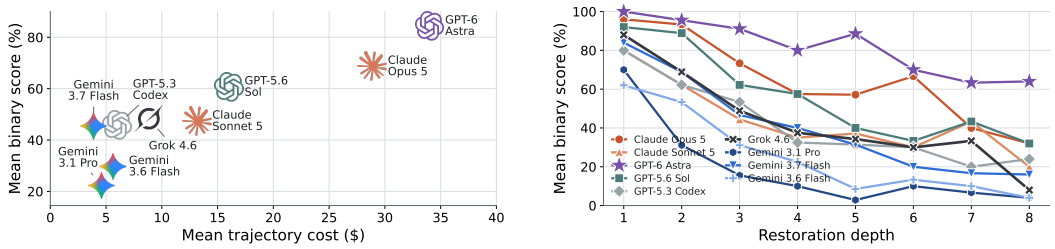


Figure 5: **Benchmark generation.** (a) Mining yields 1,975 verified behaviors. (b) The 4,063 tasks comprise 2,862 atomic and 1,201 cumulative tasks. (c) Mask scopes comprise 1,997 logic-only and 2,066 logic-and-UI tasks.

Our pipeline yields a large set of replay-verified behaviors and repair tasks across 26 applications (Figure 5). Mining starts from 2,800 proposed goals, collects 2,350 traces, reproduces 2,165 under



(a) Cost vs. mean binary score

(b) Mean binary score by restoration depth

Figure 6: **Partial-application reconstruction.** (a) Mean binary score versus mean trajectory cost. (b) Binary score by restoration depth r_L. Appendix I.2 reports the corresponding chain scores.

clean-state replay, and retains 1,975 after final verification. The resulting prerequisite trees reach depth 17, with mean depth 3.40, maximum width 60, and mean branching factor 1.78. Appendix B.1 shows the resulting lineage structure across applications.

To make evaluation tractable across nine models, we construct `ProgramDistill-300`, stratified by restoration depth with 50, 45, 45, 40, 35, 30, 30, and 25 tasks at depths 1 through 8. Round-robin selection within each depth preserves coverage of all 26 applications. The suite contains 269 distinct lineages, 50 atomic and 250 cumulative tasks, and 140 logic-only and 160 logic-and-UI masks.

4.2 Partial-Application Reconstruction

We evaluate nine frontier models at high reasoning effort: GPT-6 Astra (OpenAI, 2026), GPT-5.6 Sol (OpenAI, 2026b), Claude Opus 5 (Anthropic, 2026), Claude Sonnet 5 (Anthropic, 2026a), Gemini 3.7 Flash (Google DeepMind, 2026c), Gemini 3.6 Flash (Google DeepMind, 2026b), Gemini 3.1 Pro Preview (Google DeepMind, 2026a), Grok 4.6 (xAI, 2026), and GPT-5.3 Codex (OpenAI, 2026a). All use the same modified R2E-Gym harness (Jain et al., 2025) with `execute_bash`, `file_editor`, `search`, and `finish`, together with the `browser` CLI through Bash. Appendix D gives the full harness configuration.

Performance and cost. Figure 6(a) compares repair performance and trajectory cost across the nine models. Astra, Opus 5, and Sol achieve the highest binary scores at 84.3%, 68.7%, and 60.7%, with mean costs of $33.99, $28.86, and $16.01. Grok, Sonnet, Codex, and Gemini 3.7 achieve similar performance, with binary scores between 45.3% and 48.3%.

Cost does not consistently track performance. For example, Gemini 3.7 nearly matches Codex at 45.3% versus 45.7% while costing $3.89 rather than $5.99.

Mask scope. Every model scores substantially lower on logic-and-UI tasks, which also remove feature-owned UI, than on logic-only tasks, with binary-score gaps of 15.4 to 33.1 points (Appendix I.1). Astra drops from 92.9% to 76.9%, and the gap persists under the chain score (96.2% vs. 84.9%).

Performance declines with restoration depth. Each atomic mask is individually validated, and every cumulative task passes end-to-end replay with its gold patch, providing a positive control that each task is solvable. Even so, performance declines as restoration depth increases (Figure 6(b)). At depth 1, Astra solves all 50 tasks, compared with 96% for Opus and 92% for Sol. At depth 8, Astra reaches 64.0%, while Opus and Sol both reach 32.0%. Every other model retains less than half of its depth-1 performance. Appendix I.3 shows that this effect persists among runs with similar mean prompt sizes. Appendix I.5 shows a depth-8 `Trello (Vdevired)` task that Opus 5 repairs over 317 steps through repeated reference inspection, editing, and validation.

4.2.1 Analysis

Interaction patterns across models. Figure 7 shows distinct interaction patterns across models. Astra makes the most reference and current-app observations and the fewest edit/write steps. It

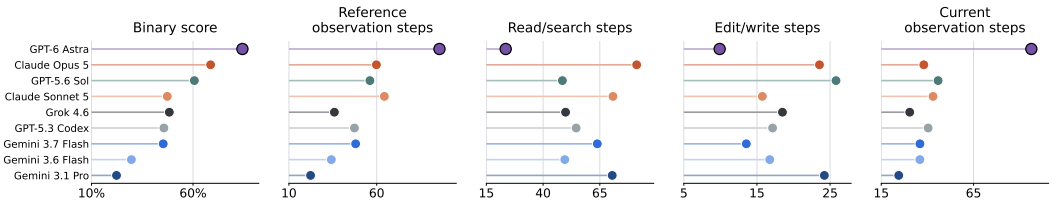


Figure 7: **Performance and interaction behavior.** Mean binary score alongside reference observation, read/search, edit/write, and current-app observation steps per trajectory. Observations count state requests, not necessarily successful checks.

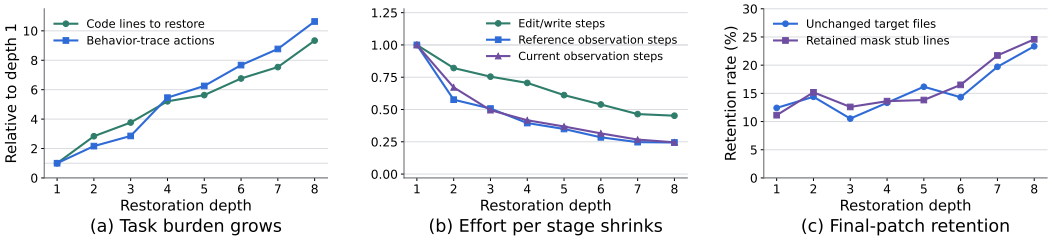


Figure 8: **Scaling with restoration depth.** (a) Repair burden and (b) agent effort per repair target, both normalized to depth 1. (c) Fractions of target files left unchanged and mask-added stub lines retained in the final patch.

averages 96.3 current-app observation steps per trajectory, about 2.1× Sol's mean of 45.8, while using only 9.9 edit/write steps, compared with 23.5 for Opus and 15.7 for Sonnet. This pattern is not due to shorter trajectories, as Astra averages 229.2 total steps, versus 209.3 for Opus and 199.4 for Sol.

Scaling with restoration depth. Figure 8 shows a growing mismatch between task burden and agent effort. From depth 1 to 8, code lines to restore grow 9.3× and browser actions across target traces grow 10.7×. Meanwhile, reference observations per target fall from 34.60 to 8.46, and current-app observations from 27.69 to 6.81, roughly 75% reductions. Edit/write effort per target falls by about 55%.

Deeper tasks also leave more implementation unrecovered. Unchanged target files rise from 12.4% to 23.3%, and retained mask-added stubs from 11.1% to 24.6%.

Shortcut attempts. We also examine requests for implementation information outside the intended repair interface, including Git history, public application source, protected harness or evaluator files, and direct reference access. At least one such request occurs in 7.2% of trajectories, ranging from 19.0% for Grok and 17.7% for Gemini 3.1 Pro to 0.3% for Astra. We block all of these channels by removing pre-mask Git history, disabling public egress, protecting grading files, and limiting reference access to the browser helper, so the rates reflect attempted access rather than successful recovery (Appendix I.6).

4.3 Full-Application Reconstruction

For full-application reconstruction, we evaluate Astra, Opus 5, and Sol at max reasoning effort on a 12-application subset of the 26-app corpus. Each agent starts from a minimal executable scaffold with a product-level capability description and access to a live reference. Evaluation uses the mined behaviors from these applications, comprising 590 atomic tests and 413 cumulative workflow tests.

Table 1 shows that Astra achieves the highest recovery rates, with 58.98% atomic and 49.15% cumulative binary score. Opus reaches 42.03% and 28.81%, followed by Sol at 33.39% and 21.07%. This ordering is consistent with partial reconstruction, suggesting that the relative performance differences persist when agents rebuild the application from a minimal scaffold. Cumulative recovery

Table 1: **Full-application reconstruction performance (%).** Atomic recovery tests one target behavior with its prerequisites. Cumulative binary recovery requires the complete workflow. Appendix I.2 reports cumulative chain scores.

Model	Reactive Resume	AWS Console	Baserow	BudgetWise	CareerLink	MailHub	Jira (Oldboyxx)	Trello (Knowankit)	StreamView	TeamChat	Trello (Vdevired)	Weights & Biases	Mean
Atomic behavior recovery													
GPT-6 Astra (max)	**50.0**	**56.0**	**14.0**	**60.0**	**62.0**	88.0	64.0	53.3	**82.0**	**68.0**	**28.9**	**78.0**	**58.98**
Opus 5 (max)	20.0	16.0	10.0	48.0	42.0	**92.0**	**82.0**	**55.6**	32.0	58.0	6.7	40.0	42.03
GPT-5.6 Sol (max)	8.0	8.0	8.0	30.0	18.0	78.0	72.0	**55.6**	32.0	50.0	26.7	16.0	33.39
Cumulative binary score													
GPT-6 Astra (max)	**43.8**	**45.8**	**5.9**	**42.6**	**55.6**	81.8	75.0	38.1	**75.9**	**45.5**	**23.5**	**72.7**	**49.15**
Opus 5 (max)	17.2	25.0	2.9	21.3	27.8	**86.4**	**86.1**	38.1	13.8	33.3	0.0	24.2	28.81
GPT-5.6 Sol (max)	4.7	0.0	2.9	12.8	8.3	59.1	75.0	**47.6**	24.1	27.3	20.6	3.0	21.07

is lower than atomic recovery for every model, indicating that recovering individual behaviors does not ensure recovery of complete workflows. Performance also varies substantially across applications. Astra's cumulative recovery ranges from 81.8% on `MailHub` to 5.9% on `Baserow`.

We hypothesize that this variation stems from differences in interaction structure. `MailHub` is largely organized around recurring message and thread operations, whereas lower-scoring applications involve more heterogeneous representations, including user-defined table schemas in `Baserow`, resume layouts and JSON import/export in `Reactive Resume`, resource configurations in `AWS Console`, and plan and SIM selections across cart and checkout in `BudgetWise`. Maintaining consistency across them can make full-application reconstruction more difficult. Full-reconstruction trajectories are also much longer, averaging about 700 agent steps and reaching 1,921 (Appendix I.4).

4.3.1 Failure Analysis

We ask why agents fail to recover behaviors that are available to inspect in a working reference. Using GPT-6 Astra, we classify the 977 failed atomic behaviors across the 36 reconstruction runs in Table 1 from the submitted source and the recorded reference and current observations into one of four categories: not observed, observed but not implemented, implemented with the wrong observable form, or implemented with the wrong state, route, or result. We also examine why the agent's own validation did not expose and correct the remaining difference.

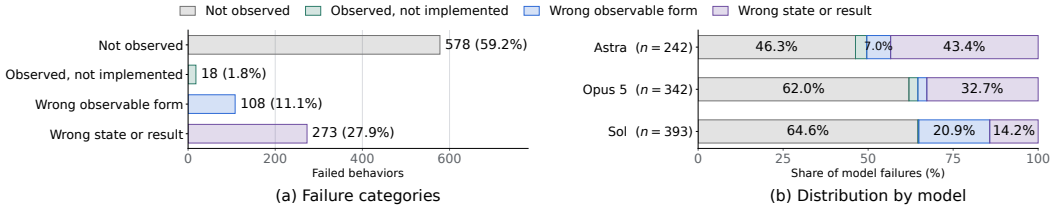
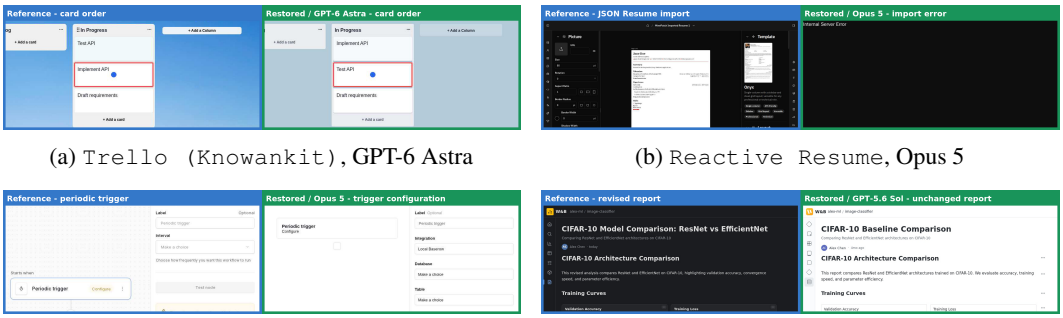


Figure 9: **Full-application reconstruction failures.** Across 36 runs on 12 applications, **(a)** counts failures in the four categories and **(b)** shows their distribution within each model's failures.

Behavior-level reconstruction outcomes. As shown in Figure 9, the largest failure category is behavior that was not observed in the reference, accounting for 59.2% of failures. The remaining failures occur after the relevant behavior has been observed. Of these, 1.8% are observed but not implemented, 11.1% are implemented with the wrong observable form, and 27.9% produce the wrong state, route, or result. These outcomes reveal difficulties both in exploring the reference and in faithfully reproducing behaviors that have already been observed.

What validation missed. Agents checked whether their reconstructed applications worked, but these checks did not always verify the exact behaviors observed in the reference. In the failures shown in Figure 10, agents often validated related functionality or only a subset of workflows, without retesting the behavior affected by their final code changes. Checks such as syntax validation, successful backend requests, or selected text matches establish that an application runs, not that it matches the reference. Partial reconstruction shows a similar gap, with current-app observations per target falling as depth increases (Figure 8).

Example failure cases. Figure 10 shows four reconstruction errors that remain in the final submissions despite agent validation. For each case, we replay the same action sequence on the reference and the final submission to expose the behavioral mismatch.



(a) `Trello (Knowankit)`, GPT-6 Astra

(b) `Reactive Resume`, Opus 5

(c) `Baserow`, Opus 5

(d) W&B, GPT-5.6 Sol

Figure 10: **Reconstruction errors missed during validation.** In each panel, the reference application appears on the left and the final submission on the right. **(a)** Dragging a card leaves the cards in the wrong order. **(b)** Importing a JSON Resume produces a rendering failure. **(c)** A periodic trigger shows database settings instead of an interval setting. **(d)** Changes to a report's title and summary are not saved.

In `Trello (Knowankit)`, dropping a card in the center leaves Astra's cards in the opposite order, and the agent revalidates other board interactions instead of the drag workflow. In `Reactive Resume`, Opus 5 imports JSON Resume data without converting it to the internal format, so rendering fails, while its validation covers only native resumes. In `Baserow`, Opus 5 renames a database trigger to `Periodic trigger` but keeps its `Database` and `Table` fields instead of the expected `Interval` setting, and never retests the periodic trigger. In W&B, Sol's report editor does not persist title and summary edits, and its final checks cover only syntax.

5 Discussion

What distinguishes stronger agents? The strongest agent neither edits most nor receives the most feedback. Astra makes the fewest edits in partial reconstruction but the most observations, while in full reconstruction Opus 5, which often batches browser calls in one shell step, receives about five times more browser feedback per step than Astra (7.9 vs. 1.6) yet recovers fewer behaviors. With the failure analysis, this suggests that *targeted* observation of the behavior an edit affects matters more than feedback volume.

Limitations and future work. ProgramDistill focuses on settings that replay deterministically. Agents observe structured browser state rather than screenshots, and the benchmark includes only self-contained web applications, excluding external services, nondeterministic state, and desktop or mobile environments. Multimodal inputs could ground interaction visually, although visual fidelity would need evaluation beyond replay-based checks. Benchmark quality also depends on construction choices. Mining and crafting use GPT-5.6 Sol, and replay selectors and agent tooling affect which behaviors are verified and how performance is measured. Replay also uses fixed inputs from seeded states, so held-out input variation and adversarial audits of the verifier remain open. Future work can examine sensitivity to these choices, use trajectories for behavioral distillation, and use restoration depth as a training curriculum.

Conclusion. ProgramDistill turns working applications into specifications, tasks, and verifiers by recording their behaviors and replaying them on new implementations. Performance declines with restoration depth, and full reconstruction leaves substantial behavior unrecovered, with most failures involving behavior never observed in the reference or workflows not retested after the relevant edit. More broadly, working software can serve as a scalable source of specifications, verifiers, tasks, and training environments.

AI USE STATEMENT

Generative AI is used as part of the research methodology described in this paper, including GPT-5.6 Sol for benchmark construction and GPT-6 Astra for failure categorization. Separately, generative AI tools were used to polish author-written text, assist with additional literature discovery for the related-work discussion, and help create the illustration in Figure 1. All AI-assisted manuscript edits were reviewed by the authors. Papers identified through AI-assisted literature search were individually examined and verified before inclusion, and the generated figure was likewise reviewed and revised as needed.

REPRODUCIBILITY STATEMENT

We will publicly release the ProgramDistill benchmark and the evaluation harness used in our experiments to facilitate reproducibility and further research. The application corpus and pinned source revisions are documented in Appendix B, with deterministic execution described in Appendix B.2. Mining and crafting prompts are provided in Appendices E and F, while agent instructions and execution settings are documented in Appendices G and D.

ETHICS STATEMENT

Reference-guided software engineering has legitimate uses in software maintenance, migration, testing, and reconstruction, but the same capability could potentially be misused to reproduce proprietary software without authorization. ProgramDistill is therefore intended for research and authorized software development.

Our experiments use only self-contained benchmark applications drawn from the OSWorld suite and public repositories. Agents run in isolated environments with public network access disabled, while hidden implementation and grading artifacts remain inaccessible through the intended interface. These safeguards limit opportunities to recover implementation details through unintended channels, but do not eliminate the broader misuse risks of the methodology. Use of ProgramDistill on third-party software should respect applicable licenses, access permissions, terms of use, and intellectual property rights.

REFERENCES

Tom Adamczewski, David Owen, David Rein, Florian Brand, Giles Edkins, Allen Hart, and Daniel O'Connell. Mirrorcode: Ai can rebuild entire programs from behavior alone. *arXiv preprint arXiv:2606.30182*, 2026.

Anthropic. Claude opus 5 system card, 2026. URL https://www.anthropic.com/claude-opus-5-system-card.

Anthropic. Claude sonnet 5 system card, 2026a. URL https://www.anthropic.com/claude-sonnet-5-system-card.

Anthropic. Browser use tool. Claude Platform documentation, 2026b. URL https://platform.claude.com/docs/en/agents-and-tools/tool-use/browser-use-tool.

Benyamin Beyzaei, Saghar Talebipour, Ghazal Rafiei, Nenad Medvidović, and Sam Malek. Automated test transfer across android apps using large language models. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):2227–2250, 2025.

Thibault Le Sellier de Chezelles, Maxime Gasse, Alexandre Lacoste, Massimo Caccia, Alexandre Drouin, Léo Boisvert, Megh Thakkar, Tom Marty, Rim Assouel, Sahar Omidi Shayegan, Lawrence Keunho Jang, Xing Han Lù, Ori Yoran, Dehan Kong, Frank F. Xu, Siva Reddy, Graham Neubig, Quentin Cappart, Russ Salakhutdinov, and Nicolas Chapados. The browsergym ecosystem for web agent research. *Transactions on Machine Learning Research*, 2025. ISSN 2835-8856. URL https://openreview.net/forum?id=5298fKGmv3. Expert Certification.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems*, 36:28091–28114, 2023.

Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. Reran: Timing-and touch-sensitive record and replay for android. In *2013 35th international conference on software engineering (ICSE)*, pp. 72–81. IEEE, 2013.

Google DeepMind. Gemini 3.1 pro model card, 2026a. URL https://deepmind.google/models/model-cards/gemini-3-1-pro.

Google DeepMind. Gemini 3.6 flash model card, 2026b. URL https://deepmind.google/models/model-cards/gemini-3-6-flash.

Google DeepMind. Gemini 3.7 flash model card, 2026c. URL https://deepmind.google/models/model-cards/gemini-3-7-flash.

JunJia Guo, Yuhang Yao, Jingdi Chen, et al. Vista: An end-to-end benchmark for visual spec-to-web-app coding agents. *arXiv preprint arXiv:2605.26144*, 2026.

Mouna Hammoudi, Gregg Rothermel, and Paolo Tonella. Why do record/replay tests of web applications break? In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 180–190. IEEE, 2016.

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. Webvoyager: Building an end-to-end web agent with large multimodal models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6864–6890, 2024.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. Deepswe: Measuring frontier coding agents on original, long-horizon engineering tasks. *arXiv preprint arXiv:2607.07946*, 2026.

Naman Jain, Jaskirat Singh, Manish Shetty, Tianjun Zhang, Liang Zheng, Koushik Sen, and Ion Stoica. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights SWE agents. In *NeurIPS 2025 Fourth Workshop on Deep Learning for Code*, 2025. URL https://openreview.net/forum?id=bbxU1MXEb9.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024.

Jeonghye Kim, Xufang Luo, Minbeom Kim, Sangmook Lee, Dohyung Kim, Jiwon Jeon, Dongsheng Li, and Yuqing Yang. Why does self-distillation (sometimes) degrade the reasoning capability of LLMs? In *Third Conference on Language Modeling*, 2026. URL https://openreview.net/forum?id=Az7guis46K.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Russ Salakhutdinov, and Daniel Fried. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 881–905, 2024.

Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pp. 272–281. IEEE, 2013.

Mike Merrill, Alexander Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations*, volume 2026, pp. 40903–40986, 2026.

Microsoft. Playwright–fast and reliable end-to-end testing for modern web apps. *https://playwright.dev/*, 2024.

Microsoft Corporation. Playwright MCP. GitHub, 2026. URL https://github.com/microsoft/playwright-mcp.

Facundo Molina, Alessandra Gorla, and Marcelo d'Amorim. Test oracle automation in the era of llms. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–24, 2025.

Magnus Müller and Gregor Žunič. Browser use: Enable ai to control your browser, 2024. URL https://github.com/browser-use/browser-use.

OpenAI. GPT-5.3-Codex system card, 2026a. URL https://openai.com/index/gpt-5-3-codex-system-card.

OpenAI. Gpt-5.6 sol: Frontier intelligence that scales with your ambition, 2026b. URL https://openai.com/index/gpt-5-6/.

OpenAI. Gpt-6 astra system card. September 2026. URL https://deploymentsafety.openai.com/gpt-6-astra/gpt-6-astra.pdf.

Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. Specrover: Code intent extraction via llms. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 963–974. IEEE, 2025.

Chenglei Si, Yanzhe Zhang, Ryan Li, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. Design2code: Benchmarking multimodal code generation for automated front-end engineering. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 3956–3974, 2025.

Atharv Sonwane, Isadora White, Hyunji Lee, Matheus Pereira, Lucas Caccia, Minseon Kim, Zhengyan Shi, Chinmay Singh, Alessandro Sordoni, Marc-Alexandre Côté, et al. Bugpilot: Complex bug generation for efficient learning of swe skills. *arXiv preprint arXiv:2510.19898*, 2025.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025.

Yifan Wu, Zhuokai Zhao, Songlin Li, Ho Hin Lee, Jiacheng Zhu, Shirley Wu, Tianhe Yu, Serena Li, Lizhu Zhang, Xiangjun Fan, et al. Swe-together: Evaluating coding agents in interactive user sessions. *arXiv preprint arXiv:2606.29957*, 2026.

xAI. Model card: Grok 4.6, 2026. URL https://media.x.ai/v1/website/card-4p6-4cd2dc57.pdf.

Jingyu Xiao, Yuxuan Wan, Yintong Huo, Zixin Wang, Xinyi Xu, Wenxuan Wang, Zhiyao Xu, Yuhang Wang, and Michael R Lyu. Interaction2code: Benchmarking mllm-based interactive webpage code generation from interactive prototyping. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 241–253. IEEE, 2025.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh J Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *Advances in Neural Information Processing Systems*, 37:52040–52094, 2024.

Zhipeng Xu, Jiahao Lu, Yining Zheng, Yuxin Wang, and Xipeng Qiu. Swe-bench science: Can coding agents resolve engineering tasks in science? *arXiv preprint arXiv:2608.19799*, 2026.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

John Yang, Carlos E Jimenez, Alex L Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R Narasimhan, Diyi Yang, Sida Wang, and Ofir Press. SWE-bench multimodal: Do AI systems generalize to visual software domains? In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=riTiq3i21b.

John Yang, Kilian Lieret, Carlos Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. *Advances in Neural Information Processing Systems*, 38, 2026a.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, et al. Programbench: Can language models rebuild programs from scratch? *arXiv preprint arXiv:2605.03546*, 2026b.

Sukmin Yun, Haokun Lin, Rusiru Thushara, Mohammad Q Bhat, Yongxin Wang, Zutao Jiang, Mingkai Deng, Jinhong Wang, Tianhua Tao, Junbo Li, et al. Web2code: A large-scale webpage-to-code dataset and evaluation framework for multimodal llms. *Advances in neural information processing systems*, 37:112134–112157, 2024.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, et al. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pp. 15585–15606, 2024.

APPENDIX CONTENTS

A Extended Related Work **15**

B Application Corpus **18**
B.1 Mined Lineage Structure . . . 18
B.2 Deterministic Execution . . . 19

C Browser Helper Implementation Details **20**
C.1 Action Vocabulary . . . 20
C.2 Replay-Stable Element Addressing . . . 20
C.3 Settled Observations . . . 21

D Agent Harness and Execution Settings **22**
D.1 Runtime Architecture and Isolation . . . 22
D.2 Observations and Execution Settings . . . 23

E Mining Agent Prompts and Trace Representation **25**
E.1 Mining-Agent Orchestration . . . 25
E.2 Planner . . . 25
E.3 Collector . . . 27
E.4 Relabeler . . . 28
E.5 Round Reflector . . . 29
E.6 Admitted Trace Representation . . . 29
E.7 Trace Evolution Across Mining Stages . . . 30

F Crafting Agent Prompts **31**
F.1 Task Generator . . . 31
F.2 Mask-Depth Critic . . . 32
F.3 Cumulative Mask Merger . . . 32

G Partial- and Full-Application Reconstruction Instructions **34**
G.1 Partial-Application Reconstruction . . . 34
G.2 Full-Application Reconstruction . . . 36

H Model Pricing and Execution Time **39**
H.1 Model Pricing . . . 39
H.2 Execution Time . . . 39

I More Experimental Results **41**
I.1 Results by Mask Scope . . . 41
I.2 Results Under the Chain Score . . . 41
I.3 Does Context Length Explain the Depth Effect? . . . 42
I.4 Agent Interaction Statistics . . . 43
I.5 Example of Successful Repair at Depth 8 . . . 44
I.6 Analysis of Shortcut Attempts . . . 44

A Extended Related Work

Section 2 compares prior benchmarks by the target they measure, their specification, and their verifier. Table 2 extends this comparison with the task unit and the way tasks compose, and the paragraphs below discuss each line of work.

Table 2: **Comparison with related benchmarks and task generators.** ProgramDistill is the only setting in which the unit being restored, the unit being graded, and the unit of composition are the same mined behavior.

Benchmark	Target measured	Specification	Verifier	Task unit	Composition
SWE-bench	Resolving real GitHub issues	Issue text	Tests from the merged fix	Issue	None
SWE-smith	Resolving synthesized bugs (training data)	LM-generated issue text	Existing repository tests that the bug breaks	Synthesized bug	Bugs combined within a file or module
Terminal-Bench	Completing hard terminal tasks	Task instruction	Human-written tests per task	Task	None
Design2Code, Interaction2Code	Generating a web page from a design or interaction prototype	Screenshots or interaction prototypes	Similarity to the reference page	Page	None
WebArena	Completing tasks on existing websites	Natural-language intent	Functional checks of the outcome	Task	None
ProgramBench	Rebuilding a program from its executable	Reference executable and usage documentation	Hidden behavioral tests generated by probing the reference, over stdout, exit codes, and files	Entire program	None
ProgramDistill	Discovering and restoring behavior from a live web application	Running reference application	Replay of the mined workflow over UI, URL, element, and persisted state	Mined behavior	Prerequisite lineage

Repository-level and terminal benchmarks. A large body of work evaluates coding agents on repository-level software engineering. SWE-bench (Jimenez et al., 2024) asks agents to resolve real GitHub issues, and open agent platforms such as OpenHands (Wang et al., 2025) standardize how such agents are built and run. DeepSWE (Huang et al., 2026) pushes the same setting toward long-horizon engineering work, with original tasks graded by hand-written verifiers. SWE-bench Multimodal (Yang et al., 2025) augments issue descriptions with visual evidence, and SWE-Together (Wu et al., 2026) extends this setting to multi-turn sessions in which users progressively clarify or correct requirements. Terminal-Bench (Merrill et al., 2026) evaluates agents on hard, realistic tasks in containerized terminal environments, each verified by human-written tests. These benchmarks increasingly capture realistic repository editing and execution, but the desired behavior is still supplied through an issue, test-derived task, visual artifact, or user interaction. ProgramDistill instead requires the agent to recover the specification by interacting with a working application, and uses the application's own executable behaviors as both repair targets and replayable verifiers.

Task construction. Pipelines such as SWE-smith (Yang et al., 2026a) and R2E-Gym (Jain et al., 2025) scale task construction by starting from code. SWE-smith injects bugs into a repository and keeps those that its existing tests catch, and R2E-Gym builds tasks from commits, collecting or generating fail-to-pass tests and back-translating problem statements. In SWE-smith, the verifier is the test suite written by the repository's developers, which first filters candidate bugs and then grades solutions, and the task unit is a code change. ProgramDistill reverses this order. It starts from a behavior discovered in the running application, masks the code responsible for it, and uses the recorded trace to both define and grade the task. Both approaches validate tasks with fail-to-pass checks, and ProgramDistill additionally requires prerequisite behaviors to keep passing, but they differ in where tasks begin, where verifiers come from, and how tasks compose. SWE-smith combines bugs within the same file or module, whereas ProgramDistill composes behaviors along prerequisite lineages, which yields tasks of controlled restoration depth and grading at the level of individual behaviors.

Visual and interactive web development. Web2Code (Yun et al., 2024) and Design2Code (Si et al., 2025) study reconstructing webpages from visual references, while Interaction2Code (Xiao

et al., 2025) extends this setting to interactive behavior. VISTA (Guo et al., 2026) further broadens the scope toward full web applications built from text, screenshots, or design files, and evaluates them with browser tests and visual similarity. ProgramDistill differs in that the target behavior is not provided as a fixed artifact. The agent must decide what to inspect, exercise a live reference to uncover stateful behavior, relate those observations to an existing codebase, implement the missing functionality, and validate the result against the running application. The distinction is thus a shift from *specification consumption* to *specification discovery*.

Behavioral reconstruction and reverse engineering. Recovering an unstated specification has been studied from the source side, where SpecRover (Ruan et al., 2025) infers program intent to guide repair, and from the verification side, where oracle automation (Molina et al., 2025) derives the assertions a test needs. In ProgramDistill both come from the running application itself, since a mined trace is at once the intent to restore and the oracle that checks it. ProgramBench (Yang et al., 2026b) and MirrorCode (Adamczewski et al., 2026) are the closest precedent, treating executable software as a behavioral specification that agents probe without its source. ProgramDistill differs from them in three ways. First, their task is an entire program. ProgramBench, for instance, grades a reimplementation with a separately generated suite of black-box tests, each of which runs the executable on given inputs and checks outputs such as stdout, exit codes, and files. In ProgramDistill, the unit being graded is the unit being restored, since each mined behavior is both what the agent repairs and what the verifier replays. Second, those tests check inputs and outputs independently, whereas web behaviors accumulate state and depend on one another, so a behavior can be checked only after its prerequisites have run. ProgramDistill turns this dependency structure into a difficulty axis, composing behaviors along prerequisite lineages into tasks of controlled restoration depth, with partial credit and per-behavior diagnosis. Third, ProgramDistill spans more settings under one verifier. Full reconstruction corresponds to the whole-program setting of ProgramBench, while atomic and cumulative repair evaluate localized changes within an existing codebase.

Record-and-replay GUI testing. Record-and-replay is a long-standing technique for GUI testing, in which a tester's interactions are recorded and later replayed as automated tests. Web tools such as Selenium IDE follow this approach, and empirical studies show that recorded tests are cheap to create but costly to maintain as applications evolve (Leotta et al., 2013), most often because the locators that identify page elements break (Hammoudi et al., 2016). On mobile platforms, RERAN replays low-level touch and sensor events with precise timing (Gomez et al., 2013). More recent work transfers recorded tests across different apps with similar functionality using LLMs (Beyzaei et al., 2025). ProgramDistill departs from this line in three ways. First, record-and-replay usually tests the application it was recorded on, whereas ProgramDistill uses each recording to construct a benchmark task, by masking the code behind it, and to grade repairs of a different implementation. Second, its traces are recorded by agents rather than by testers, and they address elements by accessible roles and names instead of generated DOM identifiers, so they do not break when those identifiers change across builds. Third, unlike test migration, which relaxes matching to fit another app's interface, its replay remains strict, because reproducing the reference's observable interface is the goal.

Browser-use agents and environments. A broad line of work studies agents that complete user-specified tasks through interaction with existing websites, including WebArena (Zhou et al., 2024), VisualWebArena (Koh et al., 2024), WorkArena (de Chezelles et al., 2025), Mind2Web (Deng et al., 2023), WebVoyager (He et al., 2024), and BU Bench (Müller & Žunič, 2024). OSWorld (Xie et al., 2024) evaluates agents on computer-use tasks involving existing web and desktop applications. A complementary line of work develops browser interfaces and execution infrastructure for such agents. BrowserGym (de Chezelles et al., 2025) provides a unified environment with common observation and action spaces across web-agent benchmarks. Browser Use (Müller & Žunič, 2024) integrates page observations and browser-control tools into an agent framework. Anthropic's browser use tool (Anthropic, 2026b) provides tools for reading page structure and interacting with referenced elements. Playwright MCP (Microsoft Corporation, 2026) exposes Playwright browser automation through accessibility snapshots and element-level actions.

ProgramDistill similarly couples coding with iterative browser-based observation and execution, but the code serves a different purpose. Rather than producing an automation program that operates a website, the agent modifies the application itself to recover behavior observed from a working

reference. It must infer intended behavior through interaction, repair or reconstruct the source implementation, and validate the result against the reference.

Taken together, prior benchmarks isolate important parts of the software development process. ProgramDistill brings them into a single closed-loop setting in which an agent must *observe, infer, implement, and validate* behavior against a live reference. This closely mirrors reference-guided web development, where a developer studies an existing product or prototype, modifies the implementation, and repeatedly checks the result.

B Application Corpus

The corpus contains 26 self-contained interactive web applications drawn from two sources. Eighteen are adapted from the OSWorld web-application suite (Xie et al., 2024), which provides a controlled and diverse interaction substrate. The remaining eight are real-world open-source projects and SaaS clones taken from public repositories. These contribute larger codebases, deeper workflows, nontrivial state, and heterogeneous architectures, spanning Django, Angular, React, and Next.js stacks with SQL and document back ends.

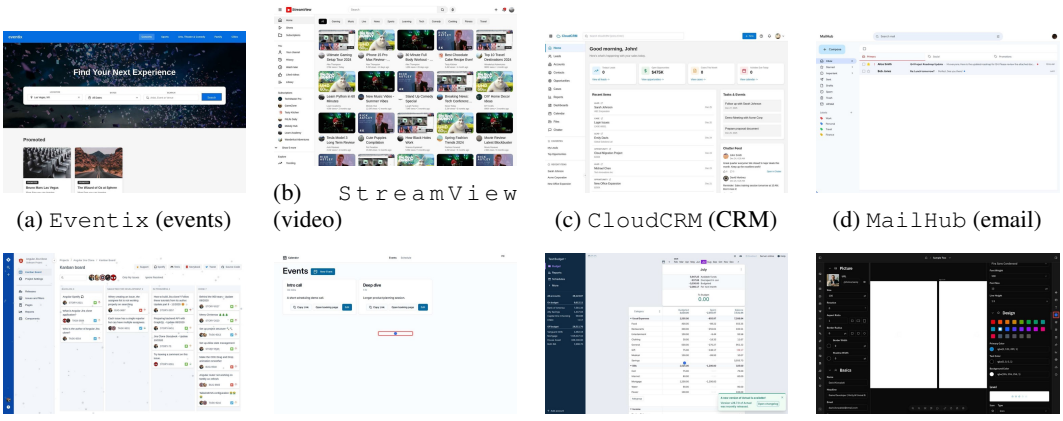
Each application is pinned to a single upstream commit. Its runtime image is rebuilt from that commit and checked to reproduce the vendored source tree, so mining, masking, repair, and replay all run against a byte-identical application. Table 3 lists each application with its source repository, pinned commit, and numbers of atomic and cumulative tasks. The number of mined tasks varies across applications, reflecting differences in how many behaviors can be reliably discovered and replay-verified under each application's state and interaction dynamics. Figure 11 shows representative interfaces from both sources.

Table 3: **The 26 applications in the ProgramDistill corpus.** Task counts are the atomic and cumulative partial-application reconstruction tasks mined from each application.

Application	Source repository	Commit	Atomic	Cum.	Total
OSWorld-derived applications					
AWS Console	`Task-Web/awsconsole_web`	`794c1cc1`	155	42	197
BaseSite	`Task-Web/basesite`	`4c06e3d7`	147	70	217
BudgetWise	`Task-Web/budgetwise_web`	`bf3903f6`	198	92	290
Calendar	`Task-Web/calendar_web`	`6a7a8d1d`	136	45	181
CareerLink	`Task-Web/careerlink_web`	`33352a1c`	197	70	267
CloudCRM	`Task-Web/cloudcrm_web`	`1c7a1653`	53	3	56
Eventix	`Task-Web/eventix_web`	`f60c3463`	20	3	23
ExpenseFlow	`Task-Web/expenseflow_web`	`487fd168`	2	0	2
FormCraft	`Task-Web/formcraft_web`	`5c20ecdf`	25	4	29
Insurance	`Task-Web/insurance_claim_web`	`720120c7`	92	49	141
MailHub	`Task-Web/mailhub_web`	`4603ec38`	198	49	247
ReviewSphere	`Task-Web/reviewsphere_web`	`8e4f9706`	69	21	90
StreamView	`Task-Web/streamview_web`	`ea965045`	105	53	158
TeamChat	`Task-Web/teamchat_web`	`ed077b4d`	161	68	229
TravelHub	`Task-Web/travelhub_ad_web`	`b45dae8e`	34	11	45
VaultBank	`Task-Web/vaultbank_web`	`e9ebb7e4`	131	33	164
Visa Application	`Task-Web/visaapplication_web`	`c0e14ed0`	52	42	94
Weights & Biases	`Task-Web/wandb_web`	`1c781220`	126	52	178
Real-world open-source projects and SaaS clones					
Actual Budget	`actualbudget/actual`	`e0880f39`	36	18	54
Reactive Resume	`amruthpillai/reactive-resume`	`dcf1b28c`	246	126	372
Baserow	`baserow/baserow`	`8d7c0111`	207	105	312
Trello (Knowankit)	`knowankit/trello-clone`	`19e9d64a`	93	46	139
Jira (Oldboyxx)	`oldboyxx/jira_clone`	`26a9e77b`	122	62	184
Jira (Trungvose)	`trungvose/jira-clone-angular`	`20b50e9a`	24	2	26
Trello (Vdevired)	`vdevired/trello-clone`	`1a40f9e5`	86	60	146
Calendly	`WebDevSimplified/calendly-clone`	`730c5f19`	147	75	222
Total			**2862**	**1201**	**4063**

B.1 Mined Lineage Structure

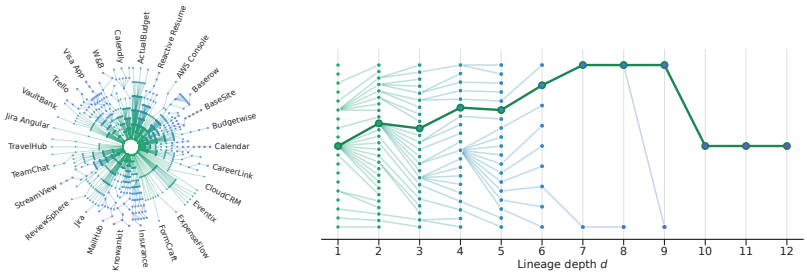
Figure 12 visualizes the mined prerequisite structure at two scales, showing a corpus-wide view across all 26 applications and a representative `StreamView` lineage reaching depth 12.



(a) `Eventix` (events) (b) `StreamView` (video) (c) `CloudCRM` (CRM) (d) `MailHub` (email)

(e) `Jiraclone` (kanban) (f) `Calendly` (scheduling) (g) `ActualBudget` (finance) (h) `ReactiveResume` (resume)

Figure 11: Example applications in the ProgramDistill corpus. Top row (a–d) shows OSWorld-derived apps. Bottom row (e–h) shows real-world open-source clones.



(a) Task lineage forest across 26 applications (b) Representative task lineage: StreamView, depth 12

Figure 12: **Prerequisite lineages across applications.** (a) Mined lineage forests across all 26 applications. (b) A representative `StreamView` lineage reaching depth 12, with one root-to-leaf prerequisite chain highlighted.

B.2 Deterministic Execution

Reliable replay requires each trace to begin from the same application state and the same application time. Our `mine-craft-patch` pipeline resets the application before every collection and replay so that each trace starts from the same seeded state. To control time-dependent behavior, each trace also records the application time at which it was collected. On replay, the application clock is shifted so that execution starts from the same time while continuing to advance normally.

The system computes a constant offset between the recorded application time and the host clock. This offset is applied through `libfaketime` to the application server, interpreter, and database. The browser helper applies the same offset to the page's `Date` implementation, keeping the frontend and backend synchronized to the same application clock.

Because the offset remains constant, time continues to advance at the normal rate during execution. This preserves relative ordering, durations, and timeout semantics while keeping behavior that depends on the current date or time consistent across replays. For example, a booking interface that renders available dates relative to "today" presents the same dates during collection and subsequent replay.

C BROWSER HELPER IMPLEMENTATION DETAILS

ProgramDistill uses a shared browser helper across mining, replay verification, and repair evaluation. The helper executes actions in a real Chromium instance and returns structured observations of the resulting application state. Table 4 summarizes the browser actions supported by the helper. Reliable replay additionally requires two properties: recorded actions must refer to stable interaction targets, and observations must avoid transient intermediate states. We describe these mechanisms below. The solver-facing view of these observations is filtered as described in Appendix D.

C.1 ACTION VOCABULARY

All stages use the same underlying browser helper, although the actions exposed to each stage differ slightly. Common actions support observation, navigation, and interaction with visible elements. Repair additionally supports joint observation of the reference and editable applications, while mining exposes actions used for trace construction. Table 4 lists the complete action vocabulary.

Table 4: **Browser-helper action vocabulary.** Common actions are available during both mining and repair. Repair additionally exposes side-by-side observation of the reference and editable applications, while mining provides several actions used for trace construction.

Command	Arguments	Effect
Common actions		
`observe`	`page`	Return a settled observation containing the current URL, visible text, and interactable elements.
`click`	`page`, `elementId`	Click an element from the latest observation.
`type`	`page`, `elementId`, `text`	Enter text into an interactable field.
`hover`	`page`, `elementId`	Hover over an element.
`focus`	`page`, `elementId`|`selector`	Move browser focus to a target element.
`press`	`page`, `key`	Send a keyboard input to the page.
`scroll`	`page`, `direction`, [`elementId`]	Scroll the page, or a specified scrollable element, upward or downward.
`wait`	`page`, `ms`	Wait for a bounded interval and return the resulting settled observation.
`drag`	`page`, `elementId`+`targetElementId`	Drag one observed element to another and record stable selectors for both endpoints.
`back` / `forward`	`page`	Navigate backward or forward through browser history.
`upload`	`page`, `elementId`, file	Upload a file through an observed file input. During mining this may reuse an artifact produced by a prerequisite trace.
Repair-only actions		
`observe-both`	none	Observe the editable and reference applications together for behavioral comparison.
`reset-reference`	none	Restore the clean seeded reference state after exploratory mutations.
Mining-only actions		
`navigate`	`path`	Load a specified in-application path.
`waitForSelector`	`selector`	Wait until a specified element appears before continuing trace collection.

C.2 REPLAY-STABLE ELEMENT ADDRESSING

Frontend-generated DOM identifiers and class names can change across application builds, making them unreliable for replay. ProgramDistill therefore separates the temporary identifiers used during live interaction from the selectors stored in recorded traces.

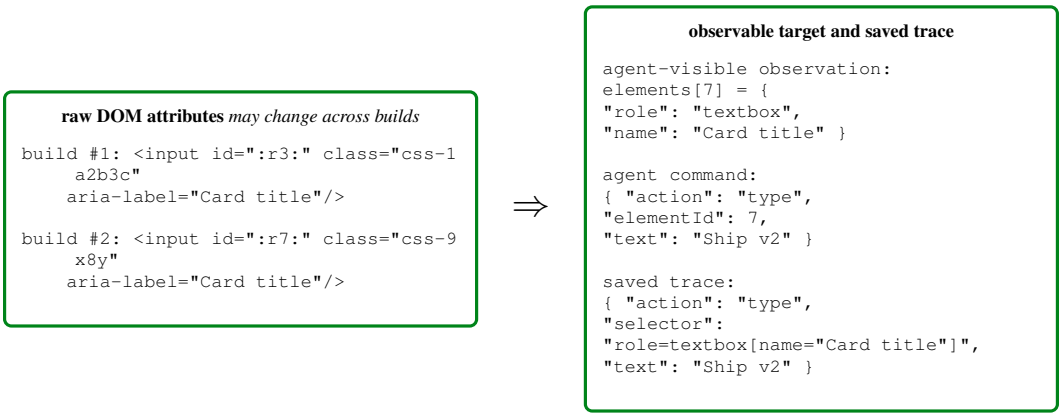


Figure 13: **Replay-stable element addressing.** Recorded actions use browser-observable descriptors rather than frontend-generated DOM identifiers, which may change across builds.

Each observation assigns a transient numeric `elementId` to every visible interactable element and exposes browser-observable attributes such as accessibility roles and names, labels, placeholders, visible text, links, and test IDs. For each element, the helper attempts to construct a selector from these attributes. Actions are issued using `elementId`, but the recorded trace stores the corresponding selector instead.

As illustrated in Figure 13, this allows recorded actions to locate the same logical element even when generated DOM attributes change across builds. If the observable attributes do not uniquely identify an element, no selector is produced and actions targeting that element are refused. Thus, every element-targeted action retained in a trace has a selector that can be resolved during replay.

C.3 Settled Observations

Browser actions can trigger asynchronous updates, so recording state immediately after an action may capture a transient intermediate state. ProgramDistill therefore waits for browser activity to settle before recording each observation.

After an action, the helper applies a short delay and monitors both network activity and DOM mutations. By default, an observation is considered settled when no tracked requests remain in flight and the DOM has remained unchanged for 400 ms. The procedure is capped at 10 s to avoid blocking indefinitely on applications with continuous updates. DOM mutations include changes to elements, attributes, and text, allowing the helper to detect updates caused by timers, WebSockets, and server-sent events.

Once settled, the helper records the current URL, interactable elements, application-state signals, observed API statuses, and an interaction screenshot. Figure 14 illustrates how this procedure avoids recording transient states. Traces that fail to reach a settled observation within the configured limit are treated as non-reproducible and excluded from the verified trace bank.

Screenshots are retained for analysis but are not used uniformly as model inputs. During mining, the Relabeler receives an animated GIF of the cleaned trace together with the executed actions and outcome evidence. The repair and full-application reconstruction agents, in contrast, receive only structured browser observations. Replay verification likewise uses recorded actions and expected-signal assertions rather than LLM-based visual judgment.

immediate snapshot


```
{ "url": "/boards/12",
  "apiRequests": [
    { "method": "PATCH",
      "path": "/api/cards/91",
      "status": null } ],
  "targets": [
    { "elementId": 7,
      "role": "textbox",
      "text": "Card title",
      "value": "" } ],
  "appStateText": [] }
```


⇒

settled observation


```
{ "url": "/boards/12",
  "apiRequests": [
    { "method": "PATCH",
      "path": "/api/cards/91",
      "status": 200 } ],
  "targets": [
    { "elementId": 7,
      "role": "textbox",
      "text": "Card title",
      "value": "Ship v2 release" } ],
  "appStateText": ["Ship v2 release"] }
```


Figure 14: **Settled observations.** Observations are recorded after bounded browser and network activity has stabilized, reducing the chance that traces encode transient intermediate states.

D Agent Harness and Execution Settings

The agent harness builds on R2E-Gym (Jain et al., 2025), retaining its coding-tool interface and adding live browser interaction through the `browser` CLI. This section describes runtime boundaries, permitted communication, observation handling, conversation-history management, and execution settings. Details of browser actions and element addressing are provided in Appendix C, while the instructions for partial-application and full-application reconstruction appear in Appendix G.

D.1 Runtime Architecture and Isolation

The harness separates the runtime into three components, the editable application, the reference-side browser service, and the verifier. Figure 15 illustrates their roles and the communication paths used during reference-guided implementation. The key design principle is that the agent may observe the behavior of the reference application without receiving its source code or a source checkout that could be copied directly.

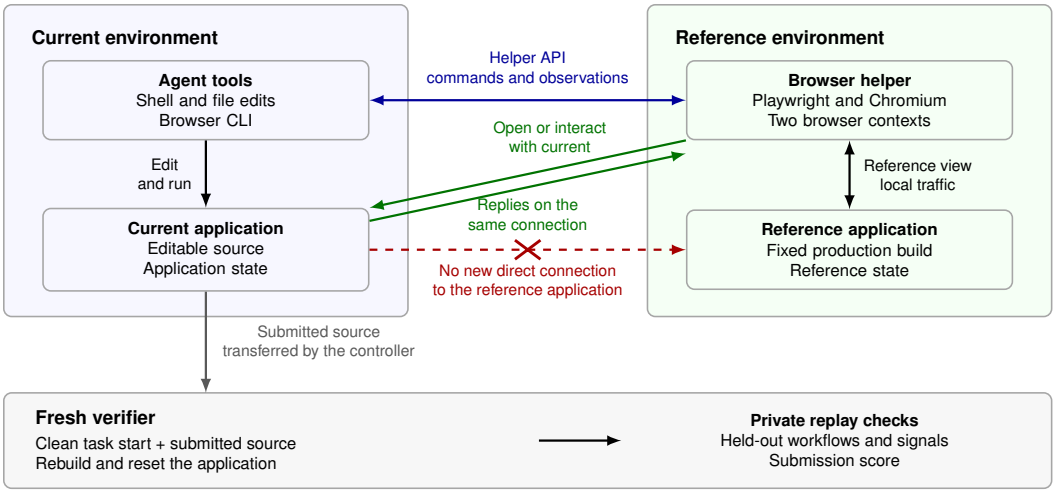


Figure 15: **Runtime layout and permitted communication.** The browser helper runs on the reference side and maintains isolated contexts for the fixed reference and editable current applications. The current context communicates with the editable application across the runtime boundary, while direct agent access to reference-application ports is blocked. Submitted source is evaluated separately in a fresh verifier. Arrows denote logical communication paths rather than a shared filesystem.

Editable current environment. The agent's shell and file tools operate only on the current application. In partial-application reconstruction tasks, the agent starts from a source tree in which selected implementation details are masked. In full-application reconstruction tasks, it starts from

a minimal executable scaffold. In both cases, the agent may edit the application source, run the resulting implementation, and inspect its behavior.

The agent's `browser` command does not launch a separate browser inside the editable environment. Instead, it acts as a thin client for the reference-side browser service, which performs browser interactions on the agent's behalf.

Reference environment and browser service. The reference environment contains the fixed production application and a Playwright-based browser helper. The helper maintains two isolated browser contexts, `reference` and `current`, with separate cookies, storage, and page state. The `reference` context opens the fixed production application locally, while the `current` context connects across the runtime boundary to the editable application. In the latter case, the browser acts only as a client of the current application. This browser connection does not expose the agent's source tree to the reference application or browser service.

The helper exposes browser interactions through a constrained API and returns structured observations to the agent. Browser recordings, temporary state, and other internal artifacts remain outside the submitted source tree.

Control traffic and application traffic. Two kinds of network traffic are involved. The agent sends *control traffic* to the browser helper on port 7788 to request browser actions and observations. Separately, the browser generates *application traffic* when its reference or current context connects to an application.

Access to the helper does not permit the agent to connect directly to reference-application ports. The current browser context, however, must receive responses from the editable application across the runtime boundary. The network filter therefore allows only reply packets that connection tracking associates with an existing browser connection. This preserves communication between the browser and current application without opening a broader network path from the agent to the reference application. The same policy is used for both evaluation settings.

Fresh verification and isolation scope. After implementation, a trusted controller transfers the submitted source to a separate verifier. The verifier starts from a clean task baseline, applies the submission, rebuilds the application, resets persistent state, and replays the evaluation workflows. It does not reuse the agent's interactive runtime state.

Private grading information, including grading traces, expected signals, and the gold patch, remains outside the agent-visible interface. Success therefore depends on whether the submitted source reproduces the required behavior under fresh verification, rather than on the state of the agent's final browser session.

These boundaries separate editable source code, observable reference behavior, and private grading material. They provide the isolation required for the intended reference-guided evaluation setting, but are not designed as a general-purpose sandbox for arbitrary hostile code.

D.2 Observations and Execution Settings

Agent-visible information. The agent receives text-based tool outputs. These include shell stdout and stderr, displayed file contents and editing feedback, search results, and structured browser observations containing the URL, visible text, element IDs, roles, accessible names, and input values. Full reconstruction additionally exposes selectors for observed targets. Screenshots, screenshot paths, raw HTML, API-request logs, and internal artifact paths are excluded from agent-facing responses.

Recoverable tool observations. The harness preserves solver-visible tool outputs separately from the model context under `/workspace/programdistill/observations/`, outside the submitted source directory. Outputs up to 20,000 characters are shown directly, while longer outputs are paginated into an initial 20,000-character segment followed by 15,000-character segments that can be read on demand. This lets agents revisit earlier observations even after the corresponding turns leave the model context, without repeating state-changing browser actions such as clicks or form submissions.

Conversation-history management. When the retained history exceeds the token limit, the harness removes older complete turns while preserving the task instructions, the first assistant turn, and recent context. Tool calls are always kept together with their corresponding results, preventing invalid partial histories. The resulting message set is checked against the model's token limit before each call.

Execution settings. Table 5 summarizes the harness configuration for partial-application and full-application reconstruction. The history cutoff determines when older turns are removed, while the token stop threshold limits the size of the active conversation rather than cumulative API token usage. None of our runs reached the token stop threshold. The maximum agent-step limit was reached only once, by Opus 5 in partial-application reconstruction.

Table 5: **Agent-harness execution settings.**

Setting	Partial	Full
Maximum agent steps	1,000	10,000
Retained-history cutoff (tokens)	150,000	500,000
Token stop threshold (tokens)	1,000,000	5,000,000
Reasoning effort	high	max
Per-tool timeout (seconds)	90	90
Aggregate solver time limit	None	None

E MINING AGENT PROMPTS AND TRACE REPRESENTATION

This appendix provides the mining prompts and trace representation. It presents the four mining-role prompts in execution order, the schema of an admitted trace, and the evolution of a trace across mining stages. The Collector handles both discovery and clean re-collection, with the latter using a guided prompt variant and additional context describing the established parent state or a previously successful route.

E.1 MINING-AGENT ORCHESTRATION

Mining runs in rounds over the current verified trace bank $\mathcal{B}^{(r)}$. Planners propose grounded goals and optional parent traces. Collectors discover successful behaviors and then re-collect them from a clean state, while the replay verifier rejects unstable traces. Relabelers describe the behavior actually achieved, and the Reflector summarizes coverage and failures to guide the next round. Accepted traces form $\mathcal{B}^{(r+1)}$. Figure 16 and Table 6 show this control flow. The main text uses the resulting verified traces and parent links as inputs to crafting.

Table 6: Specialized sub-agents within the mining agent.

Sub-agent	Role
Planner	Proposes behavior goals and route hints grounded in application evidence.
Collector	Pursues goals, records interactions, and selects observed success signals.
Relabeler	Assigns a name and description to the behavior actually achieved.
Reflector	Summarizes outcomes to guide later rounds.

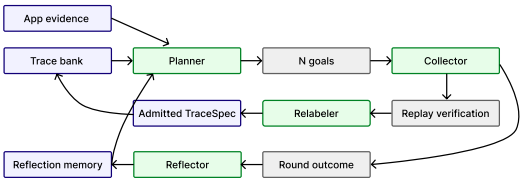


Figure 16: Mining-agent control flow. Each round proposes goals, collects and refines candidate traces, admits only replay-verified behaviors, and uses reflection to guide the next round.

E.2 PLANNER

At the beginning of each exploration round, the Planner receives the goal budget together with the current coverage, known failures, application capabilities, available parent traces, and reflections from prior rounds. It has read-only access to both the live application and the source code.

Planner system prompt

```
You are TraceDiscoveryPlanner for ProgramDistill. Use 'browser_action'
 (observe/click only) plus the read-only 'search' and 'file_editor' (
view only) tools to understand the running app and its source code,
then call 'propose_goals' with a DIVERSE set of grounded discovery
goals.

Evidence requirements for EACH goal:
- Ground the behavior and 'trace_hint' in source code you actually
searched/read and UI/routes you actually observed; do not invent
controls, transitions, or outcomes.
- Write 'trace_hint' as a concise natural-language route for the
downstream browser agent, not selectors or action JSON.
- The exact source files you actually searched/read in '
trace_hint_code_paths'.
- The exact UI text, accessible names, titles, or URL paths you
actually observed in 'trace_hint_observed_ui'. Unsupported evidence is
 rejected.

Requirements:
```

```
- Prefer uncovered CORE capabilities that directly accomplish the
product's purpose. Also reserve goals for meaningful SECONDARY or
source-discovered behaviors that add breadth or replace blocked,
redundant, or non-verifiable core outcomes. The capability map is
guidance, not an allowlist.
- Spread goals across distinct observed surfaces and varied workflow
depths: focused state transitions, medium workflows, and grounded end-
to-end workflows. Every goal must produce an independently valuable,
browser-verifiable product outcome. Do not propose goals whose only
novelty is opening a page, performing a routine UI gesture, toggling
transient UI, or inspecting debug or implementation state. Prioritize
state-changing, verifiable outcomes; do not pad goals. Split only at a
 persistent checkpoint that has standalone value and enables a deeper
follow-up.
- Deprioritize login errors, minor validation paths, generic
navigation, tiny gestures, and implementation details unless they are
necessary prerequisites for a meaningful deeper workflow.
- Propose behaviors that are semantically distinct from every trace-
ledger entry and from other goals in the same round. Treat workflows
as equivalent when they exercise the same product capability, state
transition, UI path, and verification pattern, even if they use
different recipients, records, names, text values, or other inputs. Do
 not propose parameter-only variants, such as sending the same kind of
 message to a different person or repeating the same operation on
another record, unless the changed entity causes materially different
product logic, UI, or outcome.
- Do not re-propose the same or an equivalent failed workflow when its
 required state or control was unavailable, unless current source/UI
evidence shows that the root cause has changed.
- Each goal must be independently attemptable in the browser either
from a cold start or after replaying its declared parent chain.
- Actively use the deepest relevant 'parent_eligible=true' trace when
its end-state enables the new behavior or avoids repeating meaningful
setup. The full chain replays automatically, so describe only the
novel behavior from the parent's end-state. Prefer extending that
state into a deeper or later outcome; distribute goals across relevant
 parents where useful.
- Never use the goal itself, a descendant, an unavailable trace, or an
 unrelated trace as parent. Use a root goal only for genuine cold-
start behavior that does not repeat available setup.
- After proposing goals, call 'finish'.
```

Each proposed goal has the following structure.

```
{
  "name": "...",
  "description": "...",
  "trace_hint": "...",
  "trace_hint_code_paths": ["..."],
  "trace_hint_observed_ui": ["..."],
  "novelty_reason": "...",
  "parent_trace": null
}
```

The purpose of the evidence fields is to make each proposed goal more faithful to the source and UI evidence available to the Planner. They record the source paths and interface values actually observed during that run, but are not otherwise used by the pipeline.

Each available parent-trace entry includes its ordered replay `chain`, `depth`, and boolean indicators `is_leaf` and `deepest_leaf`. The Planner also receives the current number of traces and leaf chains, along with the maximum tree depth, and is encouraged to extend a relevant leaf when possible.

E.3 COLLECTOR

Each goal is handled by a Collector that interacts with the live application using only live browser observations. As the Collector executes the goal, the environment records every browser action it takes. After each observation, the environment also provides a set of candidate success signals derived from the observed application state, such as changes in the URL, visible text, element state, or other observable outcomes. Once the Collector reaches a coherent outcome, it selects the signals that best demonstrate that the behavior was achieved. The recorded actions and selected signals together form the candidate trace, which is subsequently replay-verified.

Collector system prompt

```
You are TraceCollectorAgent for ProgramDistill. You receive one
discovery goal and an optional natural-language route hint. Attempt
the goal in the live web app through 'browser_action', recording the
browser actions you actually execute. The hint is guidance, not an
action script: work only from live observations. When a coherent
outcome is achieved, select observation-derived evidence that proves
it and save the trace.

Rules:
- Begin with 'browser_action' action=observe. Every browser action
returns a settled observation; use it to choose the next step. If it
conclusively proves success, use that response's 'signalCandidates',
call 'save_trace' immediately, and do not observe again. Call observe
separately only when the action response is genuinely inconclusive.
- Interact only with elements present in the latest observation, using
 its 'elementId' or an explicit selector grounded in an observed name,
 label, text, or test id.
- Use only credentials or input values supplied by the app or runtime.
 Never invent or brute-force them. If authentication blocks the goal
and no value is available, finish with a short reason.
- Pursue the assigned goal methodically and follow the instance
instructions about whether a partial but coherent outcome may be saved.
 Never claim the assigned goal succeeded when it did not.
- For export or download goals, treat a non-empty saved file in the
observation's 'downloads' array as success; a headless browser may show
 no separate download UI.
- Select one or more signal candidates that materially prove the
achieved outcome. When multiple meaningful signals are available,
prefer signals from different evidence types. Exclude weak, unrelated,
 or duplicative signals, and never include a signal merely to increase
 the count. Never author or modify signal content. The selected set
must collectively prove every material outcome you claim. A candidate
with 'source=form_value' proves only what is currently in an editable
control; it does not by itself prove that a save, update, filter, or
submission took effect. Likewise, a generic heading or label does not
by itself prove the claimed result. Prefer concrete rendered results or
 other outcome-specific evidence. After saving, call 'finish'.
- Stop immediately at the successful state. Do not log out, undo,
navigate away, or perform cleanup afterward, because every subsequent
browser action becomes part of the replay trace.
```

Prerequisite-state context. Some goals depend on application state established by an existing parent trace. Before the Collector begins, the system replays the validated parent trace and its ancestors to establish that prerequisite state. The Collector is then told to continue from the resulting state and record only the novel actions needed for the current goal.

Prerequisite-state block

```
PARENT STATE ALREADY ESTABLISHED: before this Collector started, the
system replayed the validated parent trace [<parent_trace>] and all of
its ancestors. The browser is already at the parent's end-state. Do NOT

execute the parent actions a second time; begin from the current state
and record ONLY the novel actions that achieve this goal. The parent
chain is replayed automatically whenever this child trace is replayed
and is not duplicated inside the child's action trace.
```

Guided clean re-collection. Initial discovery may produce long traces because the Collector explores multiple interaction paths before reaching the goal. Once a behavior has been successfully discovered, the earlier successful route is provided back to the Collector as guidance for clean re-collection. Starting again from a clean state, the Collector uses live observations to reach the same goal through a more direct path while omitting unnecessary exploratory steps. The following block provides the earlier successful route as a hint rather than as an action sequence to reproduce exactly.

Guided re-run block

```
GUIDED RE-RUN: this goal has ALREADY been confirmed reachable. Below
is the
earlier successful run's steps. Use them as route evidence, not as
instructions
to reproduce in order. From the CURRENT clean start, use live
observations to
take a direct route while preserving every step needed to reach and
prove the
assigned outcome. Omit only steps whose removal does not affect that
outcome.
'{initial_title_part_0}' in a route is symbolic: replace it with the
first
segment of the CURRENT page title from the live initial state, never
send the
braces literally.
Earlier successful route:
<discovered route>
```

E.4 RELABELER

Once a candidate passes replay verification, the Relabeler receives the executed actions, success evidence, and final observation, but not the original goal. It then assigns a name and description based only on the behavior actually achieved. Hiding the original goal prevents the label from being biased toward the Planner's intended outcome when the Collector reached something different.

Relabeler system prompt

```
You are TraceRelabeler for ProgramDistill. You are given the browser
actions an agent actually executed against a live web app, the machine-
checkable success evidence it observed, and the full animated GIF
rendered from every retained step in the cleaned replayable trace. Use
 the visual sequence to verify what was visibly accomplished and to
avoid labels contradicted by the UI. Author a concise identity for
THIS behavior, grounded ONLY in the actions and evidence shown. You
are deliberately NOT told what the agent originally intended -- do not
 guess or restate an intention; describe only what the actions and
evidence prove was accomplished.

Return STRICT JSON with exactly two keys and no other prose:
```

```
{"name": "<short lower_snake_case behavior id, e.g.
create_board_and_add_card>", "description": "<one plain sentence
stating what was accomplished>"}
Rules: name is lower_snake_case, specific, at most 6 words;
description is a single sentence describing the concrete achieved
behavior (the outcome, not raw UI mechanics). If the evidence shows
only navigation or a trivial read with no state change, name it
accordingly (e.g. view_settings_page).
```

E.5 Round Reflector

The Reflector runs after all goals in an exploration round have been processed. It reviews the application capabilities, the accumulated trace bank, the goals and outcomes from the completed round, unresolved failures, and reflections from earlier rounds. It then produces a short summary of the most important coverage gaps, failure patterns, and opportunities for deeper exploration. This reflection is appended to the `reflection_memory` provided to the Planner in the next round.

Round-reflection system prompt

```
You are TraceDiscoveryRoundReflector for ProgramDistill. Review one
completed discovery round using the app's curated product
specification, the cumulative trace bank, the round's proposed goals
and concrete outcomes, the recent proposal-to-actual outcome history,
prior failures, and recent round reflections. Produce a short advisory
 reflection for the NEXT Planner. Identify meaningful coverage bias,
neglected product surfaces, useful opportunities to go deeper through
existing parent traces, repeated failure patterns, and whether
discovery appears close to saturation. Explicitly flag recurring near-
duplicate proposals even when their names differ by comparing product
surface, state transition, parent state, and the actual relabeled
outcome. Distinguish useful depth from redundant repetition. Judge
coverage from actual outcomes rather than proposed names. Reassess
earlier concerns against current evidence and do not repeat concerns
that have already been resolved. Focus on the two to four most
consequential gaps instead of inventorying every missing capability.
Do not propose a numbered goal list, selectors, or action steps. Do
not claim a surface exists unless supported by the supplied app
specification or trace evidence. Do not blindly repeat stale guidance.

Return STRICT JSON with exactly one key and no other prose:
{"reflection": "<one concise paragraph, at most 180 words>"}
```

E.6 Admitted Trace Representation

After clean re-collection and replay verification, each admitted behavior is stored as a `TraceSpec`. The `TraceSpec` records the action sequence from the clean run with replay-stable selectors, the observation-grounded expected signals, and an optional reference to its immediate parent trace. It also stores a `trace_hint` summarizing the latest successful route for use as guidance in subsequent collection. Replay verification ignores this hint and depends only on the recorded actions, parent structure, and expected signals.

An admitted child `TraceSpec`: renaming a board list (abridged)

```
{
  "name": "rename_done_list_to_completed",
  "candidate_id": "vdevired_trello_clone",
  "description": "Renamed the Done list on the Workflow Pipeline board to Completed.",
  "parent_trace": "create_board_with_ordered_lists",
  "action_trace": [
    {
      "action": "click",
```

```
      "selector": "role=button[name=\"Rename list Done\"]"
    },
    {
      "action": "type",
      "selector": "[name=\"title\"]",
      "text": "Completed"
    },
    {
      "action": "press",
      "selector": "[name=\"title\"]",
      "key": "Enter"
    }
  ],
  "expected_signals": {
    "all_accessible_names": [
      "Completed Rename list Completed"
    ],
    "reject_visible_text": [
      "Done"
    ]
  }
}
```


The parent trace creates and opens the `WorkflowPipeline` board with the `Done`, `Ideas`, and `InProgress` lists. After reset and replay of the parent chain, the child performs three actions to rename `Done` to `Completed`. Its success signals check the renamed list's accessible name and the absence of the old `Done` text. All three recorded actions are shown; the route hint, empty signal fields, and redundant execution metadata are omitted.

E.7 Trace Evolution Across Mining Stages

Table 7 summarizes how a proposed behavior is refined from an exploratory goal into a replay-verified trace.

Table 7: How a candidate trace evolves through the mining pipeline.

Stage	Input	Output
Planning	source and UI evidence, current coverage, known failures, available parent traces, and prior reflections	a proposed behavior goal with a grounded route hint and an optional `parent_trace`
Discovery	the proposed goal and the live state established by its parent chain, if any	an exploratory action sequence, selected success evidence, and a route hint derived from the successful execution
Clean re-collection	the previously successful route and a clean state with the parent chain replayed	a more direct action sequence, newly selected success evidence, and a `trace_hint` regenerated from the clean execution
Replay verification	the clean action sequence, parent structure, and expected signals	a deterministic pass/fail result after reset and replay of the required parent chain and target actions
Relabeling	the replay-verified actions, success evidence, and final observation, without the original proposed goal	a name and description reflecting the behavior actually achieved

F CRAFTING AGENT PROMPTS

Crafting masks the source implementing a verified behavior using two LLM roles, the task generator and a stateless mask-depth critic. The generator constructs the counterfactual failures by directly editing the source code, while the critic judges whether the resulting diff removes substantive implementation or only disables it superficially. The system prompts used for the generator and critic are shown below.

F.1 TASK GENERATOR

The system prompt is assembled from common masking rules and one scope-specific clause. In each task, the generator additionally receives the app description, trace actions, expected signals, and, on a repair round, the concrete verification failure from the previous attempt.

TaskGenerator system prompt (abridged)

```
You are TaskGeneratorAgent for ProgramDistill. You build feature-
restoration coding tasks from a verified golden behavior trace of a
real OSS web app.

Your job: delete-mask every piece of source that implements that
feature, so the feature is broken, while the application still builds
and boots.

Masking rules:
- Remove the implementation rather than disabling the feature. Delete
the substantive serializer, generator, query, render, handler, export,
 or state-update logic that produces the behavior observed in the
trace. Restoration must require rewriting real logic.
- Reject shallow masks such as changing a feature flag, permission,
configuration, environment value, enum, or allowlist; adding or
editing a guard or early return; or disabling a call site while its
implementation remains intact. If reverting one switch or line
restores the feature, remove the deeper implementation instead.
- Keep the module compilable. Preserve imports, exports, declarations,
 signatures, and route registrations. Replace only substantive bodies
with minimal inert stubs such as an error, 'return null', a 501
response, 'pass', an empty value, or a no-op handler.
- Mask every applicable feature-owned surface, including the frontend
component, API endpoint or route, and service or model logic. Leaving
any implementation path intact makes the task invalid.
- For full UI + logic restoration, remove the feature-owned rendered
markup as well as its handlers. Replace forms, controls, sections,
views, or lists owned by the feature with an inert placeholder while
preserving the export and route so the app still boots. Leave
unrelated layout, navigation, and surrounding UI intact.
- Do not touch unrelated features, shared utilities used broadly, auth/
bootstrap needed just to reach the feature, or the build config.

Verification you are designing for (inverse of validate_instances):
after your mask_patch is applied in a dev pod, the app still boots,
but the golden trace that used to pass must now fail because the
feature is gone.

Workflow: explore with search/read_file/list_dir using the trace's
source hints and UI steps; delete the real implementation with
edit_file (old_str must be unique); review with view_changes; then
submit_task with a solver-facing problem_statement; then finish.

problem_statement rules: state only what the feature is -- its user-
facing purpose and workflow and which surfaces it spans -- anchored to
 the feature's own name/description from the trace. Do not state the
```

```
observable outcomes that prove it works (on-screen text, toast/
validation/error messages, counts, generated identifiers/filenames,
URLs, or request/response shapes): those are the verification contract,
 and the solver must discover them empirically by driving the working
feature in the reference app through the browser helper. Do not list
the masked files/paths, and never include function names, code, or a
step-by-step fix. Scale length to the feature: one sentence for a
single surface, a brief per-surface list for a multi-surface or
asynchronous one.
```

F.2 MASK-DEPTH CRITIC

Judging feature deletion from behavioral failure alone is insufficient, as a one-line feature-flag change can also make a trace fail by "turning-off" code segments. The critic therefore receives the feature description, success signals, solver-facing statement, and production-to-masked diff. The critic fails the generated diff unless restoration requires rewriting substantive logic.

MaskDepthCritic system prompt

```
You are MaskDepthCritic for ProgramDistill. A task-generator agent was
 asked to build a feature-restoration coding task by deleting the real
 implementation of one feature from an app's source, so a solver must
rewrite that logic to restore the feature. You are given the feature
description, its success signals, and the unified diff the generator
produced (prod -> masked: what it removed/changed).

Your only job: decide whether the diff genuinely deletes the feature's
 implementation, or merely disables it shallowly.

A mask is deep (good) when the substantive logic that produces the
feature's observable behavior is gone -- e.g. the serializer/query/
render/export builder/state-update/handler body is removed and
replaced by an inert stub that returns empty/placeholder data or
throws. Restoring the feature would require writing real logic from
scratch.

A mask is shallow (bad) when the implementation is left intact and the
 feature is only switched off, for example: flipping a boolean /
permission / capability / feature flag ('return True'->'return False',
 'canExport = false'); changing a config / env / enum / allowlist
value; adding or editing an early-return, guard, or 'if' condition; or
 commenting out a single call site while the function it calls still
exists. If a developer could restore the feature by reverting a single
 flag/guard/line without writing new logic, the mask is shallow.

UI features: the intended scope for this mask is full restoration (UI +
 logic). So a mask is also shallow if it guts only the event handlers /
 submit / API logic but leaves the feature's rendered markup (the form
 fields, buttons, view the user interacts with) fully intact. A deep
mask must remove the feature's returned JSX/template markup as well as
 its logic (a still-rendered, still-interactive form with dead
handlers is shallow).

Respond with only a JSON object, no prose, no code fences:
{"verdict": "deep" | "shallow", "reason": "<one or two sentences>", "
restoration_requires_rewrite": true | false}
```

F.3 CUMULATIVE MASK MERGER

The cumulative mask merger is invoked when atomic masks conflict while being applied to the same source baseline. Conflicts arise when multiple masks modify overlapping lines. The merger begins

from a partial merge in which all non-conflicting masks have already been applied, leaving only the conflicting components to resolve.

For each conflicting mask, the merge agent receives the conflicting region, the mask's original diff and problem statement, and the masked source produced by that mask independently. Using the same source-editing tools as the task generator, it modifies the partial merge until every mask component is preserved and submits a resolution for each conflict. If validation rejects the submission, the agent receives the failure and revises the merge.

Among the 1,201 cumulative tasks, 629 masks compose deterministically and 572 require the merge agent. All 572 agent-assisted merges succeed without omitting an intended mask component. The fraction requiring agent-assisted merging rises with restoration depth, from 25.6% at depth 2 to 89.7% at depth 8 and 100% beyond depth 8.

Cumulative merger system prompt

```
You are a conservative conflict merge agent for benchmark masking.
Large certified inputs live in workspace artifacts: read only what you
 need. Resolve overlaps in the seeded source with the smallest
coherent union of removals. Never invent implementation or alter
unrelated code. Success requires submit_merge.
```

Cumulative merger user message

```
Resolve the cumulative mask conflict in the seeded in-memory source
workspace. Read the artifact manifest first with read_file:
<manifest_path>

The real production baseline is available at the normal source paths.
Those paths currently expose a branch-local seed that already
preserves the newest and all non-conflicting masks; the manifest
identifies the components missing from that seed. Start with
priority_component_ids; components outside that set are already
certified in the seed and do not need inspection unless validation
feedback says otherwise. Inspect only the conflict ranges,
authoritative patches, problem statements, and certified variants you
need. Edit the normal source paths to add every missing masking intent
 without restoring any included component or changing unrelated code.
Then call submit_merge with every expected component exactly once and
one resolution per component. Do not paste complete source files into
the response. If validation_feedback is present, correct it.
```

G PARTIAL- AND FULL-APPLICATION RECONSTRUCTION INSTRUCTIONS

For both partial-application and full-application reconstruction, the agent receives one system message and one user message at the start of each task. The system message is fixed, while the user message is instantiated from a fixed template. Its `{problem_statement}` slot is replaced by the task instruction, which combines the accepted problem statement with fixed runtime guidance. The `{tool_call_instruction}` slot selects the single- or parallel-tool-call variant used for the model under evaluation. Accordingly, the first two listings below show the system and user messages, while the third shows the task instruction inserted into the `{problem_statement}` slot of the user message.

G.1 PARTIAL-APPLICATION RECONSTRUCTION

Partial-application reconstruction system prompt

```
You are a software repair agent restoring behavior intentionally
removed from a full application repository. You have a masked current
app and a live reference app that is the behavioral oracle. Use the
reference empirically, make the correct source change, and verify the
restored behavior in the current app.
```

Partial-application reconstruction user message

```
The full application repository is available at {repo_path}.

<programdistill_task>
{problem_statement}
</programdistill_task>

Restore the removed behavior in non-test application source files only.
 Do not
modify generated build output, runtime state, logs, browser artifacts,
dependencies, caches, or harness/tooling files.

Work in this order:
1. Explore the repository to locate the source that implements the
feature, reading only the surrounding code needed to understand
project conventions.
2. Before editing, exercise the feature in the reference app with the
'browser' CLI.
3. Compare the current app where useful, then implement a focused
repair in the source you identified.
4. Drive the repaired workflow in the current app and compare its
observable behavior with the reference.
5. When the repair is complete, submit it using the 'finish' tool.

Restore the task description and every relevant observable reference
behavior
completely and accurately, while avoiding unrelated or out-of-scope
changes.

{tool_call_instruction}
```

Partial-application reconstruction task instruction (injected at `{problem_statement}`)

```
You are working in the full <repository> repository at <repo_path>.
The application is <app_description>.

Restore <feature_description>.
```

```
Source-change boundary: modify only application source files needed for
 this repair. Do not edit generated build output, runtime state, logs,
 browser artifacts, dependencies, caches, or harness/tooling files. Do
 not intentionally trigger or include those files as part of the
solution. In this application, excluded path prefixes include: <
excluded_path_prefixes>.

Use the surrounding source files to infer the full expected behavior
and match the existing project conventions.

Reference behavior via the 'browser' command (IMPORTANT):
A live browser with TWO pages is available through 'execute_bash'
using the 'browser' command:
- 'reference': the FULL, working app with the target feature intact.
Use it as the ground-truth oracle for how the feature should look and
behave. You can observe and interact with it, but you CANNOT read its
source or shell -- only its rendered behavior.
- 'current': the app in THIS repo (the one you are editing), with the
feature removed. Use it to reproduce the broken state and to confirm
your fix restores the behavior.

Recommended workflow: (1) 'browser observe reference' to see the
correct behavior and the exact UI/text/state it produces; (2) drive
the reference through the feature's steps (click/type) to learn the
full flow; (3) restore the implementation in the source; (4) 'browser
observe current' and re-drive it to verify it now matches the
reference. Always re-'observe' a page after every interaction --
element ids are assigned by the latest observation.

Commands (each is one execute_bash call):
- 'browser observe <current|reference>' -- render a page; returns
visible text, url, title, and interactable elements with numeric ids.
- 'browser observe-both' -- observe both pages at once (side-by-side).
- 'browser click <page> <elementId>' -- click an element by its id
from the latest observe.
- 'browser type <page> <elementId> <text>' -- type into a field.
- 'browser hover <page> <elementId>' / 'browser focus <page> <
elementId>' / 'browser press <page> <key>' / 'browser scroll <page> <
up|down>' / 'browser wait <page> [ms]' / 'browser back <page>' / '
browser forward <page>'.
- 'browser upload <page> <elementId> <localFile>' -- upload a local
file through a file input from the latest observation.
- To exercise a drag-based feature (reordering a list, moving a kanban
 card), either focus the drag handle and use press (Space to lift,
Arrow keys to move, Space to drop), or 'browser drag <page> <elementId>
 <targetElementId>' to drag one element onto another.
Example: 'browser observe reference', then 'browser click reference
12', then 'browser observe reference'.

Evaluation replays the recorded user workflow against the current app
and checks whether the reference app's observable behavior has been
restored completely and accurately.

Command tip: each action is executed through a timeout wrapper. For
commands that change directories or use shell operators, wrap the
command explicitly, e.g. 'bash -lc 'cd /workspace && ...''.
```

G.2 FULL-APPLICATION RECONSTRUCTION

Full-application reconstruction system prompt

```
You are implementing a complete web application in a minimal starter
repository. Use the application specification and the live reference
application to understand the expected behavior. Explore the reference
 as needed, implement the application in the current workspace, and
compare the result with the reference to validate your work.
```

Full-application reconstruction user message

```
The blank application scaffold is available at {repo_path}. Build the
complete
application there without attempting to recover hidden source, build
output,
caches, installed artifacts, or harness and tooling files.

<programdistill_task>
{problem_statement}
</programdistill_task>

Implement the application in non-test application source files only.
Do not
modify generated build output, runtime state, logs, browser artifacts,
dependencies, caches, or harness/tooling files.

Work in this order:
1. Systematically explore the reference app with the 'browser' CLI,
including its routes, controls, state transitions, validation, and
secondary functionality.
2. Implement the observed behavior in the blank scaffold.
3. Exercise the same workflows in the current app and compare their
observable results with the reference.
4. Repeat exploration and implementation until the complete product
specification and all additional discoverable behavior are covered.
5. When the reconstruction is complete, submit it using the 'finish'
tool.

Reproduce every relevant observable reference behavior completely and
accurately while keeping harness and evaluator internals out of the
submission.

{tool_call_instruction}
```

The task-instruction template uses placeholders for the repository path and product specification. Tasks requiring sign-in additionally include a reference-access block with seeded login instructions.

Full-application reconstruction task instruction (injected at `{problem_statement}`)

```
Rebuild the complete application from the provided blank scaffold.

No current application implementation is available. Do not attempt to
recover it from caches, installed artifacts, build outputs, or harness
and tooling files.

## Browser environment

A live browser with two pages is available through the 'browser'
command:
```

```
- 'reference': the complete, working original application. Use it as
the
  ground-truth oracle for the application's observable information and
 behavior. You may
  observe and interact with it, but you cannot access its source or
shell.
- 'current': the application you are implementing. Use it to exercise
your
  implementation and compare it with the reference.

Reconstruct the application from the behavior observable through the
reference browser. Faithfully match every observable aspect, including
 but
not limited to exposed text and element structure, semantic identity (
e.g., element type,
role, accessible name, and label), exposed element selectors,
navigation,
validation, and resulting state.

## Evaluation and reproduction requirements

Evaluation replays user interactions using selectors and checks the
application
state after each action. Similar text or element structure in
observations is
not sufficient. Faithfully reproduce selectors and state changes under
 the same
conditions and action sequence.

Treat exposed selectors as part of the observable interface that must
be
reproduced, not merely as reference information. In equivalent
observable
states, implement corresponding controls on 'current' so that they
expose
the same selectors as 'reference'.

Successful clicking or typing through an element ID does not establish
 selector
compatibility. A similar final result alone does not establish that
intermediate
interactions and state transitions have been implemented identically.

Reproduce intermediate screens, menus and dialogs, input validation,
success
and error messages, navigation, data changes, and saved results
throughout
the same interaction sequence.

## Exploration, implementation, and comparison

Systematically explore the entire reference application before and
during
implementation. Follow its routes and controls, exercise its
interactions
and state transitions, and re-observe after every action.

Implement what you observe, then reproduce equivalent starting states
and the
same interaction sequence on 'reference' and 'current'. Compare all of:
```

```
- Corresponding element types, roles, accessible names, labels, and
exposed selectors.
- Screens, menus, dialogs, and navigation paths immediately after each
 action.
- Input validation and success or error responses.
- Created, updated, or deleted data and its effect on subsequent
interactions.
- Saved results, downloads, and other side effects observed in the
reference.

If a corresponding control on 'current' exposes a different selector or
 no
selector, inspect the preceding state, element type, role, label, DOM
structure,
and other relevant details, then correct your implementation to match
the
reference. If behavior or state changes differ, find and fix the cause
 and
repeat the same sequence.

Element IDs belong to the latest observation of each page. Do not
reuse a
reference element ID on 'current' or reuse IDs from an earlier
observation.
Re-observe after interactions and use the latest IDs for the relevant
page.

Browser commands:
- 'browser observe <current|reference>'
- 'browser observe-both'
- 'browser recover <current|reference>' (reopen the app entry page
without resetting application data)
- 'browser reset-reference' (restore the clean seeded reference state
after exploratory mutations)
- 'browser click <page> <elementId>'
- 'browser type <page> <elementId> <text>'
- 'browser hover <page> <elementId>'
- 'browser focus <page> <elementId>'
- 'browser press <page> <key>'
- 'browser scroll <page> <up|down>'
- 'browser wait <page> [ms]'
- 'browser back <page>' / 'browser forward <page>'
- 'browser upload <page> <elementId> <localFile>'
- 'browser drag <page> <elementId> <targetElementId>'

## Product specification

Purpose: <app_purpose>.

Core capabilities:
- <core_capability_1>
- <core_capability_2>
- ...

Secondary capabilities:
- <secondary_capability_1>
- <secondary_capability_2>

Treat this product specification as required but non-exhaustive
coverage.
Faithfully implement every item above, but do not limit the
reconstruction
```

```
to those items. Discover and reproduce all additional functionality
and
user-observable behavior available through the reference application.

## Implementation contract

- Work only inside '<repo_path>'.
- 'app.js' must export 'async function handle(req, res, context)'.
- Do not call 'listen()'; the task-owned runtime owns all ports.
- Keep application state in 'context.state' so the verifier can reset
it.
- Use 'context.readBody()', 'context.json()', and 'context.html()' as
needed.
- The runtime hot-reloads repository modules on every request.
- Faithfully implement the routes, labels, forms, selectors, state
transitions,
  downloads, and validation behavior observed in the reference.
```

H Model Pricing and Execution Time

H.1 Model Pricing

Table 8 lists list price per model, in $ per 1M tokens, read from the LiteLLM `model_cost` table under the same bare model identifiers the agent harness uses (e.g. `claude-opus-5`, `gpt-5.6-sol`).

Figures 6(a) and 17(a) report mean cost per trajectory in USD. For each trajectory, we multiply the number of uncached input, cached input, and output tokens by their respective per-token prices from LiteLLM, then sum the three amounts.

Table 8: **List price per model.** Input and output price in $ per 1M tokens, from LiteLLM's `model_cost` table.

Model	Input ($/1M)	Output ($/1M)
GPT-6 Astra	10.00	50.00
Claude Opus 5	5.00	25.00
GPT-5.6 Sol	4.00	20.00
GPT-5.3 Codex	1.75	14.00
Gemini 3.1 Pro Preview	2.00	12.00
Claude Sonnet 5	2.00	10.00
Grok 4.6	2.00	6.00
Gemini 3.6 Flash	0.75	3.75
Gemini 3.7 Flash	0.75	3.75

H.2 Execution Time

To give a concrete sense of the time required to run the benchmark, we report one representative evaluation run. GPT-6 Astra at high reasoning effort completed all 300 tasks in approximately 557 minutes of wall-clock time (9 hours 17 minutes) using four CPU servers. Each server had two Intel Xeon Gold 6238R processors at 2.20 GHz (56 physical cores, 112 logical CPUs) and ran up to eight trials concurrently, for a maximum of 32 concurrent trials. Each trial was allocated six logical CPUs, split evenly between the current and reference applications, with verification reusing the current application's allocation. The servers hosted the applications, browsers, and verification processes, while model inference was served through a remote API.

Wall-clock time is measured from the earliest trial start to the final trial completion and includes per-trial setup, agent execution, verification, and API waiting time, but excludes one-time environment provisioning. Individual trials took 48.2 minutes on average, with a median of 43.1 minutes. This

example is intended to provide a practical estimate of benchmark runtime rather than a measure of standalone model-inference speed.

I More Experimental Results

I.1 Results by Mask Scope

Every task in ProgramDistill-300 carries one of two mask scopes. A logic-only task removes the behavior behind an interface that remains unchanged, so the agent restores handlers, endpoints, queries, and state updates behind existing controls. A logic-and-UI task also removes the UI owned by the feature, requiring the agent to reconstruct both the observed interface and its underlying behavior. Table 9 compares performance across these two settings using binary and chain scores.

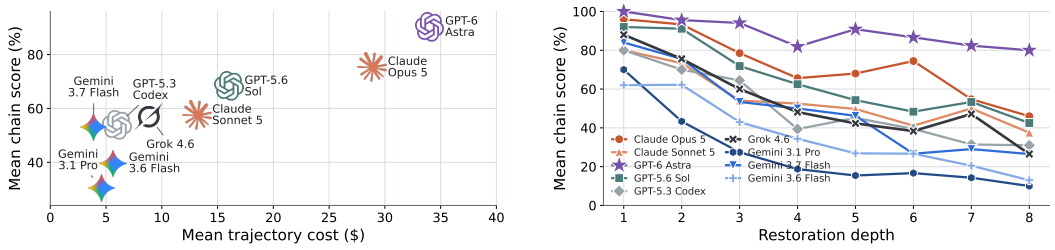
Astra achieves 92.9% binary success on logic-only tasks and 76.9% on logic-and-UI tasks, a 16.0-point gap. The chain score gives partial credit for behaviors recovered before the first failure in a cumulative task. Under this metric, Astra achieves 96.2% on logic-only tasks and 84.9% on logic-and-UI tasks, an 11.3-point gap. The scope comparison therefore captures both complete task success and the extent of partial restoration.

Table 9: **Performance by mask scope on ProgramDistill-300.** Logic-only tasks mask the behavior behind an unchanged interface, whereas logic-and-UI tasks also remove the UI owned by the feature. The suite contains 140 logic-only and 160 logic-and-UI tasks. Δ denotes the difference between logic-only and logic-and-UI performance.

	Binary score				Chain score			
Model	**Overall**	**Logic-only**	**Logic-and-UI**	**Δ**	**Overall**	**Logic-only**	**Logic-and-UI**	**Δ**
GPT-6 Astra	84.3	92.9	76.9	16.0	90.2	96.2	84.9	11.3
Claude Opus 5	68.7	80.0	58.8	21.2	75.2	85.7	66.0	19.7
GPT-5.6 Sol	60.7	76.4	46.9	29.6	68.2	81.5	56.5	25.0
Grok 4.6	48.3	64.3	34.4	29.9	57.1	72.7	43.5	29.2
Claude Sonnet 5	47.3	65.0	31.9	33.1	57.5	73.9	43.1	30.8
GPT-5.3 Codex	45.7	59.6	33.4	26.2	53.7	67.4	41.7	25.7
Gemini 3.7 Flash	45.3	61.4	31.2	30.2	53.2	69.5	38.9	30.6
Gemini 3.6 Flash	29.7	37.9	22.5	15.4	39.6	49.6	30.9	18.7
Gemini 3.1 Pro Preview	22.3	31.1	14.7	16.4	30.5	41.6	20.8	20.8

I.2 Results Under the Chain Score

The binary score gives credit only when all targets in a cumulative task pass, while the chain score credits the fraction recovered before the first failure. We compare these scores for the two evaluation settings.



(a) Cost vs. mean chain score

(b) Mean chain score by restoration depth

Figure 17: **Partial-application reconstruction performance under the chain score.** The chain-score counterpart of Figure 6. Results cover all 300 tasks in `ProgramDistill-300`, including both atomic and cumulative tasks. **(a)** Mean chain score versus mean trajectory cost (USD), using the same cost aggregation as Figure 6(a). **(b)** Mean chain score by restoration depth r_L. The decline with depth is shallower than under the binary score because partial restoration still earns credit, but the overall depth effect remains.

Partial-Application Reconstruction. For the setting in Section 4.2, Figure 17 is the chain-score counterpart of Figure 6, covering all 300 tasks in `ProgramDistill-300`. Table 10 compares

Table 10: **Partial-application reconstruction under binary and chain scores.** Mean scores (%) over the 250 cumulative tasks of `ProgramDistill-300`.

Model	Binary score	Chain score
GPT-6 Astra	81.2	88.2
Claude Opus 5	63.2	71.1
GPT-5.6 Sol	54.4	63.4
Claude Sonnet 5	40.8	53.0
Grok 4.6	40.4	50.9
GPT-5.3 Codex	38.8	48.4
Gemini 3.7 Flash	37.6	47.0
Gemini 3.6 Flash	23.2	35.2
Gemini 3.1 Pro Preview	12.8	22.6

binary and chain scores on the 250 cumulative repair tasks. Astra achieves 81.2% binary and 88.2% chain score. The gap reflects cases where part of the restoration lineage is recovered without a complete task pass. Opus 5 increases from 63.2% to 71.1%, and Sol from 54.4% to 63.4%.

Full-Application Reconstruction. For the setting in Section 4.3, Table 11 reports chain scores in the same model and application order as Table 1. Mean binary/chain scores are 49.15/64.58% for Astra, 28.81/47.42% for Opus 5, and 21.07/34.22% for Sol. The larger gaps between the scores indicate that agents recover workflow prefixes more often than complete workflows.

Table 11: **Full-application reconstruction under the chain score (%).** The chain-score counterpart of Table 1. The Mean column averages all evaluated cumulative workflows, and bold indicates the best result among the three models.

Model	Reactive Resume	AWS Console	Baserow	BudgetWise	CareerLink	MailHub	Jira (Oldboyxx)	Trello (Knowankit)	StreamView	TeamChat	Trello (Vdevired)	Weights & Biases	Mean
GPT-6 Astra (max)	**68.5**	**45.8**	**36.7**	**56.0**	**65.3**	89.0	85.6	73.2	**91.4**	**47.0**	**44.3**	**80.9**	**64.58**
Opus 5 (max)	43.7	35.4	35.3	39.8	41.4	**93.9**	**93.3**	73.9	28.0	42.9	29.6	35.3	47.42
GPT-5.6 Sol (max)	16.8	8.3	29.0	21.5	12.4	82.0	87.5	**76.7**	32.2	35.6	42.6	9.1	34.22

I.3 Does Context Length Explain the Depth Effect?

Deeper tasks generally expose agents to longer contexts, raising an alternative explanation for the performance decline in Section 4.2. Models may perform worse simply because they operate over longer contexts rather than because they must compose more dependent repairs. We examine this possibility in three ways using mean prompt size per step as a measure of context exposure across trajectories from all nine models.

First, the growth in context length does not closely track the decline in performance. The median prompt size per step increases from 47.5k tokens at restoration depth 1 to 107.5k at depth 8, a factor of 2.3, while the mean binary score falls from 83.4% to 23.1%. Most of the increase in context length occurs by depth 4. Beyond that point, context length changes little while performance continues to decline.

Table 12: **Mean binary score by restoration depth among runs with similar context lengths.** Runs are grouped by their mean prompt size per step, and columns indicate restoration depth. This diagnostic uses the task outcomes paired with the analyzed trajectories.

	Restoration depth							
Mean prompt size / step	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
40–80k	82.6	77.1	65.0	51.8	48.1	46.4	45.3	24.2
80–120k	77.0	58.9	43.4	39.5	32.5	30.1	30.0	22.5
120–200k	54.5	68.8	33.9	33.3	30.0	25.4	23.2	23.1

Second, performance still declines with restoration depth among runs with similar context lengths. Within each context band in Table 12, binary score shows an overall downward trend as depth

increases. From depth 1 to depth 8, mean binary score drops by 58.4 points in the 40–80k band, 54.5 points in the 80–120k band, and 31.5 points in the 120–200k band. Thus, deeper tasks remain substantially harder even at comparable context lengths.

Third, within each restoration depth, prompt size has only a modest negative Pearson correlation with binary score, ranging from -0.20 to -0.05. After centering both prompt size and binary score within each task to account for task differences, the pooled correlation is weakly positive ($+0.11$).

Together, these analyses show that the depth-related performance decline cannot be explained by longer contexts alone and are consistent with the increasing difficulty of composing multiple dependent repairs.

I.4 Agent Interaction Statistics

Partial-Application Reconstruction. Table 13 provides the full interaction statistics supporting Figure 7 in Section 4.2.1. It reports the minimum, mean, and maximum numbers of agent steps, observation steps, and edit/write steps for each model. The lower panel groups the same trajectories by restoration depth and reports means across all nine models.

Table 13: **Agent interaction behavior.** The upper panel reports min, mean, and max interaction statistics for each model on `ProgramDistill-300`. The lower panel reports per-trajectory means by restoration depth, pooling all nine models. An observation step is an agent step that requests application state through an `observe` command, a state-returning browser interaction, or a supported direct browser or API check of the current app.

	Steps			Observation steps			Edit/write steps		
Model	Min	Mean	Max	Min	Mean	Max	Min	Mean	Max
GPT-6 Astra	42	229.2	757	35	191.3	581	1	9.9	58
Claude Opus 5	29	209.3	1000	1	95.3	329	0	23.5	117
Claude Sonnet 5	27	222.4	522	8	107.2	364	1	15.7	53
GPT-5.6 Sol	42	199.4	377	15	101.2	236	1	25.8	85
GPT-5.3 Codex	58	196.4	342	11	86.7	195	0	17.1	60
Grok 4.6	41	160.9	370	2	65.9	173	1	18.5	69
Gemini 3.1 Pro	35	159.6	858	2	46.3	152	1	24.2	106
Gemini 3.7 Flash	50	206.8	626	2	82.5	218	1	13.5	59
Gemini 3.6 Flash	55	180.2	386	9	68.9	184	0	16.8	62

Restoration depth	**Mean steps**	**Mean observation steps**	**Mean edit/write steps**
1	124.0	61.2	7.5
2	164.0	75.9	12.4
3	187.0	92.8	17.0
4	216.0	99.9	21.2
5	225.9	110.5	23.0
6	230.8	110.5	24.3
7	235.7	110.8	24.5
8	250.6	121.0	27.2

Full-Application Reconstruction. Full reconstruction produces substantially longer trajectories than partial reconstruction, with mean trajectory lengths around 700 agent steps and a maximum of 1,921. Table 14 reports per-run agent steps, observation steps, and browser state returns. Observation steps count agent steps that request application state, whereas browser state returns count individual helper responses, including those returned by failed actions. A single agent step can therefore produce multiple browser states.

Opus often issues multiple browser CLI calls within a single shell step, yielding 7.9 browser state returns per agent step on average, compared with 1.6 for Astra and 1.2 for Sol. Despite receiving substantially more browser feedback in this form, Opus recovers fewer behaviors than Astra. Observation steps and browser state returns therefore capture different aspects of agent interaction, and feedback volume alone does not explain reconstruction performance.

Table 15 reports agent steps by application for the full-application runs in Table 1. Astra averages 719.4 steps, compared with 627.5 for Opus 5 and 719.0 for Sol.

Table 14: **Agent activity during full reconstruction.** Per-run min, mean, and max counts. Observation steps count agent steps that request application state. Browser state returns count individual helper responses, including those from failed actions, so one agent step can return multiple states.

	Steps			Observation steps			Browser state returns		
Model	Min	Mean	Max	Min	Mean	Max	Min	Mean	Max
GPT-6 Astra (max)	292	719.4	1921	221	643.3	1796	406	1154.1	3843
Opus 5 (max)	371	627.5	940	59	258.2	557	1427	4932.1	12821
GPT-5.6 Sol (max)	348	719.0	1165	204	419.3	705	277	854.7	1441

Table 15: **Agent steps per full-application reconstruction run.**

Model	Reactive Resume	AWS Console	Baserow	BudgetWise	CareerLink	MailHub	Jira (Oldboyxx)	Trello (Knowankit)	StreamView	TeamChat	Trello (Vdevired)	Weights & Biases	Mean
GPT-6 Astra (max)	853	1921	923	292	824	421	489	434	895	385	700	496	719.4
Opus 5 (max)	620	837	572	663	940	417	371	595	601	608	533	773	627.5
GPT-5.6 Sol (max)	1165	856	860	580	861	348	598	446	990	488	470	966	719.0

I.5 Example of Successful Repair at Depth 8

Although performance declines with restoration depth, some deep tasks are recovered through sustained iterative refinement. Figure 18 shows a depth-8 `Trello(Vdevired)` task repaired by Opus 5 over 317 steps. The trajectory combines reference and repository inspection with targeted edits and repeated validation of residual mismatches, illustrating sustained observe–edit–validate cycles across dependent behaviors.

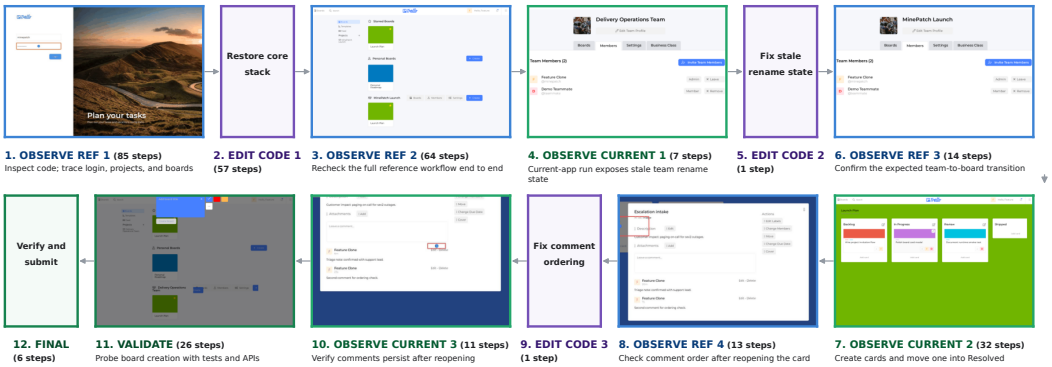


Figure 18: **Successful depth-8 repair in `Trello(Vdevired)`.** Eight stages span login, project setup, team/board setup, list/card creation, permissions, card details, cross-list movement, and comment editing. Across 317 steps, Opus alternates between reference observation, code editing, and current-app observation.

I.6 Analysis of Shortcut Attempts

Agents sometimes request implementation information through channels outside the intended reference-guided repair workflow. We identify requests involving Git history, public application source, protected harness or evaluator files, and direct reference access. At least one such request occurs in 7.2% of partial-reconstruction trajectories.

Rates vary substantially across models. Grok has the highest rate at 19.0%, followed by Gemini 3.1 Pro at 17.7%, while Astra has a rate of 0.3%. Figure 19 reports the breakdown by access channel and model.

The evaluation environment blocks these channels by removing pre-mask Git history, regenerating stale artifacts, disabling public egress, protecting grading files, and restricting reference access to the browser helper. Appendix D.1 describes the corresponding runtime isolation. These measurements capture access requests rather than successful recovery of hidden information or evidence of intent.

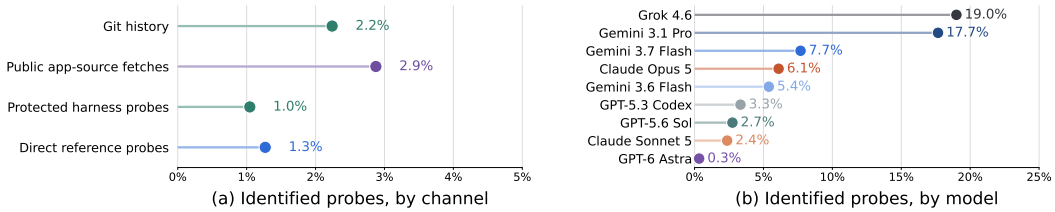


Figure 19: **Identified shortcut attempts.** (a) Trajectory share with a request in each channel. (b) Within-model share with any request. Channels may overlap, and each trajectory counts once in (b).